Rules in IdentityIQ

IdentityIQ Version: 7.0, 7.1, 7.2

This document describes the usage and writing of rules to implement custom logic in the IdentityIQ product.

The menu navigations changed in version 7.0, so this new document has been created from the old Rules white paper as of version 7.0. The document content for the rules which exist in all of the 6.x and 7.x IdentityIQ versions has not changed; only the navigation instructions are different. Some rule types, as noted, only exist in the newer product versions and are therefore only found in this document version.

Document Revision History

Revision Date	Written/Edited By	Comments
Nov 2016	Jennifer Mitchell	Split off from old Rules white paper and updated for 7.0, 7.1; changed UI navigation instructions to rule editor locations and added rules for SAP HR and PeopleSoft HRMS aggregation/provisioning and before/after operation rules for WebServices connector
Dec 2017	Jennifer Mitchell	Edited for 7.2: Added Alert-related rules and made some error corrections (argument type in WebServiceAfterOperationRule rule, change to FallbackWorkItemForward rule details because of change to allowSelfCertification variable (changed in 6.1)) Added previously-missed rules: Report-related rules, TargetCustomization, RACFPermissionCustomization, PolicyNotification

Table of Contents

Rules Overview

Rules are the construct through which IdentityIQ allows the addition of custom business logic at specific points within the execution flow of the product. Rules are written in BeanShell, a lightweight scripting language based on Java.

This guide describes how to create rules and associate them with system activities. It discusses each type of rule available in IdentityIQ, explains the general usage of the rule type along with its input and output arguments, and provides examples of how to implement each rule type.

Creating Rules

Rules are created within IdentityIQ in one of two ways:

1) Through the UI Rule Editor
2) By importing rule XML objects

UI Rule Editor

Rules are associated with system activities on a variety of pages throughout the IdentityIQ user interface. At these points, an existing rule can be attached to the activity, the Rule Editor can be opened to write a new rule, or an existing rule can be opened and edited in the Rule Editor. Each of these rule selection boxes allows only rules of a prescribed type to be created for or associated to the given activity.

To open the Rule Editor and create a new rule, ensure that no rule is selected and click [...].
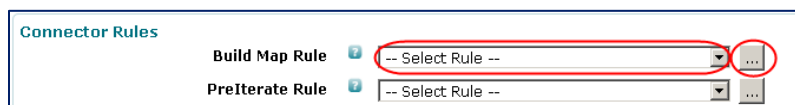
Figure 1: Create new rule

To associate an existing rule with the system activity, select the rule from the list.
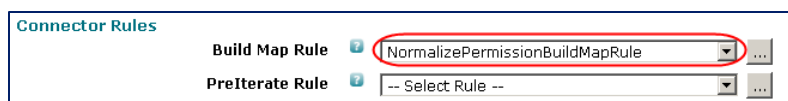
Figure 2: Connect rule to object

To edit an existing rule in the Rule Editor, select the rule from the list and click [...].
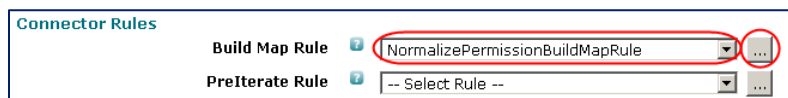
Figure 3: Edit existing rule

NOTE: A single rule can be reused in many places throughout the product; for example, two applications could share the same Build Map Rule. Changes made to the rule will affect the functionality in all locations where it is

used, so if the functionality varies slightly between usages, separate rules must be created for each functional need.

When a rule is opened or newly created in the rule editor, the editor displays the current content of the rule (or none, in the case of a new rule). It displays (in the panel to the right) the name of the rule, its type, its return type, and its arguments. Though that right-hand panel also declares a return value in the Returns section, the specified variable name is simply a placeholder and does not have to be used in the rule. In fact, the variable name listed there is not actually available for use in the rule until it is specifically declared in the rule's BeanShell code.
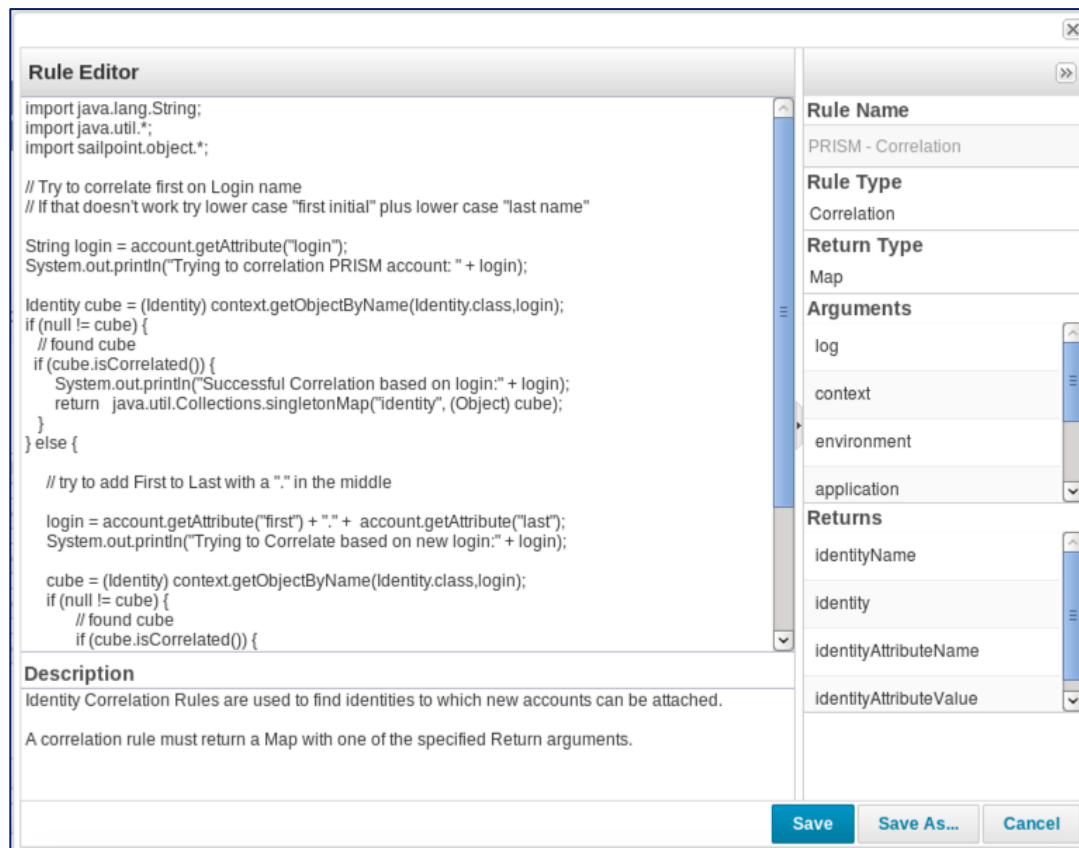
Figure 4: Rule Editor

Importing Rule XML

Rules can be written as standalone XML objects and loaded into IdentityIQ through the GUI importer (Gear menu -> **Global Settings** -> **Import from File**), or via the IdentityIQ console (**iiq console**, **import** command). A single rule XML file can contain one or more rules to be loaded at one time. See *Appendix A: Loading Rules* for steps to import rules.

Once in the system, these rules can be associated to activities through the user interface (as describe above). Alternatively, the XML objects that drive execution of those rules (applications, certifications, tasks, etc.) can be edited directly to reference rules; in fact, a few rule types (as noted in this document) can *only* be connected to

objects through the XML because no UI options currently exist for specifying them. In the XML, some rules are pointed to through attributes map entries while others are connected by references.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE Rule PUBLIC "sailpoint.dtd" "sailpoint.dtd">
<Rule created="1347913648799" id="4028460239d5e7f80139d5ea0a9f025d" language="beanshell" name="NormalizePermissionBuildMapRule" type="BuildMap">
  <Source>
    import sailpoint.tools.xml.XMLObjectFactory;
    import sailpoint.connector.DelimitedFileConnector;
    import sailpoint.object.Permission;

    Map map = DelimitedFileConnector.defaultBuildMap(cols, record);
    ... [rest of beanshell code appears here]
    return map;

  </Source>
</Rule>
```

Figure 5: Rule object XML

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE Application PUBLIC "sailpoint.dtd" "sailpoint.dtd">
<Application connector="sailpoint.connector.DelimitedFileConnector" created="1347913656253" featuresString="PASSWORD, DIRECT_PERMISSIONS, NO_RANDOM_ACCESS,
  DISCOVER_SCHEMA" id="4028460239d5e7f80139d5ea27bd0286" modified="1349105731053" name="Active_Directory" profileClass="" type="Delimited File Parsing Connector">
  <Attributes>
    <Map>
      <entry key="buildMapRule" value="NormalizePermissionBuildMapRule"/>
      ...
```

Figure 6: Application XML naming "NormalizePermissionBuildMapRule" as its Build Map Rule (by map entry)

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE Application PUBLIC "sailpoint.dtd" "sailpoint.dtd">
<Application connector="sailpoint.connector.DelimitedFileConnector" created="1350329365653" featuresString="DIRECT_PERMISSIONS, NO_RANDOM_ACCESS, DISCOVER_SCHEMA"
id="402846023a65e596013a65e6f4950281" modified="1350401850066" name="Active_Directory" profileClass="" type="Delimited File Parsing Connector">
  ...
<CorrelationRule>
  <Reference class="sailpoint.object.Rule" id="402846023a65e596013a65e6d881025a" name="Platform Correlation Rule"/>
</CorrelationRule>
```

Figure 7: Application XML referencing "Platform Correlation Rule" as its Correlation Rule (by reference)

Common Rule Arguments

All rules are universally passed two input parameter objects in addition to the rule-specific input parameters listed for each individual rule. The common parameters are used for logging and for querying the IdentityIQ database, respectively. They are:

Argument	Type	Purpose
log	org.apache.log4j.Logger	Can be used for logging from within rules Use any of the log4j methods to log messages; e.g.: log.debug(), log.trace(), log.info(), etc.
context	sailpoint.api.SailPointContext	Provides a starting point for using the SailPoint API. From this context, the rule can interrogate all aspects of the IdentityIQ data model including: • Finding Identities, Identity Attributes • Finding Accounts, Account Attributes • Finding Roles, Role Attributes

Custom Log4J Logging in Rules

Though the **log** parameter is available by default from within any rule, its logging level is set according to the logging level of the code that invokes it. Turning up logging (e.g. to the debug or trace level) in some system components can result in a large volume of messages being generated, so the ability to adjust logging for the

rule by itself can be very helpful in the debugging process. This targeted logging adjustment is possible through creation of a custom logger that applies only to the single rule. Creating a custom logger for a rule involves adding a custom logger entry in the log4j.properties file and adding a custom logger object to the rule code.

In the log4j.properties file, create a custom logger using this naming convention and designate the desired logging level:

> log4j.logger. [uniquename].[ruleName]=[loglevel]

> e.g.: `log4j.logger.XYZCorp.FinanceCorrelationRule=debug`

NOTE: This naming convention is recommended but is not strictly required. For example, it is possible to use the same custom logger for all rules, if desired. The only requirement is that the name in the log4j.properties file match the name specified in the rule code.

Include this code in the rule to create a logger object that uses the custom logger:

> import org.apache.log4j.Logger;
> Logger custLog = Logger.getLogger(" [uniquename].[rulename]");

> e.g.:   `import org.apache.log4j.Logger;`
> `        Logger custLog = Logger.getLogger("XYZCorp.FinanceCorrelationRule");`

Use this logger object in the rule logic to write messages of various levels to the log4J log file. The log level to which the custom logger is set in the log4j.properties file determines which messages get written to the log file.

```
custLog.fatal("This is a fatal error message.");
custLog.error("This is an error message.");
custLog.warn("This is a warn message.");
custLog.info("This is an info message.");
custLog.debug("This is a debug message.");
custLog.trace("This is a trace message.");
```

Printing the Beanshell Namespace

Though this document outlines the set of variables available in each of the rule types, sometimes a single rule type may be called from multiple places in IdentityIQ, and different arguments may apply in each context.   In those cases, the list of arguments shown in the Rule Editor and in this document may represent only the set that is universally available to rules of that type. This code snippet can be used in the rule to print all of the variables available in the beanshell namespace for the currently executing rule so they can be examined and better understood.

```
print("Beanshell namespace:");
for (int i = 0; i &lt; this.variables.length; i++) {
   String name = this.variables[i];
   // skip "transient" variable in workflow steps because
   // transient is a Java reserved word and fails eval()
   if ("transient".equals(name)) {continue;}
   Object value = eval(name);
   if (value == void)
       print(name + " = void");
   else if (value == null)
```

```
            print(name + " = null");
        else if (value instanceof sailpoint.object.SailPointObject) {
            // print XML representation if it is a SailPointObject
            print(name + "(" + value.getClass().getSimpleName() + ")=" + value.toXml());

        } else
            print(name + ": " + value.getClass().getSimpleName() + " = " + value);
    }
```

Managing Rule Arguments

Rules are often passed references to objects such as Applications or Certifications. In general, these objects should not be modified by the rule, since changes to them may be persisted to the database in subsequent steps of IdentityIQ's processing. The primary exception to this recommendation is when a rule does not return a value but instead expects one of the rule arguments to be modified in place. This document clearly notes when this is the expected behavior of a rule.

Rule Types

This section describes each rule type in detail. The rule types are grouped by the system functionality to which they relate. Some rules may fall in multiple categories and are therefore described in one category and mentioned (with a reference to the description location) in other sections. Rules that don't fall neatly into any grouping are described in the *Miscellaneous Rules* section at the end.

Connector Rules

Connector Rules are used during aggregation from specific connectors, such as (commonly) DelimitedFile, JDBC, SAP and RuleBasedFileParser. Connector rules run before Aggregation rules in the aggregation process. These rules are used to:

- implement pre-processing of data
- implement post-processing of data
- manipulate, merge or otherwise transform the incoming data as it's being read

The rules that are listed in this section run in the order they are specified here, though only some of these rules apply to certain connectors (as noted in the rule descriptions).

PreIterate

Description

A PreIterate Rule applies only to DelimitedFile and RuleBasedFileParser connectors. It is run immediately after the file input stream is opened, before all other connector rules. It can be used to execute any processing that should occur prior to iterating through the records in the file. Commonly it is used to configure global data that will be used by the other rules, unzip/move/copy files, or validate files.

PreIterate rules often work in conjunction with PostIterate rules. Sometimes actions performed in the PreIterate rule are concluded or cleaned up by the PostIterate rule, and sometimes the PostIterate rule is used to record information that will be accessed and acted upon by the PreIterate rule during the next aggregation for the application.

A PreIterate rule only runs once during an aggregation of a delimited file connector. As a result, this rule generally has a minimal impact on performance.

Definition and Storage Location

PreIterate rules are associated to an application in the UI when defining an application of type DelimitedFile or RuleBasedFileParser.

> Applications -> Application Definition -> select or create an application of Application Type: DelimitedFile -> Rules -> PreIterate Rule

> Applications -> Application Definition -> select or create an application of Application Type: RuleBasedFileParser -> Configuration -> Settings -> preIterateRule

The reference to the rule is recorded in the attributes map of the Application XML.

```
<entry key="preIterateRule" value="[PreIterate Rule Name]"/>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
application	sailpoint.object.Application	A reference to the Application object
schema	sailpoint.object.Schema	A reference to the Schema object for the delimited file source being read
stats	java.util.Map	A map passed by the connector of the stats for the file about to be iterated. Contains keys: • fileName: (String) filename of the file about to be processed • absolutePath: (String) absolute filename • length: (Long) file length in bytes • lastModified: (Long) last time the file was updated (Java GMT)

Outputs: None, usually. The rule's logic generally performs updates to objects outside of the aggregation data flow, so subsequent aggregation steps do not expect a return value from this rule.

NOTE: A preIterate rule *can* optionally return an inputStream. If it does, this new stream will replace the opened file inputStream in the remainder of the delimited file processing.

Examples

This example PreIterate rule reads data recorded in a configuration object by a previous aggregation run's PostIterate rule and compares it to the current aggregation statistics. PreIterate and PostIterate rules are commonly used together in this way; this can provide some continuity between aggregations. Data is stored into a custom configuration object (in this case, named "[AppName]_aggregationStats") by the PostIterate rule and read from it by the PreIterate rule on the next aggregation run.

```
import sailpoint.api.SailPointFactory;
import sailpoint.api.SailPointContext;
import sailpoint.tools.GeneralException;
import sailpoint.tools.xml.XMLObjectFactory;
import sailpoint.object.Configuration;

SailPointContext ctx = SailPointFactory.getCurrentContext();
if ( ctx == null ) {
    throw new GeneralException("Unable to get sailpoint context.");
}

String name = application.getName() + "_aggregationStats";
Configuration config = ctx.getObject(Configuration.class,name);
// The existence of a config object means the post rule
```

```
// has created an object and the stats should be checked
if ( config != null ) {
    if ( log.isDebugEnabled() ) {
      log.debug("CurrentStats: \n" + XMLObjectFactory.getInstance().toXml(stats));
      log.debug("Config : \n" + config.toXml());
    }
    String key = schema.getObjectType();
    Map lastStats = (Map)config.get(key);
    if ( lastStats != null ) {
       Long lastMod = (Long)lastStats.get("lastModified");
       Long currentMod = (Long)stats.get("lastModified");
       if ( currentMod < lastMod ) {
         throw new GeneralException("Last modification date is older than it was
during the last aggregation!");
       }

       // This scenario probably isn't real world (the size could decrease
       // without a problem); including it here for illustration
       Long currentLength = (Long)stats.get("length");
       Long lastLength = (Long)lastStats.get("length");
       if ( currentLength < lastLength ) {
         throw new GeneralException("The data file's length is less than it was during
the last aggregation!");
       }
    } else {
      if ( log.isDebugEnabled() ) {
        log.debug("Configuration for ["+key+"] was not found...Nothing checked.");
      }
    }
} else {
    if ( log.isDebugEnabled() ) {
      log.debug("Configuration ["+name+"] was not found...Nothing checked.");
    }
}
```

This example PreIterate rule places some data into CustomGlobal for use by the BuildMap (or some other) rule during aggregation. CustomGlobal is a class used to maintain a static Map of custom attributes; it was designed as a tool for maintaining global variables across calls to custom rules. **NOTE**: When the process is done with the CustomGlobal contents, another rule (such as the PostIterate rule) should clean up by removing the entry from CustomGlobal.

```
import sailpoint.object.CustomGlobal;
import java.util.HashMap;

log.debug("\n\nStarting Pre-Iterate Rule");

HashMap myHashMap = new HashMap();

myHashMap.put("length",stats.get("length"));
myHashMap.put("lastModified",stats.get("lastModified"));

CustomGlobal.put("FileStatMap",myHashMap);

return null;
```

BuildMap

Description

A BuildMap rule applies only to applications of type DelimitedFile. It is run for each row of data as it is read in from a connector. A BuildMap rule is used to manipulate the raw input data (provided via the rows and columns in the file) and build a map out of the incoming data.

If no BuildMap rule is specified, the default behavior is to traverse the column list (from the file header record or Columns list) and the parsed record, assigning each record element to the columns in order and inserting those pairs into a map. For example:

Columns: Name, ID, Phone
Record: John Doe, 1a3d3f, 555-555-1212
Map: Name, John Doe; ID, 1a3d3f; Phone, 555-555-1212

A convenience method is available to BuildMap rules that performs this default behavior. The remainder of the rule can then make modifications to the map. The convenience method is:

```
DelimitedFileConnector.defaultBuildMap(cols, record);
```

The rule must import the sailpoint.connector.DelimitedFileConnector class to use this method.

NOTE: Because this rule is run for each record in the input file, it can have a noticeable effect on performance if it contains time-intensive operations. Where possible, complicated lookups should be done in the PreIterate rule, with the results stored in CustomGlobal for use by the BuildMap rule; the global data should be removed by the PostIterate rule.

Definition and Storage Location

This rule is associated to an application in the UI on the Rules tab when defining an application of type DelimitedFile.

> Applications -> Application Definition -> create or select an application of Application Type:
> DelimitedFile -> Rules -> Build Map Rule

The rule name is recorded in the attributes map of the application XML.

```
<entry key="buildMapRule" value="[BuildMap Rule Name]"/>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
application	sailpoint.object.Application	A reference to the Application object.
schema	sailpoint.object.Schema	A reference to the Schema object for the Delimited File source being read.
state	java.util.Map	A Map that can be used to store and share data between executions of this rule during a single

		aggregation run
record	java.util.List	An ordered list of the values for the current record (parsed based on the specified delimiter)
cols	java.util.List	An ordered list of the column names from the file's header record or specified Columns list

Outputs:

Argument	Type	Purpose
map	java.util.Map	Map of names/values representing a row of data from the delimited file resource.

Example

This example BuildMap rule first invokes the default logic to create a map based on the defined columns and the record's values. It then manipulates targets and rights into direct permission objects by joining the map's target and rights values into a single direct permission value which is added to the map. The original target and rights are then removed from the map.

```
import sailpoint.connector.DelimitedFileConnector;
import sailpoint.object.Permission;

// Execute default build map logic
Map map = DelimitedFileConnector.defaultBuildMap(cols, record);

String strTarget = (String) map.get("target");
String strRights = (String) map.get("rights");

//Manipulate Target and Rights into Permissions
if ( strTarget != null && strRights != null )  {
   Permission perm = new Permission();
   perm.setRights(strRights);
   //probably need some annotations for these
   perm.setAnnotation("Annotation For Target: " + strTarget);
   perm.setTarget(strTarget);
   permList = new ArrayList();
   permList.add (perm);
   map.remove("target");
   map.remove("rights");
   map.put("directPermissions", permList);
}

return map;
```

JDBCBuildMap

Description

A JDBCBuildMap rule applies only to applications of type JDBC. It functions for JDBC applications just like the BuildMap rule does for Delimited File applications: it is used by the JDBC connector to create a map representation of the incoming ResultSet. The rule is called for each row of data as it is read in from the JDBC

connector. It is used to manipulate the raw input data (provided via the rows and columns) to build a map out of the incoming data.

If no JDBCBuildMap rule is called, the default logic builds the map out of the result data by directly matching the columns and values just as they come from the connector. There is a convenience method available to the rule to execute this default logic and build the basic map; the remainder of the rule can then make modifications to the default map. This convenience method is:

```
JDBCConnector.buildMapFromResultSet(result, schema);
```

The rule must import the sailpoint.connector.JDBCConnector class to use this method.

NOTE: Since this rule is run for every row of data returned from the resource, time-intensive operations performed within this rule can have a noticeable impact on aggregation performance. Try to avoid lengthy or complex operations in this rule.

Definition and Storage Location

The rule is associated to the application on the Rules tab when defining an application of type JDBC.

> Applications -> Application Definition -> create or select an application of Application Type: JDBC -> Rules -> Build Map Rule

The rule name is recorded in the attributes map of the application XML.

```
<entry key="buildMapRule" value="[JDBCBuildMapRuleName]"/>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
application	sailpoint.object.Application	A reference to the Application object
schema	sailpoint.object.Schema	A reference to the Schema object for the JDBC source being read
state	java.util.Map	A Map that can be used to store and share data between executions of this rule during a single aggregation run
result	java.sql.ResultSet	The current ResultSet from the JDBC Connector
connection	java.sql.Connection	A reference to the current SQL connection

Outputs:

Argument	Type	Purpose
map	java.util.Map	Map of names/values representing a row of data from the JDBC resource

Example

This basic rule performs the default mapping and then replaces the "status" value read from the database with a Boolean "inactive" attribute in the map.

```
import sailpoint.connector.*;
Map map = JDBCConnector.buildMapFromResultSet(result, schema);

string status = (String) map.get("status");
if "inactive".equals(status) {
    map.put("inactive", true);
} else {
    map.put("inactive", false);
}
map.remove("status");


return map;
```

SAPBuildMap

Description

An SAPBuildMap rule applies only to applications of type SAP. This rule differs from the Delimited File BuildMap rule and the JDBCBuildMap rule in that the SAP connector builds the attribute map for each object read from the connector before it calls this rule, so it passes the rule a prebuilt Map object instead of requiring the rule to build the map from a record or resourceObject. This rule can then modify the map as needed. The rule also receives a "destination" object through which it can make SAP calls to retrieve extra data.

NOTE: Since an SAPBuildMap rule is run once for every object read from an SAP data source, performing time-intensive operations in this rule can have a negative performance impact.

Definition and Storage Location

An SAPBuildMap rule is associated with the application on the Configuration-Settings tab when defining an application of type SAP.

> Applications -> Application Definition -> create or edit application of Application Type: SAP - Direct or SAP HR/HCM -> Configuration -> Settings -> SAP JCO Connection Settings section -> BuildMap Rule

The rule name is recorded in the attributes map of the application XML.

```
<entry key="buildMapRule" value="SAPBuildMapRuleName"/>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
application	sailpoint.object.Application	A reference to the Application object
schema	sailpoint.object.Schema	A reference to the Schema object that represents the object we are building

state	java.util.Map	A Map that can be used to store and share data between executions of this rule during a single aggregation run
destination	com.sap.conn.jco.JCoDestination	A connected and ready to use SAP destination object that can be used to call BAPI function modules and call to SAP tables.
object	sailpoint.object.Attributes	A reference to a SailPoint attributes object (basically a Map object with some added convenience methods) that holds the attributes that have been built up by the default connector implementation. The rule should modify this object to change, add or remove attributes from the map.
connector	sailpoint.connector.SAPInternalConnector	A reference to the current SAP Connector

Outputs: None. The rule modifies the "object" attribute directly to change the map, and subsequent IdentityIQ logic acts on the map as modified, making the "object" attribute the effective return value from the rule.

Example

This example SAP Build Map rule constructs an Initials attribute from the first character of the FirstName and LastName attributes and changes the name of the "InitDate" attribute to "HireDate".

```
import java.util.HashMap;


// Create initials
String firstName = object.get("FirstName");
String lastName = object.get("LastName");
String initials = "";

if (firstName != null && firstName.length() > 0) {
   char letter = firstName.charAt(0);
   letter = Character.toUpperCase(letter);
   initials = letter + ".";
}
if (lastName != null && lastName.length() > 0) {
   letter = lastName.charAt(0);
   letter = Character.toUpperCase(letter);
   initials += letter + ".";
}

object.put("Initials", initials);
object.put("HireDate", object.remove("InitDate"));
```

SAPHRManagerRule

Description

SAP HR/HCM contains HR data and therefore often contains the manager-employee relationship data for the organization. There are a couple of different predefined manager relationship models which are supported in

the application, but some customers may have a custom manager relationship model implemented. This rule allows those customers to apply their model to the data read from the SAP HR/HCM application to calculate the appropriate manager user for each identity. This rule type was introduced in IdentityIQ version 7.1.

Definition and Storage Location

An SAPHRManagerRule is associated with the application on the Configuration-Settings tab when defining an application of type SAP HR/HCM.

> Applications -> Application Definition -> select or create an application of Application Type: SAP HR/HCM
> -> Configuration -> Settings -> Manager Configuration section -> Custom -> Manager Rule

The rule name is recorded in the attributes map of the application XML.

```
<entry key="sapHRCustomManagerRule" value="SAP HR Manager Rule Name"/>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
application	sailpoint.object.Application	A reference to the Application object
schema	sailpoint.object.Schema	A reference to the Schema object that represents the object we are building
destination	com.sap.conn.jco.JCoDestination	A connected and ready to use SAP destination object that can be used to call BAPI function modules and call to SAP tables.
connector	sailpoint.connector.SAPInternalConnector	A reference to the current SAP Connector

Outputs:

Argument	Type	Purpose
supervisor	String	ID of the manager's Identity

Example

Usually, one of the built-in manager models would be used for SAPHR, so this rule type only applies when those are not in use. This example presumes the existence of a remote function in SAPHR called ZGETSUPERVISORID which accepts a positionId and employeeId value and returns the SupervisorId in its results. The actual logic needed in this rule would be completely dependent on the customer implementation details.

```
import java.util.*;
import com.sap.conn.jco.JCoDestination;
import com.sap.conn.jco.JCoException;
import com.sap.conn.jco.JCoFunction;
import com.sap.conn.jco.JCoTable;
String supervisor = null;
```

```
// Either a custom BAPI or the HRP1001 table  can be used to determine the supervisor
// id of an employee. For example, if there were a remote function module
// ZGETSUPERVISORID which required inputs of position id ,employee id to
// determine the supervisor, this rule could be used

JCoFunction getDetail = connector.getFunction(destination, "ZGETSUPERVISORID");
getDetail.getImportParameterList().setValue("POSITION", position);
getDetail.getImportParameterList().setValue("EMPLOYEEID", employeeID);
executeFunction(destination, getDetail);

Object supervisorId = getDetail.getExportParameterList().getValue("SUPERVISOR");
supervisor = (String)supervisorId;

return supervisor;
```

PeopleSoftHRMSBuildMap

Description

A PeopleSoftHRMSBuildMap rule applies only to applications of type PeopleSoft HCM Database. This rule was introduced in version 7.0 with the PeopleSoft HCM Database Connector.

As with the SAPBuildMap rule above, the PeopleSoft HRMS connector builds the attribute map for each object read from the connector before it calls this rule, so it passes the rule a prebuilt Map object instead of requiring the rule to build the map from a record. This rule can then modify the map as needed.

NOTE: Since a PeopleSoftHRMSBuildMap rule is run once for every object read from a PeopleSoft HRMS data source, performing time-intensive operations in this rule can have a negative performance impact.

Definition and Storage Location

A PeopleSoftHRMSBuildMap rule can be specified on the application settings page:

Applications -> Application Definition -> select application or create new of ApplicationType = PeopleSoft HCM Database -> Configuration -> Settings -> JDBC Connection Settings section -> Build Map Rule

The rule name is recorded in the attributes map of the application XML.

```
<entry key="buildMapRule" value="PeopleSoftHRMSBuildMapRule Name"/>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
application	sailpoint.object.Application	A reference to the Application object
schema	sailpoint.object.Schema	A reference to the Schema object that represents the object we are building
state	java.util.Map	A Map that can be used to store and share data between executions of this rule during a single aggregation run

identity	String	Name of the target identity
connection	java.sql.connection	Connection to the application database
connector	sailpoint.connector.PeopleSoftHR MSConnector	A reference to the current PeopleSoft HRMS Connector
map	java.util.Map	Map of attributes pre-built by the connector, to which the rule can add more attributes, as needed

Outputs:

Argument	Type	Purpose
map	java.util.Map	Map of names/values representing a person/account in the PeopleSoft HRMS application

Example

This example rule adds an extra attribute (MFC_IMUSERID) to the map of attributes being aggregated from PeopleSoft, providing the networkID for the user as an account attribute.

```
import java.sql.Statement;
import java.sql.ResultSet;
import java.util.HashMap;
import java.util.Map;
import sailpoint.connector.JDBCConnector;
import sailpoint.connector.PeopleSoftHRMSConnector;
import sailpoint.connector.PeopleSoftDirectConnector;

Statement statement=null;
ResultSet resultSet = null;

String type = schema.getObjectType();

// Return the map with all attribute values which are present in the schema. The
// connector object will call the buildMap method which contains the resource
object.
// The type will be account.

// The query retrieves the Network ID of the PeopleSoft HR record which has primary
flag
// as 'Y'. The Network ID is a collection attribute defined in PeopleSof HRMS.
Table for
// accessing the Network ID is PS_PERSON_IMCHAT and MCF_IMUSERID is the attribute
which
// is mapped as NetworkID in PeopleSoft portal.
String query = "select MCF_IMUSERID from PS_PERSON_IMCHAT where PREF_CHATID_FLAG =
'Y' and EMPLID ='" + identity + "'";

//The connection object is in context and executing the above query. The result
will
//return the data of the field.
statement = connection.createStatement();
resultSet = statement.executeQuery(query);
if (resultSet.next()) {
  String networkIDValue = resultSet.getObject(1);
    map.put("MCF_IMUSERID",networkIDValue);
```

```
    }else{
        if(log.isDebugEnabled()){
            log.debug("No NetworkID exist for user:"+ identity );
        }
    }
    statement.close();
    resultSet.close();

    //Return the map of which contains the final resource object map with the new
field value.
    return map;
```

FileParsingRule

Description

A FileParsingRule is used with applications of type RuleBasedFileParser, which is used to parse non-delimited files. This connector can read account and group data from non-standard or free format text. The rule is called to retrieve each complete record from the file; logic in the rule determines what constitutes a complete record – whether that is on one line in the file or whether it spans multiple lines.

NOTE: Since the FileParsingRule rule runs to extract every account or group record from the file and build it into a map, any time-intensive operations performed in this rule can have a negative performance impact.

Definition and Storage Location

The rule is associated to the application on the Configuration-Settings tab when defining an application of type RuleBasedFileParser.

> Applications -> Application Definition -> select or create application of Application Type: RuleBasedFileParser -> Configuration -> Settings -> parseRule

The rule name is recorded in the attributes map of the application XML.

> `<entry key="parseRule" value="[FileParsingRule Name]"/>`

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
application	sailpoint.object.Application	A reference to the Application object.
schema	sailpoint.object.Schema	A reference to the Schema object for the Delimited File source being read.
config	sailpoint.object.Attributes	Attributes Map of Application configuration attributes
inputStream	java.io.BufferedInputStream	A reference to the file input stream
reader	java.io.BufferedReader	A reader wrapping the inputStream
state	java.util.Map	A Map that can be used to store and share data between executions of this rule during a single aggregation run

Outputs:

Argument	Type	Purpose
map	java.util.Map	Return value representing a Map of names/values from the connected resource.

Example

This example rule reads records from a file and parses records with a specific tag present according to a fixed record layout.

```
import java.io.BufferedInputStream;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

HashMap map = new HashMap();

String record;

// Read a record from file; look for XYZ string in record and
// parse those records into substrings to extract data
if ( ( record = reader.readLine() ) != null) {
    if (record.contains("XYZ")) {
        String userId = record.substring(record.indexOf("XYZ") + 8,
record.indexOf("XYZ") + 12);
        map.put("UserId", userId);
        String fullname = record.substring(record.indexOf("XYZ") + 16,
record.indexOf("XYZ") + 36).trim();
        map.put("FullName",fullname);
        String permission = record.substring(record.indexOf("XYZ") + 40,
record.indexOf("XYZ") + 44);
        map.put("Permission", permission);

}

return map;
```

MergeMaps

Description

A MergeMaps rule is used to specify a custom basis for merging of rows from a Delimited File or JDBC application. The connectors include a default merge algorithm that merges the rows based on the defined merge parameters. If a MergeMaps rule is specified, it overrides the default merge operation with the rule's custom behavior.

A convenience method is available that performs the default merge algorithm, allowing the remainder of the rule to apply customizations to that default merging. This convenience method is:

```
AbstractConnector.defaultMergeMaps(current, newObject, mergeAttrs);
```

The sailpoint.connector.AbstractConnector class must be imported into the rule to use this method. (Alternatively, since both the DelimitedFileConnector and the JDBCConnector classes extend AbstractConnector, the applicable one of those classes could be imported with the method call naming that class instead.)

NOTE: Since the MergeMaps rule runs for every row or ResultSet of data from a delimited file or JDBC data source, performing lengthy operations in this rule can have a negative effect on aggregation performance.

Definition and Storage Location

The rule is associated to the application on the Rules tab when defining an application of type DelimitedFile or JDBC.

> Applications -> Application Definition -> create or edit application of Application Type: Delimited File or JDBC -> Rules -> MergeMaps Rule

The rule name is recorded in the attributes map of the application XML:

```
<entry key="mergeMapsRule" value="MergeMapsRuleName"/>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
application	sailpoint.object.Application	A reference to the Application object.
schema	sailpoint.object.Schema	A reference to the Schema object for the Delimited File or JDBC source being read.
current	java.util.Map	The current Map object
newObject	java.util.Map	The map representation of the next row that potentially needs to be merged into the current object based on mergeAttrs
mergeAttrs	java.util.List (of Strings)	Names of attributes that need to be merged, specified as part of the application configuration

Outputs:

Argument	Type	Purpose
merged	java.util.Map	Map of names/values representing a merged row of data from the connected resource.

Example

For each attribute in the mergeAttrs list, this example MergeMaps rule first tries to merge any values from the new object attribute into a List in the current object attribute. If the current object does not contain a list for that attribute (or if the attribute is null), the rule replaces the current object value with the new object value.

```
import java.util.Map;
import java.util.HashMap;

Map merged = new HashMap(current);

for ( String attrName : mergeAttrs ) {
    Object currentValue = current.get(attrName);
    Object additionalValue = newObject.get(attrName);
    if ( currentValue != null ) {
       if ( additionalValue != null ) {
          if ( currentValue instanceof List ) {
             if ( additionalValue instanceof List ) {
                // loop through additional values list adding to current
                // value list if not already there
                for ( Object value : (List)additionalValue ) {
                   if (!((List)currentValue).contains(value)) {

                      ((List)currentValue).add(value);
                   }
                }
             } else {
                if (!((List)currentValue).contains(additionalValue) ) {
                   // Add value to list if not already there
                   ((List)currentValue).add(additionalValue);
                }
             }
          } else { // currentValue is not list
             // replace attribute with new object value in return map
             merged.put(attrName, additionalValue);
          }
       }
    } else {  // current value is null
        if ( additionalValue != null ) {
          // Add additionalValue as attribute in map
         merged.put(attrName, additionalValue);
        }
    }
} // end for

return merged;
```

Transformation

Description

This rule is run for every account or group read from a delimited file or JDBC application. It runs after the BuildMap rule (and MergeMaps, if applicable) and is used to control the transformation of each map into a ResourceObject. Connectors must get data into this ResourceObject format before it can be processed by the aggregator and the aggregation rules.

If no transformation rule is specified, the transformation is performed through the defaultTransformObject method in the AbstractConnector class. This method is available to the rule as a convenience method and can

be used to do the basic conversion, allowing the rule to do further customization on the ResourceObject in the remainder of its logic. The convenience method is:

```
AbstractConnector.defaultTransformObject(schema, object);
```

The sailpoint.connector.AbstractConnector class must be imported into the rule to use this method.

NOTE: Since the Transformation rule runs for every map created from the source data, time-intensive operations performed in it can have a negative impact on aggregation performance.

Definition and Storage Location

The rule is associated to an application on the Rules tab when defining an application of type DelimitedFile or JDBC.

> Applications -> Application Definition -> create or edit application of Application Type: DelimitedFile or
> JDBC -> Rules -> Map To ResourceObject Rule

The rule name is recorded in the attributes map of the application XML.

```
<entry key="mapToResourceObjectRule" value="[Transformation Rule Name]"/>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
application	sailpoint.object.Application	A reference to the Application object.
schema	sailpoint.object.Schema	A reference to the Schema object for the Delimited File source being read.
object	java.util.Map	The incoming Map object

Outputs:

Argument	Type	Purpose
resourceObject	sailpoint.object.ResourceObject	Return value representing a ResourceObject constructed from the incoming Map

Example

This example transformation rule would be specified for an application that is loading AD data from a delimited file. It examines the memberOf attribute for an Admin group membership and adds an "isAdmin" attribute to the resourceObject where applicable.

```
import sailpoint.connector.AbstractConnector;
import sailpoint.object.ResourceObject;

ResourceObject ro = AbstractConnector.defaultTransformObject(schema, object);
```

```
        List groups = (List)ro.getAttribute("memberOf");
        if ( groups != null ) {
            for ( String group : groups ) {
                if ( ( group != null ) &&
                    ( group.startsWith("cn=Domain Admins") ) ) {
                        ro.put("isAdmin", true);
                }
            }
        }
        return ro;
```

PostIterate

Description

A PostIterate Rule can be specified only for an application of Type: DelimitedFile or RuleBasedFileParser. It runs after all other connector rules and can be used to execute any processing that should occur after the connector iteration is complete. Commonly, it is used to remove any global data used by the other rules (e.g. from CustomGlobal), clean up files, or mark statistics in a configuration object that will be used by the PreIterate rule during a subsequent aggregation. Since it runs only once per aggregation, this rule generally has a minimal impact on aggregation performance.

Definition and Storage Location

The rule is associated with the application in the UI when defining an application of type DelimitedFile or RuleBasedFileParser.

> Applications -> Application Definition -> select or create an application of Application Type: DelimitedFile -> Rules -> PostIterate Rule

> Applications -> Application Definition -> select or create an application of Application Type: RuleBasedFileParser -> Configuration -> Settings -> postIterateRule

The Rule name is recorded in the attributes map of the Application XML.

```
<entry key="postIterateRule" value="[PostIterate Rule Name]"/>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
application	sailpoint.object.Application	A reference to the Application object
schema	sailpoint.object.Schema	A reference to the Schema object for the Delimited File/JDBC source being read
stats	java.util.Map	A map of the stats for the file just iterated Contains keys: • fileName : (String) filename of the file about to be processed • absolutePath : (String) absolute filename

		• length : (Long) length in bytes • lastModified : (Long) last time the file was updated Java GMT • columnNames : (List) column names that were used during the iteration • objectsIterated : (Long) total number of objects iterated during this run

Outputs: None. The rule's logic acts upon objects outside of the aggregation data flow, so subsequent steps in the process do not expect a return value from this rule and will not act upon it if one were provided.

Examples

This example PostIterate rule records some information in a custom configuration object so that it can be read and acted upon by a subsequent PreIterate rule.

```
import sailpoint.api.SailPointFactory;
import sailpoint.api.SailPointContext;
import sailpoint.tools.GeneralException;
import sailpoint.object.Configuration;
import sailpoint.object.Attributes;

SailPointContext ctx = SailPointFactory.getCurrentContext();
if ( ctx == null ) {
    throw new GeneralException("Unable to get sailpoint context.");
}

String name = application.getName() + "_aggregationStats";
Configuration config = ctx.getObject(Configuration.class,name);
if ( config == null ) {
    if ( log.isDebugEnabled() ) {
  log.debug("Configuration ["+name+"] was not found creating new one.");
    }
    config = new Configuration();
    config.setName(name);
}

Attributes attrs = config.getAttributes();
if ( attrs == null ) attrs = new Attributes();

String key = schema.getObjectType();
attrs.put(key, stats);
config.setAttributes(attrs);
if ( log.isDebugEnabled() ) {
    log.debug("Newly created Configuration object :\n"+ config.toXml());
}
ctx.saveObject(config);
ctx.commitTransaction();
```

This example PostIterate rule removes data from CustomGlobal that was stored there by a PreIterate rule for a BuildMap or other rule to use (see PreIterate rule example).

```
import sailpoint.object.CustomGlobal;
```

```
System.out.println("In Post-Iterate Rule...");

// Remove the Map from custom global...
if (CustomGlobal.get("FileStatMap"))
{
    CustomGlobal.remove("FileStatMap");
}
```

WebServiceBeforeOperationRule

Description

The WebService connector supports interaction with (through both read and write operations) systems which provide a REST API to allow other applications to connect to them. The appropriate REST calls can be configured for each desired interaction type. For each interaction type, a "before" and "after" operation rule can be specified, as described here and in the next rule type. This connector type, and therefore this rule, was added to IdentityIQ in version 7.1.

NOTE: The WebService connector only supports JSON Responses from the API for both read and write operations.

The WebServiceBeforeOperationRule is run immediately before running the associated REST call to the target system. Examples of actions it can perform include fetching a run-time generated token from the target system to be used in the REST API call or modifying the request to handle paged results.

Definition and Storage Location

This rule can be specified in the application definition for applications using the Web Services connector.

> Applications -> Application Definition -> choose or add new application of Application Type: Web Services -> Configuration -> Settings -> Connector Operations: Add Operation -> click pencil icon to edit -> Before Rule -> Before Operation Rule

It appears in the application XML as part of the connectionParameters entry in each operation's map like this:

```
<entry key="connectionParameters">
  <value>
    <List>
      <Map>
        <entry key="beforeRule" value="Before Operation Rule Name"/>
        …
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
application	sailpoint.object.Application	Application whose data file is being processed
requestEndPoint	sailpoint.connector.webservices.EndPoint	Current request information; contains the header, body, context url, method type, response attribute map, successful response code
oldResponseMap	Java.util.Map	Earlier response object

restClient	sailpoint.connector.webserv ices.WebServicesClient	REST client object

Outputs:

Argument	Type	Purpose
endpoint	Sailpoint.connector.webserv ices.EndPoint	Updated EndPoint Object

Example

This example rule works in conjunction with the example WebServicAfterOperationRule, shown below, to provides paging support when retrieving data from DropBox. If the retrieved data is paged, the AfterOperationRule sets the cursor into a "transientValues" map in the application. Then this rule checks to see if there is a cursor present (indicating paged results) and modifies the postBodyMap and requestEndPoint URL accordingly so the operation itself will retrieve the next page of data.

```
import com.google.gson.Gson;
import sailpoint.tools.Util;
Map obj = (Map) application.getAttributeValue("transientValues");
if(null != obj) {
  String cursor = obj.get("cursor");
if(Util.isNotNullOrEmpty(cursor)) {
    Map postBodyMap = (Map) requestEndPoint.getBody();
    postBodyMap.put("jsonBody", "{\"cursor\":\"" + cursor + "\"}");
    requestEndPoint.setFullUrl(requestEndPoint.getFullUrl() + "/continue");
  }
}
return requestEndPoint;
```

WebServiceAfterOperationRule

Description

The WebService connector allows IdentityIQ to connect to applications through the applications' REST API methods, for both read and write operations. The appropriate REST calls can be configured for each desired operation type. For each operation type, a "before" and "after" operation rule can be specified, as described here and in the rule type above. This connector type, and therefore this rule, was added to IdentityIQ in version 7.1.

The WebServiceAfterOperationRule is run immediately after running the associated REST call to the target system.

Definition and Storage Location

This rule can be specified in the application definition for applications using the Web Services connector.

Applications -> Application Definition -> choose or add new application of Application Type: Web Services -> Configuration -> Settings -> Connector Operations: Add Operation -> click pencil icon to edit -> After Rule -> After Operation Rule

It appears in the application XML as part of the connectionParameters entry in each operation's map like this:

```
<entry key="connectionParameters">
  <value>
    <List>
      <Map>
        <entry key="afterRule" value="After Operation Rule Name"/>
        …
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
application	sailpoint.object.Application	Application whose data file is being processed
requestEndPoint	sailpoint.connector.webservices.EndPoint	Current request information; contains the header, body, context url, method type, response attribute map, successful response code
processedResponseObject	List<Map<String, Object>>	Response Object processed by the Web Services connector
rawResponseObject	String	Response Object returned from the end system
restClient	sailpoint.connector.webservices.WebServicesClient	REST client object

Outputs:

Argument	Type	Purpose
updatedAccountOr GroupList	Java.util.Map	Updated information map containing parsed list of accounts

Example

This example rule works in conjunction with the example WebServicBeforeOperationRule, shown above, to provides paging support when retrieving data from DropBox. This AfterOperationRule checks for "cursor" and "has_more" attributes in the operation's JSON response, which indicate that the response contains paged results, and sets them into a "transientValues" map in the application definition. Then the BeforeOperationRule checks to see if there is a cursor present (indicating paged results) in the application's transientValues map and modifies the postBodyMap and requestEndPoint URL accordingly so the operation itself will retrieve the next page of data.

```
import com.google.gson.Gson;
```

```
import com.google.gson.JsonElement;
import com.google.gson.JsonObject;
import com.google.gson.JsonParser;
import sailpoint.tools.Util;

Gson gson = new Gson();
JsonObject jsonObject = new JsonParser().parse(rawResponseObject).getAsJsonObject();
String cursor = jsonObject.get("cursor").getAsString();
boolean hasMore = jsonObject.get("has_more").getAsBoolean();

if(Util.isNotNullOrEmpty(cursor)) {
  Map transientValues = application.getAttributeValue("transientValues");
  if(transientValues == null) {
    transientValues = new HashMap();
    application.setAttribute("transientValues", transientValues);
  }
  transientValues.put("cursor", cursor);
  transientValues.put("hasMore", hasMore);
```

This example code snippet shows how you could update the processedResponseObject in your rule (this example sets the value of a specific attribute to upper case).

```
import java.util.Map.Entry;

for (Map hm : processedResponseObject) {
        Set entries2 = hm.entrySet();
        for (Entry entry : entries2) {
            if(entry.getKey().equals("PRIVId")) {
                String value = (String)entry.getValue();
                entry.setValue( ((String)entry.getValue()).toUpperCase() );
                    }
                }
        }
```

RACFPermissionCustomization

Description

The RACFPermissionCustomization rule is run during aggregation from the newer (version 7.1+) RACF direct connector. It is used to customize the permissions map.

Definition and Storage Location

This rule is an XML-only configuration. It appears in the application XML as part of the Attributes map like this:

```
<Application connector="sailpoint.connector.RACFConnector" name="RACF" type="RACF">
  <Attributes>
    <Map>
      <entry key="permissionCustomizationRule" value="Perm Customize Rule"/>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
permission	sailpoint.object.Permission	Permission object (map), as built from

		RACF record (line) directly
line	String	Individual record read from RACF

Outputs:

Argument	Type	Purpose
permission	sailpoint.object.Permission	Updated permission object

Aggregation/Refresh Rules

Aggregation Rules are used during part of the aggregation process that occurs after the connector has created valid ResourceObjects for the accounts or groups being aggregated, i.e. after the defined connector rules have all been run. There are two types of aggregation: account and account group. All rules discussed in this section except AccountGroupRefresh apply only to account aggregations; the AccountGroupRefresh rule applies only to account group aggregation. A Refresh rule can be specified to run at the end of account aggregation but also can also be run from an Identity Refresh task.

The rules described in this section can be used to perform these actions:

- Modify the ResourceObjects provided by the connector before they are correlated and aggregated
- Correlate ResourceObjects to existing Identities
- Control attribute population during creation new of Identities when a matching Identity is not found for correlation, particularly when aggregating from an authoritative source
- Correlate manager Identities
- Customize the creation of Managed Entitlements during aggregation/refresh
- Customize an Identity before storing it (at the end of aggregation or during Identity Refresh)
- Customize account group attributes before storing (during account group aggregation)

Aggregation rules are described here in the order in which they are run when specified for a given aggregation task.

ResourceObjectCustomization

Description

A ResourceObjectCustomization rule runs prior to any other aggregation rule to customize the resource object provided by the connector before aggregation begins. Connectors that provide a transformation rule may not need to use a ResourceObjectCustomization rule, since the transformation rule can modify the ResourceObject as needed. However, many connectors directly provide a resource object, without the hooks for processing the data through rules like a transformation rule, so this rule allows customization of the resource object before IdentityIQ attempts to correlate the object to an Identity.

NOTE: Since the ResourceObjectCustomization rule runs for every ResourceObject provided by the connector, time-intensive operations performed in it can have a negative impact on task performance. It runs even when optimized aggregation has been specified, so customizations made by this rule can impact the optimization decisions for the ResourceObject.

Definition and Storage Location

This rule is associated to an application in the UI in the application definition:

> Applications -> Application Definition -> select existing or create new application -> Rules -> Customization Rule

The reference to the rule is recorded in the Application XML:

```xml
<CustomizationRule>
    <Reference class="sailpoint.object.Rule" id="GUID ID" name="[Customization Rule Name]"/>
</CustomizationRule>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
object	sailpoint.object.ResourceObject	A reference to the resource object built by the connector
application	sailpoint.object.Application	A reference to the Application object
connector	sailpoint.connector.abstractConnector	A reference to the Connector object used by this application
state	java.util.Map	A Map that can be used to store and share data between executions of this rule during a single aggregation run

Outputs:

Argument	Type	Purpose
object	sailpoint.object.ResourceObject	The modified resource object

NOTE: Even though the ResourceObject is passed to the rule where it can be modified directly, it must be returned from the rule for its changes to be used by the rest of the aggregation process. Otherwise, changes made to it inside the rule are not transferred back to the rest of the process.

Example

This example ResourceObjectCustomization rule performs the same function as the example Transformation rule but would apply to an AD application that aggregated directly from AD (where the Transformation rule is not availalbe) rather than through a delimited file.

```
import sailpoint.tools.xml.XMLObjectFactory;
```

```
List groups = (List)object.getAttribute("memberOf");
if ( groups != null ) {
    for ( String group : groups ) {
        if ( ( group != null ) &&
            ( group.startsWith("cn=Domain Admins") ) ) {
            object.put("isAdmin", true);
        }
    }
}
return object;
```

Correlation

Description

A Correlation Rule is used to select the existing Identity to which the aggregated account information should be connected. Correlation can be specified on the application definition through a simple attribute match process or it can be managed with a rule. If both are specified, the correlation rule supersedes the correlation attribute specification and the simple attribute match will only be attempted if the rule does not return an Identity.

Every time an aggregation task runs, except when the optimize aggregation option has been selected or when an account has been manually correlated to an Identity, the Identity to which the account should be connected is reassessed; if the existing correlation is found to be incorrect or no longer applicable, that connection is broken and a new one is established to the correct Identity.

If the correlation rule returns null or if the information returned from the correlation rule does not match to an Identity (and the attribute-matching process also fails to select an Identity), a new Identity will be created (see the *IdentityCreation* rule) and the account will be correlated to that Identity.

NOTE: In IdentityIQ 6.0, optimized aggregation is the default behavior, which means that no changes will be made to a Link object if the corresponding managed system account has not changed. Consequently, accounts will not be recorrelated to Identities in subsequent aggregations if nothing has changed on the application account. (Other actions will be bypassed too, including attribute promotion, manager correlation, etc., but the skipped recorrelation is usually the most noticeable effect of this setting.) Optimized aggregation can be turned off by selecting the **Disable optimization of unchanged accounts** option in the aggregation task options or specifying <entry key="noOptimizeReaggregation" value="true"/> in the TaskDefinition XML attributes map.

NOTE: Except as noted above with respect to optimized aggregation, the Correlation rule runs for every Link created in the aggregation. Therefore, time-intensive operations performed in it can have a negative impact on aggregation performance.

Definition and Storage Location

This rule is associated to an application in the UI in the application definition:

> Applications -> Application Definition -> select existing or create new application -> Rules -> Correlation Rule

The reference to the rule is recorded in the Application XML:

```
<CorrelationRule>
    <Reference class="sailpoint.object.Rule" id="GUID ID" name="[Correlation Rule
Name]"/>
</CorrelationRule>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
environment	java.util.Map	Map of arguments passed to the aggregation task
application	sailpoint.object.Application	A reference to the Application object
account	sailpoint.object.ResourceObject	A reference to the ResourceObject passed from the connector
link	sailpoint.object.Link	A reference to the existing link identified based on the resourceObject, if any

Outputs:

Argument	Type	Purpose
map	java.util.Map	Map that identifies an Identity; may contain any of the following key-value pairs: "identityName", "[identity.name value]" or "identity", [identity object] or "identityAttributeName", "[attribute name]" "identityAttributeValue", "[attribute value]" where the attribute value uniquely identifies one Identity

Examples

This example Correlation rule concatenates a firstname and lastname field from the account (resourceObject) to build an Identity name for matching to an existing Identity.

```
Map returnMap = new HashMap();

String firstname = account.getStringAttribute("firstname");
String lastname = account.getStringAttribute("lastname");

if ( ( firstname != null ) && ( lastname != null ) ) {
    String name= firstname + "." + lastname;
    returnMap.put("identityName", name);
}
return returnMap;
```

This example correlation rule correlates the account to an Identity based on a combination of region and employee ID from the application account, which together can be used to match the unique employee ID recorded on the Identity.

```
import java.util.Map;
```

```
import java.util.HashMap;

String empNum = account.getStringAttribute("employeeId");
String region = account.getStringAttribute("region");
String empId = region + empNum;

Map returnMap = new HashMap();

if ( empId != null ) {
    returnMap.put("identityAttributeName", "empId");
    returnMap.put("identityAttributeValue", empId);
}
return returnMap;
```

IdentityCreation

Description

If the correlation rule cannot find an Identity that corresponds to the account, one must be created. By default, the Identity Name is set to the display attribute from the resource object (or the identity attribute if display attribute is null) and the Manager attribute is set to false. An IdentityCreation rule specifies any other Identity attribute population, or any change to these two attribute values, based on the account data. It can also be used to set values like a default IdentityIQ password for the Identity.

If the application is not an authoritative application, any Identities created for its accounts must later be manually correlated to an authoritative Identity or the accounts will have to be recorrelated through an automated process to connect them to the correct authoritative Identities.

IdentityCreation rules are most commonly specified for authoritative applications, since new Identities created from those accounts are real, permanent Identities. However, they can also be used for non-authoritative application accounts to set attributes that can make manual correlation easier.

An IdentityCreation rule can also optionally be specified as an Auto-Create User Rule in the IdentityIQ Login Configuration, as described later in this document.

Definition and Storage Location

This rule is associated to an application in the UI through the application definition:

> Applications -> Application Definition -> select existing or create new application -> Rules -> Creation Rule

The reference to the rule is recorded in the Application XML:

```
<CreationRule>
    <Reference class="sailpoint.object.Rule" id="[Rule ID]" name="[Identity Creation
Rule Name]"/>
</CreationRule>
```

This rule type can also be associated to an account aggregation task:

Setup -> Tasks -> select existing or create new account aggregation task -> Optionally select a rule to assign capabilities or perform other processing on new identities

When connected to a task, the rule ID is recorded in the attributes map of the TaskDefinition XML as the value for the "creationRule" attribute:

```
<Attributes>
  <Map>
    <entry key="creationRule" value="402846023a65e596013a65e7a9fa0505"/>
  </Map>
</Attributes>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
environment	java.util.Map	Map of the task arguments for the aggregation task
application	sailpoint.object.Application	Reference to the application object from which the account was read
account	sailpoint.object.ResourceObject	Reference to the ResourceObject representing the account
identity	sailpoint.object.Identity	Reference to the Identity being created

Outputs: None. The identity object passed as parameter to the rule should be edited directly by the rule.

Example

This example IdentityCreation rule concatenates a firstname and lastname field from the account (resourceObject) to create an Identity name (firstname.lastname), assigns an IdentityIQ capability based on the account's group membership, and sets a default password for the Identity. The identity parameter is directly modified by the rule; no return value is expected.

```
import sailpoint.object.Identity;
import sailpoint.object.Capability;
import sailpoint.object.ResourceObject;

// change the name to a combination of firstname and lastname
String firstname = account.getStringAttribute("firstname");
String lastname = account.getStringAttribute("lastname");
String name  = firstname + "." + lastname;
identity.setName(name);

// add capabilities based on group membership
List groups = (List)account.getAttribute("memberOf");
if ( ( groups != null ) && ( groups.contains("Administrator") ) ) {
    identity.add(context.getObjectByName(Capability.class,
"ApplicationAdministrator"));
}
identity.setPassword("P@ssw0rd");
```

ManagerCorrelation

Description

As with Identity correlation, manager correlation can be specified on the application definition through a simple attribute match process or it can be managed with a rule. If a rule is defined, it is used to determine the correct manager Identity; if no rule is defined, the Identity attribute match process is used to find the manager Identity. A ManagerCorrelation Rule identifies an Identity's manager based on an application account attribute (or attributes) on the account being aggregated or refreshed.

The ManagerCorrelation rule runs as part of the identity refresh process that occurs either in an identity refresh task or at the end of an account aggregation task, though the manager correlation option is hidden from the UI on the aggregation task and is more often performed only as part of a refresh task. It runs before both the managedAttributeCustomization rule(s) and the Refresh rule (if any).

NOTE: In general, manager correlation is bypassed if no change has been detected in the source attribute. This is because manager correlation can be time-intensive activity that negatively impacts aggregation/refresh performance. To force manager correlation to occur for every Identity, even if it has not changed, set the alwaysRefreshManager attribute to "true" in the task attributes map. When this option is set, time-intensive operations performed in the ManagerCorrelation rule can have a negative impact on task performance.

Definition and Storage Location

This rule is associated to an application in the UI in the application definition:

> Applications -> Application Definition -> select existing or create new application -> Rules -> Manager Correlation Rule

The reference to the rule is recorded in the Application XML:

```
<ManagerCorrelationRule>
    <Reference class="sailpoint.object.Rule" id="GUID ID" name="[Manager Correlation
Rule Name]"/>
</ManagerCorrelationRule>
```

The promoteAttributes option must be turned on for the aggregation or refresh task for manager correlation to run during the task. The alwaysRefreshManager option forces manager refresh to occur even when the source link's attribute value has not changed; alternatively, the noManagerCorrelation option bypasses all manager correlation regardless of whether or not the source attribute value has changed.

```
<entry key="promoteAttributes" value="true"/>
<entry key="alwaysRefreshManager" value="true"/>
<entry key="noManagerCorrelation" value="true"/>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose

environment	java.util.Map	Arguments passed to the aggregation task
application	sailpoint.object.Application	A reference to the Application object
instance	String	Application instance name (if not null)
connector	sailpoint.connector.AbstractConnector	A reference to the connector object used by this application instance
link	sailpoint.object.Link	A reference to the account link
managerAttributeValue	String	Manager attribute value being used to correlated to an Identity

Outputs:

Argument	Type	Purpose
map	java.util.Map	Map that identifies the manager's Identity; may contain any of the following: "identityName", "[identity.name value]" or "identity", [Identity object] or "identityAttributeName", "[attribute name]" "identityAttributeValue", "[attribute value]" where the attribute value uniquely identifies one Identity

Example

This example ManagerCorrelation rule is provides the same functionality as using the attribute-correlation option on the Correlation tab of the application definition; an installation might specify this as a rule if they have a preference for rules over UI configurations or for other organization-specific reasons.

```
import java.util.Map;
import java.util.HashMap;

Map returnMap = new HashMap();

if (managerAttributeValue != null) {
    returnMap.put("identityAttributeName", "empId");
    returnMap.put("identityAttributeValue", managerAttributeValue);
} else {
    returnMap.put("identityName", "");
}

return returnMap;
```

This example ManagerCorrelation rule provides a more complex demonstration of this rule type. The Identity name is a 7-character employee ID that is the prefix to accountIDs on the application from which Managers are correlated, so only the first 7 characters of the manager's accountID value need to be returned for the correlation.

```
// Account IDs contain the 7 character employee ID plus an app-specific
// suffix. Strip all accounts down to the first 7 characters for
// Identity name and use that to correlate to manager.
```

```
import java.lang.String;
import java.util.Map;
import java.util.HashMap;

Map returnMap = new HashMap();

String name = link.getStringAttribute("MgrID");

if ( name != null ) {
  name = name.trim();

  if (7 &lt; name.length()) {
    name = name.substring(0,7);
  }
  name = name.toLowerCase();
  returnMap.put("identityName", name);
}

return returnMap;
```

ManagedAttributeCustomization / ManagedAttributePromotion

Description

The ManagedAttributeCustomization rule for an application is run by an aggregation task or an identity refresh task for which the "Promote managed attributes" option selected. This rule can set values on managed attributes as they are promoted for the first time, and it only runs when a managed attribute is initially created through promotion (i.e. on create, not on update). The rule directly modifies the ManagedAttribute passed to it, so it does not have a return value.

This rule type was renamed in IdentityIQ version 6.2 to ManagedAttributePromotion. The new name is more descriptive of when this rule is run and how it should be used.

The ManagedAttributeCustomization/Promotion Rule runs during a refresh process that occurs either in an identity refresh task or at the end of an account aggregation task. Managed attributes are promoted after identity attributes are refreshed (including manager correlation) but before any Refresh rule specified for the task is run.

Definition and Storage Location

This rule is associated to an application in the UI through the application definition.

> Applications -> Application Definition -> select existing or create new application -> Rules -> Managed Entitlement Customization Rule

The reference to the rule is recorded in the Application XML:

```
<ManagedAttributeCustomizationRule>
    <Reference class="sailpoint.object.Rule" id="GUID ID" name="[Managed Attribute
Customization Rule Name]"/>
</ManagedAttributeCustomizationRule>
```

The promoteManagedAttributes option must be turned on for the aggregation or refresh task for managed attribute promotion and the ManagedAttributeCustomization rule to run during the task.

```
<entry key="promoteManagedAttributes" value="true"/>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
attribute	sailpoint.object.ManagedAttribute	A reference to the managed attribute being created
application	sailpoint.object.Application	A reference to the application object to which this managed attribute belongs
state	java.util.Map	Map in which any data can be stored; available to the rule in subsequent rule executions within the same task so expensive data (requiring time-intensive lookups) can be saved and shared between rule executions.

Outputs: None; the ManagedAttribute object passed as parameter to the rule should be edited directly by the rule.

Example

This example ManagedAttributeCustomization rule sets the owner as the application owner and sets a description for the managed attribute.

```
import sailpoint.object.*;
import java.util.Locale;


// set the owner to the app owner.
Identity owner = null;
if ((null != application) && (null != application.getOwner())) {
    owner = application.getOwner();
} else {
    owner = getObjectByName(Identity.class,"spadmin");
}
attribute.setOwner(owner);

//make attribute requestable
attribute.setRequestable(true);

String description = "friendly description";

// In 6.0+, use this logic to set descriptions for managed attributes

attribute.addDescription(Locale.US.toString(), description);

// In versions prior to 6.0, set description like this:

attribute.setExplanation("default", description);
```

Refresh

Description

A Refresh rule runs at the end of the Identity Refresh process, both during an Identity Refresh task and at the end of an Aggregation task. It allows custom manipulation of Identity attributes while the Identity is being refreshed. Refresh rules are most commonly used in manually-executed refresh tasks configured for data cleanup when erroneous aggregation configurations have resulted in unintended data consequences on Identities. However, they can also be used in normal refresh or aggregation tasks to set attributes (usually custom attributes).

NOTE: IdentityIQ 6.0 introduced a second option for running a Refresh rule at the *beginning* of the Identity Refresh process in addition to the end. To have the rule run at the beginning, specify it as a "preRefreshRule", as shown in the *Definition and Storage Location* section below. This option will rarely be used but is available if attribute values need to be manipulated before the Identity and its associated links and roles are refreshed.

NOTE: Since the Refresh rules run for every Identity involved in the aggregation or refresh task, time-intensive operations performed in it can have a negative impact on task performance.

Definition and Storage Location

This rule is specified as a task argument to a refresh task but can only be added to the task XML, not through the UI.

The rule name is recorded in the taskDefinition XML as:

```
<entry key= "refreshRule" value = "[Refresh Rule Name]" />
<entry key= "preRefreshRule" value = "[Refresh Rule Name]" />
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
environment	java.util.Map	Arguments passed to the aggregation or refresh task
identity	sailpoint.object.Identity	Reference to the Identity object being refreshed

Outputs: None. The identity object passed as parameter to the rule should be edited directly by the rule.

Example

This example Refresh rule is used to reset passwords for all refreshed Identities except the Administrator to a default password. It also sets a custom attribute to record the date of the reset on the Identity.

```
import sailpoint.object.Identity;
import sailpoint.object.Capability;

String name = identity.getName();

if ( !name.equals("spadmin")) {
```

```
        identity.setPassword("xyzzy");
        identity.setAttribute("pwdResetDate", new Date());
    }
```

This example Refresh rule removes links for a specific application from the Identities and from the system.

```
        import sailpoint.object.Identity;
        import sailpoint.object.Link;
        import java.util.*;
        import sailpoint.api.*;

        sailpoint.api.Terminator termin = new sailpoint.api.Terminator(context);
        if (identity != null ) {
            List listOfLinks=identity.getLinks();
            if(listOfLinks!=null)
            {
                for(int index=0; index<listOfLinks.size();index++)
                {
                    Link link =(Link)listOfLinks.get(index);
                    String applicationName=link.getApplicationName();
                    if "PRISM".equals(applicationName) {
                        identity.remove(link);
                        context.removeObject(link);
                    }
                }
            }
        }
```

This example Refresh rule runs as a preRefreshRule to turn off the negative assignment flag for roles that have been revoked by a certification. This allows them to be reassigned by an assignment rule running during the refresh. Other conditions could be added to restrict it to only certain roles or certain Identities. (This use case would be unusual, since most organizations want certification decisions to supersede automatic assignments.)

```
        import sailpoint.object.RoleAssignment;

        if (identity != null ) {
            List roleAssignments = identity.getRoleAssignments();
            Boolean flag = false;
            if (roleAssignments != null) {
                for (RoleAssignment roleAssignment : roleAssignments) {
                    if (roleAssignment.isNegative()){
                        roleAssignment.setNegative(flag);
                        roleAssignment.setSource("Rule");
                    }
                }
            }
        }
```

AccountGroupRefresh/GroupAggregationRefresh

The AccountGroupRefresh rule type was renamed in version 6.2 to GroupAggregationRefresh to more generically apply to all group object aggregation processes.

Description

An AccountGroupRefresh or GroupAggregationRefresh rule runs during an Account Group Aggregation task. It allows custom manipulation of group attributes while the group is being refreshed (on both create and update).

NOTE: This rule runs for every group object involved in the aggregation task, so time-intensive operations performed in it can have a negative impact on task performance.

Definition and Storage Location

This rule is specified as a task argument to an account group aggregation task. In earlier product versions, it could only be added to the task's XML, but version 6.4 introduced a UI task argument for it.

> Setup -> Tasks -> select existing or create new Account Group Aggregation task -> Group Aggregation Refresh Rule

The rule name is recorded in the taskDefinition XML as:

```
<entry key="accountGroupRefreshRule" value ="[Account Group Refresh Rule Name]">
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
environment	java.util.Map	Arguments passed to the aggregation or refresh task
obj	sailpoint.object.ResourceObject	Reference to the resourceObject from the application
accountGroup	sailpoint.object.ManagedAttribute	Reference to the account group being refreshed
groupApplication	sailpoint.object.Application	Reference to the application being aggregated

Outputs:

Argument	Type	Purpose
accountGroup	sailpoint.object.ManagedAttribute	The refreshed (modified) account group object

Example

This example AccountGroupRefresh/GroupAggregationRefresh rule extracts the first DN listed in the account group's "owner" attribute and parses the user name out of that string to identify the account group owner. It sets the Identity corresponding to that name as the account group owner and returns the account group.

```
import java.util.List;
import java.util.ArrayList;
import sailpoint.object.ResourceObject;
import sailpoint.object.AccountGroup;
import sailpoint.object.Identity;

String ownerDN = null;
String ownerName = null;
Identity identity = null;
Object owner = obj.getAttribute("owner");
```

```
if(owner instanceof List){
      ownerDN = (String)owner.get(0);
}else{
      ownerDN = (String)owner;
}

if(ownerDN != null){
      ownerName = ownerDN.substring(ownerDN.indexOf("uid=")+4,ownerDN.indexOf(","));
}

if (null != ownerName) {
      identity = context.getObjectByName(Identity.class, ownerName);
}

if (null != identity) {
      accountGroup.setOwner(identity);
}

return accountGroup;
```

AccountSelector

Description

The AccountSelector rule was introduced in IdentityIQ version 6.3 to support provisioning of entitlements through role assignments when a user holds more than one account on the target application. It provides the logic for selecting a target account for provisioning entitlements for an IT role (or any role type with an entitlement profile).

Account selector rules run during an identity refresh task with the **Provision assignments** option selected, when a business role is assigned which has required IT roles that specify these rules. This rule must provide the logic for choosing the account to which the entitlement should be provisioned. Account selector rules also run to chose a target account when a role is requested through Lifecycle Manager; if it does not select a target account, the LCM requester is prompted to select one from a list in the UI.

One or more account selector rules can be specified for each IT role; the system supports a global rule which applies to all applications involved in the role profile as well as a rule per application.

Definition and Storage Location

AccountSelector rules can be selected and specified through the Role Editor on an IT role (or any role type that supports automatic detection with profiles).

> Setup -> Roles -> edit or create an IT role -> Provisioning Target Account Selector Rules section -> General Rule or rule per application

The rules are referenced in the Bundle XML inside the attributes map entry for key "accountSelectorRules". Application-specific rules also include an ApplicationRef pointing to the application itself.

```
<entry key="accountSelectorRules">
  <value>
   <AccountSelectorRules>
     <ApplicationAccountSelectorRules>
```

```
        <ApplicationAccountSelectorRule>
          <ApplicationRef>
              <Reference class="sailpoint.object.Application"
   id="ff8080814612b067014612b07cbe0005" name="Financials"/>
          </ApplicationRef>
          <RuleRef>
              <Reference class="sailpoint.object.Rule"
   id="ff808081461af94d01461fdc6d1c0233" name="financials link selector"/>
          </RuleRef>
        </ApplicationAccountSelectorRule>
      </ApplicationAccountSelectorRules>
    </AccountSelectorRules>
  </value>
</entry>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
source	String (sailpoint.object.Source enum value)	Enum value defining the source of the request (UI, LCM, Task, etc.)
role	sailpoint.object.Bundle	The IT role being provisioned
identity	sailpoint.object.Identity	The Identity to whom the role is being provisioned
application	sailpoint.object.Application	The Target application on which the entitlements will be provisioned
links	ArrayList of sailpoint.object.Link objects	List of all available links held by the Identity
isSecondary	Boolean	True if this is not the first assignment of this role to this user
project	sailpoint.object.Provisioning Project	Provisioning project for the provisioning request
accountRequest	sailpoint.object.AccountReq uest	Account request containing details to be provisioned to the selected target account
allowCreate	Boolean	True if account creation is allowed (i.e. if the system can accept and act upon the return from the rule of a new Link with no nativeIdentity

Outputs:

Argument	Type	Purpose
selection	sailpoint.object.Link or String	Can return any of these: • one of the available Links (accounts) currently held by the the Identity • a Link with a null nativeIdentity value – this tells the system to create a new Link (any values on the returned Link are ignored and the Link is created based on the role and application provisioning

		policies)
		• a null value – causes the system to prompt the requester for an account selection
		• the string "prompt" – tells the system to prompt the requester for an account selection
		The difference between null and "prompt" is that "prompt" forces the prompting to occur per IT role so that if there are multiple IT roles involved in the role assignment which all target the same application, a separate target account can be specified for each IT role. Null causes the system to obey configuration settings as they have been specified in LCM and on the role itself, so the prompting may be at the IT role level or at the business role level, depending on that configuration.

Example

This is an example AccountSelector rule which acts differently depending on the source of the request. For requests with a source of UI or LCM, it returns a null so the user will be prompted to select an account in the UI. Otherwise, it look for and returns the first non-privileged account Link found, if one exists for the user. (The app2_privileged attribute on the target application designates whether the account is privileged or not.)

```
import sailpoint.object.Link;


if ("UI".equals(source) || "LCM".equals(source))
     return null;

if (null != links) {
    for (Link link : links) {

       if ("false".equals(link.getAttribute("app2_privileged"))) {
            return link;
       }
    }
}
return null;
```

Certification Rules

Certification Rules run during the certification creation process and during the certification lifecycle. These rules allow customization of behavior around the processing of the certification including:

- exclusion of items from a certification
- assignment of certifiers
- population of custom fields on a certification entity or item
- management of activities that occur during certification phase changes, including closing or second-level signoff

- control of processing when items are marked complete
- handling of escalations
- management of certification events (triggering and determining identities to which it applies)

Rules used during the Certification process include:

- CertificationExclusion
- CertificationPreDelegation
- Certifier
- CertificationItemCustomization
- CertificationEntityCustomization
- CertificationPhaseChange
- CertificationEntityRefresh
- CertificationEntityCompletion
- CertificationItemCompletion
- CertificationAutomaticClosing
- CertificationSignOffApprover
- WorkItemEscalationRule
- IdentityTrigger
- IdentitySelector

NOTE: One of the input parameters passed to most certification rules is the CertificationContext, which is an interface used to create the certification. This was more necessary in early versions of IdentityIQ, when the certification specification details were not readily accessible through the certification itself. Currently, this parameter will generally not be used in custom rules; attributes available through it are more easily accessed through the Certification and its connected (parent and child) objects.

CertificationExclusion

Description

A CertificationExclusion rule is used to exclude specific items from a Certification based on an evaluation of the entity being certified or the certifiable items that are part of the certification. The certificationEntity depends on the type of certification being run and can be a bundle (role), account group, or Identity. CertificationItems are dependent on the entity and the certification type. For example, for Identities, items can be roles or entitlements; for account groups, they can be permissions or members. For roles, they are required/permitted roles, entitlements, or members.

The rule is passed two lists: items and itemsToExclude. Items includes all CertificationItems belonging to the CertificationEntity that are identified for inclusion in the certification. The rule must remove items from that list to exclude them from the certification and must put them in the itemsToExclude list to make them appear in the Exclusions list for the certification.

NOTE: This rule runs for each CertificationEntity identified by the certification specification, so complex processing included in this rule can significantly impact the performance of certification generation. For

continuous certifications, this rule runs each time the certification returns to the CertificationRequired state and the Refresh Continuous Certifications task runs.

Definition and Storage Location

This rule is specified in the UI during creation of a new certification. It is an option on the Advanced page of the specification.

> Setup -> Certifications -> Create new certification (any type) -> Advanced -> Exclusion Rule

The exclusion rule name is recorded in the attributes map in the CertificationDefinition XML.

```
<entry key="exclusionRuleName" value="[CertificationExclusion Rule Name"/>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
entity	sailpoint.object.AbstractCertifiableEntity	The entity being considered for certification: a Bundle, Account Group, or Identity object
certification	sailpoint.object.Certification	The current certification object being constructed
certContext	sailpoint.object.CertificationContext	The CertificationContext interface used in generating this certification (rarely used within rules; the values accessible through this are also available on the certification or certificationDefinition)
items	List <sailpoint.object.Certifiable>	List of Certifiable items for a given identity; this rule must remove items from this list to prevent them from being certified
itemsToExclude	List <sailpoint.object.Certifiable>	List of Certifiable items to be excluded from the certification; this rule must add items to this list to have them included in the Exclusions list visible from the certification after it is generated
state	java.util.Map	Map in which any data can be stored; shared across multiple rules in the certification generation process
identity	sailpoint.object.AbstractCertifiableEntity	A second copy of the AbstractCertifiableEntity if it is an Identity object; this is passed in for backward compatibility only; newly written rules should reference the *entity* argument instead

Outputs:

Argument	Type	Purpose

explanation	String	An optional explanation describing why the entity's items were excluded; this is shown on the Exclusions list for each item excluded from the certification; if rule excludes items for different entities for different reasons, this can identify the applicable exclusion conditions when the exclusion list is examined

Example

This example rule checks for inactive identities or identities identified as "Contractors" and removes all certification items if either check evaluates as true.

```
import sailpoint.object.Identity;

log.trace("Entering Exclusion Rule.");
String explanation = "";

Identity currentUser = (Identity) entity;

if ( currentUser.isInactive()) {
   log.trace("Inactive User: " + currentUser.getDisplayName());
   log.trace("Do not certify.");
   itemsToExclude.addAll(items);
   items.clear();
   explanation = "Not certifying inactive users";
} else if (currentUser.getAttribute("status").equals("Contractor")) {
   log.trace("Identity is Contractor: " + currentUser.getDisplayName());
   log.trace("Do not certify.");
   itemsToExclude.addAll(items);
   items.clear();
   explanation = "Not certifying contractors";
} else {
   log.trace("Active Employee: " + currentUser.getDisplayName());
   log.trace("Do certify.");
    }
return explanation;
```

CertificationPreDelegation

Description

A CertificationPreDelegation rule runs during certification generation. It runs once for every CertificationEntity to determine whether the entity should be pre-delegated to a different certifier. This rule can also be used to reassign entities to a different certifier. The difference between reassignment and delegation is that reassigned certifications do not return to the original certifier for review and approval when the assignee has completed signoff and delegated items do.

NOTE: This rule runs for each entity identified by the certification specification, so complex processing included in this rule can significantly impact the performance of certification generation. For continuous certifications, this rule runs each time the certification returns to the CertificationRequired state and the Refresh Continuous Certifications task runs.

Definition and Storage Location

This rule is specified in the UI during creation of a new certification. It is an option on the Advanced page of the specification.

Setup -> Certifications -> Create new certification (any type) -> Advanced -> Pre-Delegation Rule

The PreDelegation rule name is recorded in the attributes map in the CertificationDefinition XML.

```
<entry key="preDelegationRuleName" value="[CertificationPreDelegation Rule Name]"/>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
certification	sailpoint.object.Certification	Reference to the Certification object being created
entity	sailpoint.object.Certification Entity	Reference to the CertificationEntity object being considered for predelegation
certContext	sailpoint.api.CertificationContext	Reference to the CertificationContext interface being used to create this certification (rarely used within rules; the values accessible through this are also available on the certification or certificationDefinition)
state	java.util.Map	Map in which any data can be stored; shared across multiple rules in the certification generation process

Outputs:

Argument	Type	Purpose
map	java.util.Map	A map of values including the following entries: • **recipient**: Identity object to whom the certificationEntity should be delegated (recipient or recipientName must be specified) • **recipientName**: (string) name of the recipient identity (alternative to recipient) • **description**: (string) description for delegation (if not provided, is generated as "Certify [CertificationEntityName]") • **comments**: (string) comments for the recipient of the delegation/reassignment • **certificationName**: (String) name of the new certification to which this entity is being reassigned (only needed for reassignment, not delegation) • **reassign**: (Boolean) flag indicating whether this is a reassignment or delegation (true=>reassignment)

Example

This example PreDelegation rule delegates the certification responsibility in a manager certification to each employee who reports to the manager. Each employee first evaluates and certifies their own access; then their decisions are returned to the manager for review and approval or modification.

```
import sailpoint.object.Identity;

Map results = new HashMap();
String theCertifiee = entity.getIdentity();

results.put("recipientName", theCertifiee);
results.put("description", "Please certify your own access");

results.put("comments", "This is the access currently granted to you: " + theCertifiee
+ ". Please determine whether it is appropriate for your job function.";

return results;
```

Certifier

Description

The Certifier rule is used only with Advanced certifications that are certifying members of GroupFactory-generated Groups. It identifies the certifier for each Group. This rule runs once for each Group generated from the specified GroupFactory; if the certification includes more than one GroupFactory, a separate rule can be specified for each GroupFactory.

Definition and Storage Location

The Certifier rule is specified in the UI during creation of a new certification. It is an option on the Basic page of an Advanced Certification specification.

Setup -> Certifications -> Create new Advanced certification -> Basic -> select Group Factory to Certify -> Certifier Rule

The rule is recorded in the "factoryCertifierMap" within the CertificationDefinition's attributes map. The key is the ID of the groupFactory and the value is the name of the Certifier rule applied to groups from that groupFactory:

```
<entry key="factoryCertifierMap">
  <value>
    <Map>
      <entry key="402846023a65e596013a65e719ff029f" value="[Certifier Rule Name]"/>
    </Map>
  </value>
</entry>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose

factory	sailpoint.object.GroupFactory	The groupFactory object that created the group(s) being certified
group	sailpoint.object.GroupDefinition	The group object whose members are being assigned a certifier by the rule
state	java.util.Map	Map in which any data can be stored; shared across multiple rules in the certification generation process

Outputs:

Argument	Type	Purpose
certifiers	String, sailpoint.object.Identity, list of strings or list of Identities	Identifies the Certifier(s) for each group created from the groupFactory; can return an identity name, a CSV list of identity names, an Identity object, a List of Identity objects, or a List of Identity names

Example

This example Certifier rule assigns the group owner as the Certifier for each group. If no group owner is specified, it assigns the certification to the Administrator. Note that the owner is returned as an Identity object and the Administrator is returned as a string Identity name; this is possible due to the flexible nature of this rule's return value.

```
import sailpoint.object.Identity;

Identity groupOwner = group.getOwner();

if (groupOwner != null) {
    return groupOwner;
} else {
    return "spadmin";
}
```

CertificationEntityCustomization

Description

A CertificationEntityCustomization rule runs when a certification is generated. It allows the CertificationEntity to be customized; for example, default values can be calculated for the custom fields. This rule is generally used only when custom fields have been added to CertificationEntity for the installation. It runs for every CertificationEntity in a certification.

NOTE: The CertificationItemCustomization rule (discussed next) runs for each certifiable item attached at a certificationEntity before that entity's CertificationEntityCustomization rule runs.

Definition and Storage Location

The CertificationEntityCustomization rule is specified in the System Configuration XML or can be specified in the XML of individual CertificationDefinitions. If specified in the System Configuration, it runs for every certification created.   If specified in a CertificationDefinition (by editing the XML directly after it is generated based on the UI

certification specification), it applies only to certifications generated from that definition. In either case, it is added to the attributes map.

```
<entry key="certificationEntityCustomizationRule" value="[Cert Entity Customization
Rule Name]"/>
```

This rule cannot be written through the UI Rule Editor; it must be written in XML and imported into the system.

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
certification	sailpoint.object.Certification	Reference to the Certification to which the item is being added
certifiableEntity	sailpoint.object.AbstractCertifiableEntity	Reference to the AbstractCertifiableEntity from which the certifiable was retrieved (Bundle, Identity, or AccountGroup object)
entity	sailpoint.object.CertificationEntity	Reference to the certificationEntity to be customized
certContext	sailpoint.api.CertificationContext	CertificationContext being used to build the certification (rarely used in a rule)
state	java.util.Map	A Map that can be used to store and share data between executions of this rule during a single certification generation process; rules executed in the same certification generation share this state map, allowing data to be passed between them

Outputs: None. The CertificationEntity object passed as parameter to the rule should be edited directly by the rule.

Example

This example CertificationEntityCustomization rule sets default values for the custom attributes defined for CertificationEntity if the certifiableEntity is an Identity and Identity is part of the Accounting department.

```
if (certifiableEntity instanceof Identity) {
   Identity identity = (Identity) certifiableEntity;
   if ("Accounting".equals(identity.getAttribute("department"))) {
      entity.setCustom1("custom attribute 1");
      entity.setCustom2("custom attribute 2");

      Map customMap = new HashMap();
      customMap.put("LevelNum", 42);
      entity.setCustomMap(customMap);
   }
} else {
   return null;
}
```

CertificationItemCustomization

Description

A CertificationItemCustomization rule is run when a certification is generated. It allows the CertificationItem to be customized; for example, default values can be calculated for the custom fields. This rule is generally used only when custom fields have been added to CertificationItem for the installation.

NOTE: The CertificationItemCustomization rule runs for each certifiable item attached at a certificationEntity before that entity's CertificationEntityCustomization rule (discussed above) runs.

Definition and Storage Location

The CertificationItemCustomization rule is specified in the System Configuration XML or can be specified in the XML of individual CertificationDefinitions. If specified in the System Configuration, it runs for every certification created.   If specified in a CertificationDefinition (by editing the XML directly after it is generated based on the UI certification specification), it applies only to certifications generated from that definition. In either case, it is added to the attributes map.

```
<entry key="certificationItemCustomizationRule" value="[Cert Item Customization Rule
Name]"/>
```

This rule cannot be written through the UI Rule Editor; it must be written in XML and imported into the system.

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
certification	sailpoint.object.Certification	Reference to the Certification to which the item is being added
certifiable	sailpoint.object.Certifiable	Reference to the Certifiable item being created into a CertificationItem
certifiableEntity	sailpoint.object.AbstractCertifiableEntity	Reference to the AbstractCertifiableEntity from which the certifiable was retrieved (Bundle, Identity, or AccountGroup object)
item	sailpoint.object.CertificationItem	Reference to the certificationItem to be customized
certContext	sailpoint.api.CertificationContext	CertificationContext being used to build the certification (rarely used in a rule)
state	java.util.hashMap	A Map that can be used to store and share data between executions of this rule during a single certification generation process; rules executed in the same certification generation share this state map, allowing data to be passed between them

Outputs: None. The CertificationItem object passed as parameter to the rule should be edited directly by the rule.

Example

This example CertificationItemCustomization rule checks whether the certifiableEntity is an Identity. If it is, it uses a custom identity attribute value to look up an entry in the state map or it runs a query to populate the state map (this is populated into state so this query only needs to run once for the whole certification). It then sets a custom attribute on the CertificationItem based on that cross-reference.

```
if (certifiableEntity instanceof Identity) {
   if (state = null) {
       // Load levelCode position mappings into state from a Custom object.
       state = new HashMap();
       Custom mappings = context.getObjectByName(Custom.class,
         "LevelCodePositionMappings");
       state.putAll(mappings.getAttributes());
   }
   Identity ident = (Identity) certifiableEntity;
   String levelCode = ident.getAttributes("levelCode");
   String level = state.get(levelCode);
   item.setCustom1(level);

}
```

This example rule assumes that a custom object has been imported into IdentityIQ that contains the level code position mappings; it might look like this:

```
<Custom name="LevelCodePositionMappings">
  <Attributes>
    <Map>
      <entry key="302" value="Supervisor"/>
      <entry key="443" value="Manager"/>
    </Map>
  </Attributes>
</Custom>
```

CertificationPhaseChange

Description

CertificationPhaseChange rules allow custom processing to occur at the beginning of any new certification phase. They are connected to the certification definition as the Period Enter Rule for any certification phase (e.g. Active Period Enter Rule, Challenge Period Enter Rule, etc.). Continuous certifications provide the rules with both the Certification and the CertificationItem, since individual items are phased separately for continuous certifications. Normal certifications only provide the Certification, since the whole certification is phased as a unit.

CertificatonPhaseChange rules are commonly used for actions like:

- creating a data snapshot to send to an external system
- sending an update or report to a certification monitoring team

- (Active Period Enter Rule) reporting on how long it took a certification to generate by comparing rule-fire time to certification start timestamp)
- (Active Period Enter Rule) pre-deciding certain line items in the certification (can be overridden during review)
- (Challenge Period Enter Rule) emailing managers that they should be expecting challenges to revocations

Definition and Storage Location

CertificationPhaseChange rules are specified in the UI during creation of a new certification specification. These are options on the Lifecycle page of any certification specification.

Setup -> Certifications -> Create new certification (any type) -> Lifecycle -> ___ Period Enter Rule

The rule is recorded in the CertificationDefinition's attributes map according to the phase with which it is associated.

Active: `<entry key="certificationActivePhaseEnterRule" value="[CertificationPhaseChange Rule Name]"/>`
Challenge: `<entry key="certificationChallengePhaseEnterRule"… />`
Remediation: `<entry key="certificationRemediationPhaseEnterRule"… />`
End: `<entry key="certificationFinishPhaseEnterRule"… />`

In addition to the Period Enter Rules that are offered in the UI, period exit rules can be specified for the Active, Challenge, and Remediation phases using these entries in the Certification XML:

Active: `<entry key="certificationActivePhaseExitRule"… />`
Challenge: `<entry key="certificationChallengePhaseExitRule"… />`
Remediation: `<entry key="certificationRemediationPhaseExitRule"… />`

NOTE: For traditional certifications, each of these rules runs once per certification. For continuous certifications, these each run once per certificationItem each time the certification item enters the corresponding phase; this includes the Active Period Enter Rule, which runs once per certificationItem when it reaches the CertificationRequired state.

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
certification	sailpoint.object.Certification	Certification object undergoing phase transition
certificationItem	sailpoint.object.CertificationItem	CertificationItem undergoing phase transition; only passed in for transitions of continuous certifications, where certificationItems are phased individually
previousPhase	sailpoint.object.Certification.Phase enumeration	phase being exited (may be null)

	(Staged, Active, Challenge, Remediation, End)	
nextPhase	sailpoint.object.Certification.Phase enumeration	phase to which the certification is being transitioned

Outputs: None. This rule is expected to act directly on the certification or certificationItem passed to the rule.

Example

This example End Phase Enter Rule (an instance of a CertificationPhaseChange Rule) looks at all certificationItems as the certification enters the End Phase and automatically rejects, causing remediation of, all undecided certificationItems. It then invokes Certificationer methods to refresh and sign the certification.

```
import sailpoint.object.Certification;
import sailpoint.object.CertificationAction;
import sailpoint.object.CertificationEntity;
import sailpoint.object.CertificationItem;
import sailpoint.api.SailPointContext;
import sailpoint.api.Certificationer;
import sailpoint.api.certification.RemediationManager;
import sailpoint.tools.GeneralException;
import sailpoint.tools.Message;
import sailpoint.tools.Util;


private void rejectUnfinished(CertificationItem certificationItem)
     throws GeneralException {

   List children = certificationItem.getItems();
   if (Util.isEmpty(children)) {
       // a leaf item, must have a status, but only reject if there
       // is not already a decision.
       if ((null == certificationItem.getAction()) || (null ==
certificationItem.getAction().getStatus())) {
           RemediationManager remedMgr = new RemediationManager(this.context);

           RemediationManager.ProvisioningPlanSummary planSummary =
remedMgr.calculateRemediationDetails(certificationItem,
               CertificationAction.Status.Remediated);

           CertificationAction.RemediationAction remediationAction = planSummary !=
null ? planSummary.getAction() : null;

           certificationItem.remediate(context, null, null, remediationAction, null,
null, null, null, null);
       }
   }
   else {
       // a parent item, does not need a status
       List childItems = certificationItem.getItems();

       if (childItems != null) {
           for (int i=0; i<childItems.size(); ++i) {
               CertificationItem childItem = (CertificationItem) children.get(i);
               rejectUnfinished(childItem);
           }
       }
   }
```

```
    }

    private void showErrorsIfExists(List errors) {
        if (!Util.isEmpty(errors)) {
            Iterator errorsIterator = errors.iterator();
            while (errorsIterator.hasNext()) {
                Message error = (Message) errorsIterator.next();
                System.out.println(error.getLocalizedMessage());
            }
        }
    }

    private Certification refreshCert(SailPointContext context, Certificationer
    certificationer, Certification certification)
            throws GeneralException {

        List messages = certificationer.refresh(certification);
        showErrorsIfExists(messages);

        return context.getObjectById(Certification.class, certification.getId());
    }


    // Main rule logic starts here
    List entities = certification.getEntities();
    if (entities != null) {
        Iterator entitiesIterator = entities.iterator();
        while (entitiesIterator.hasNext()) {
            CertificationEntity entity = (CertificationEntity) entitiesIterator.next();
            List items = entity.getItems();
            if (items != null) {
                Iterator itemsIterator = items.iterator();
                while(itemsIterator.hasNext()) {
                    CertificationItem childItem = (CertificationItem) itemsIterator.next();
                    rejectUnfinished(childItem);
                }
            }
        }
    }

    Certificationer certificationer = new Certificationer(context);
    certification = refreshCert(context, certificationer, certification);

    List errors = certificationer.sign(certification, null);
    showErrorsIfExists(errors);
```

CertificationEntityRefresh

Description

The CertificationEntityRefresh rule runs when any certificationEntity is refreshed. Refresh of a certificationEntity occurs when decisions made for that entity or any of its certificationItems is saved. The rule's logic could, for example, be used to copy a custom field value from one item to another or from the CertificationEntity down to its certificationItems.

This rule was created to permit custom logic around CertificationItem extended attributes. In practice these extended attributes and this rule type are seldom used.

Definition and Storage Location

The certification entity refresh rule is specified in the System Configuration XML or can be specified in the XML of individual CertificationDefinitions. If specified in the System Configuration, it is applied to every certification, so it runs every time a certification entity is refreshed on any certification of any type.   If specified in a CertificationDefinition (by editing the XML directly after it is generated based on the UI certification specification), it applies only to certifications generated from that definition. In either case, it is added to the attributes map.

```
<entry key="certificationEntityRefreshRule" value="[Cert Entity Refresh Rule Name]"/>
```

This rule cannot be written through the UI Rule Editor; it must be written in XML and imported into the system.

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
certification	sailpoint.object.Certification	Reference to the certification object to which this entity belongs
entity	sailpoint.object.CertificationEntity	Reference to the certificationEntity object that was refreshed, causing launch of this rule

Outputs: None. The rule can directly modify the CertificationEntity object passed to it as a parameter.

Example

This example certificationEntityRefresh rule copies values from custom attributes on the certification entity down to the certification items associated with that entity.

```
import sailpoint.object.CertificationItem;

String custom1 = entity.getCustom1();
String custom2 = entity.getCustom2();
Map customMap = entity.getCustomMap();

if (null != entity.getItems()) {
    for (Iterator it=entity.getItems().iterator(); it.hasNext(); ) {
        CertificationItem item = (CertificationItem) it.next();
        item.setCustom1(custom1);
        item.setCustom2(custom2);
        item.setCustomMap(customMap);
    }
}
```

CertificationEntityCompletion

Description

A Certification Entity completion rule is run when a CertificationEntity is refreshed and has been determined to be otherwise complete (i.e. all certification items on the entity are complete). The certification refresh process

occurs when changes to an access review are saved by the user. This rule determines whether the entity is still missing any information. For example, the entity may require a 'classification' value to be present in a custom field to be complete. If errors are found, the error messages (either plain-text messages or keys that map to messages in the message catalog) are added to a List and returned to the caller, which tells IdentityIQ to mark the Entity as still incomplete.

This rule was created to permit custom logic around CertificationItem extended attributes. In practice these extended attributes and this rule type are seldom used.

Definition and Storage Location

The certification entity completion rule is specified in the System Configuration XML or can be specified in the XML of individual CertificationDefinitions. If specified in the System Configuration, it is applied to every certification, so it runs every time a certification entity is completed on any certification of any type.   If specified in a CertificationDefinition (by editing the XML directly after it is generated based on the UI certification specification), it applies only to certifications generated from that definition. In either case, it is added to the attributes map.

```
<entry key="certificationEntityCompletionRule" value="[Cert Entity Completion Rule
Name]"/>
```

This rule cannot be written through the UI Rule Editor; it must be written in XML and imported into the system.

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
certification	sailpoint.object.Certification	A reference to the Certification object being refreshed
entity	sailpoint.object.CertificationEntity	A reference to the CertificationEntity object being refreshed.
state	java.util.Map	Map in which any data can be stored; shared across multiple rules run in the same completion process (e.g. certificationItemCompletion and CertificationEntityCompletion rules can share a state map)

Outputs:

Argument	Type	Purpose
messages	List of sailpoint.tools.message objects or strings	List of message objects or strings if errors were found (any contents in list mean that the Entity is not complete); null if entity is complete

Example

This example CertificationEntityCompletion rule performs some data validation on custom attributes: Custom1 and Custom2 must be non-null, the "priceScale" entry in the CustomMap attribute must be either "dollars" or "euro", and the "cost" entry in the CustomMap attribute must be greater than or equal to zero. It returns error messages if any of these validations fail.

```
import java.util.List;
import java.util.ArrayList;
import java.util.Map;
import java.util.HashMap;

List errors = new ArrayList();

String e1 = entity.getCustom1();
String e2 = entity.getCustom2();
String scale = null;
int cost = -1;

Map extendedMap = entity.getCustomMap();
if (null != extendedMap) {
    scale = (String) extendedMap.get("priceScale");
    Integer costInteger = (Integer) extendedMap.get("cost");
    if (null != costInteger) {
        cost = costInteger.intValue();
    }
}

if ((e1 == null) || (e1.equals(""))) {
    // plain-text message
    errors.add("The custom1 field must be filled out in order to complete this
item.");
}

if ((e2 == null) || (e2.equals(""))) {
    // key for the message in the messages catalog
    errors.add("custom2_missing_info");
}

if (scale == null) {
    // key for the message in the messages catalog, plus message arguments
    List list = new ArrayList();
    list.add("err_missing_custom_cert_info");
    list.add(entity.getIdentity());
    list.add(entity.getType());
    errors.add(list);
}

if (!(("euro".equals(scale) || "dollars".equals(scale)) || cost < 0) {
    errors.add("Cost cannot be negative and must be stated in dollars
        or euro.");
    entity.
}

return errors;
```

CertificationItemCompletion

Description

A CertificationItemCompletion rule is run when a CertificationItem is refreshed and appears to be complete. This rule determines whether the item is still missing any information. The rule returns a Boolean value: true if the item is complete according to the rule's evaluation or false if the rule found the item to be still in an incomplete state. The system then marks the item accordingly.

This rule was created to permit custom logic around CertificationItem extended attributes. In practice these extended attributes and this rule type are seldom used.

Definition and Storage Location

The certificationItemCompletion rule is specified in the System Configuration XML. It is applied to every certification, so it runs every time a certification item is completed on any certification of any type. Specify it in the SystemConfiguration attributes map with this entry:

```
<entry key="certificationItemCompletionRule" value="[Cert Item Completion Rule
Name]"/>
```

This rule cannot be written through the UI Rule Editor; it must be written in XML and imported into the system.

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
certification	sailpoint.object.Certification	A reference to the Certification object to which the Item (and entity) belong
item	sailpoint.object.CertificationItem	A reference to the CertificationItem object being completed
entity	sailpoint.object.CertificationItem	A second reference to the CertificationItem object being completed; exists as a synonym for item
state	java.util.Map	A map of values that can be shared between rules; allows passing of data between rules

Outputs:

Argument	Type	Purpose
complete	Boolean	Returns true if item is deemed complete and false if it is not

Example

This example CertificationItemCompletion rule checks that the custom1 attribute on certificationItem is not null. If it is null, the item is deemed not complete.

```
String c1= item.getAttribute("custom1");
if (c1 != null)
    return true;
else
    return false;
```

CertificationAutomaticClosing

Description

A CertificationAutomaticClosing rule can be used to apply custom logic to certifications that have not been finished by a certifier when the automatic closing date arrives (automatic closing date is configurable based on certification end date). The perform maintenance task is responsible for automatically closing certifications for which automatic closing has been enabled. Each certification set for automatic closing on or before the task's run date is identified and its automatic closing rule is run. Then the remaining auto-closing specifications are applied to any of its items still in an incomplete or unfinished state.

Definition and Storage Location

The CertificationAutomaticClosing rule is specified in the UI during creation of a new certification. It is selected on the Lifecycle page of any certification specification when Enable Automatic Closing is selected.

> Setup -> Certifications -> Create new certification (any type) -> Lifecycle -> Enable Automatic Closing -> Closing Rule

The rule is recorded in the CertificationDefinition's attributes map.

```
<entry key="certificationAutomaticClosingRule" value="[Cert Automatic Closing Rule
Name]"/>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
certification	sailpoint.object.Certification	A reference to the Certification object being closed

Outputs: None; the rule should update the certification and its entities/items directly (or it may perform actions outside the flow of the certification process, such as sending an email notice to someone about the incomplete items).

Example

This example CertificationAutomaticClosing rule sends an email to the certification owner notifying them of the items on which no decision was made. It iterates through the certification's entities and items looking for items on which no action has been taken, collecting those into a hash map. That map and an email template (created independently) that specifies the message contents for this notification are used to send the email to the owner.

```
import sailpoint.object.Identity;
import sailpoint.object.Certification;
import sailpoint.object.CertificationEntity;
import sailpoint.object.CertificationItem;
import sailpoint.object.EntitlementSnapshot;
import sailpoint.object.EmailOptions;
import sailpoint.object.EmailTemplate;
import sailpoint.object.Attributes;

// This email notification goes to the cert owner
Identity owner =
certification.getCertificationDefinition(context).getCertificationOwner(context);
Map identityMap = new HashMap();
List entities = certification.getEntities();

// Iterate through the entities on this certification
for ( CertificationEntity entity : entities ) {
    String identityName = "";
    List items = entity.getItems();
    List openItems = new ArrayList();
  // Iterate through the items for each entity
    for ( CertificationItem item : items ) {
            EntitlementSnapshot ent = item.getExceptionEntitlements();
            if ( null != ent ) {
                Attributes attrs = ent.getAttributes();
                if ( null != attrs ) {
                        List attrNames = attrs.getKeys();
                        for ( String attrName : attrNames ) {
                                String attrVal = attrs.getString(attrName);
            // items with no action attached are still open
            // and need to be in the email message
                            if (item.getAction() == null) {
                                    openItems.add(attrName + "      " + attrVal);
                                }
                        }
                }
            }
            if ( item != null )
                        context.decache(item);
        }
    Identity remediatedUser = entity.getIdentity(context);
    String identityName = remediatedUser.getDisplayableName();
    identityMap.put(identityName,openItems);

    if ( entity != null )
                context.decache(entity);
}
String templateName = "AutoClosed Cert";
EmailTemplate template = (EmailTemplate) context.getObject(EmailTemplate.class,
templateName);
template.setTo(owner.getEmail());
template.setCc("");

EmailOptions options = new EmailOptions();
```

```
options.setSendImmediate(true);
options.setNoRetry(true);
options.setVariable("certification", certification);
options.setVariable("identityMap", identityMap);

context.sendEmailNotification(template, options);
```

CertificationSignOffApprover

Description

A CertificationSignOffApprover rule is used to specify one or more additional levels of approval for a certification. When the certification is signed off, this rule runs (if one is specified for the certification) to identify the next approver to whom the certification should be forwarded for review and approval. This rule runs every time a certification is signed off, including second-level signoffs. As long as the rule returns an Identity, the certification will be forwarded to that Identity for review and signoff; when it returns null, the forwarding process terminates for the certification.

NOTE: If the logic in this rule could potentially reroute the certification to the same Identity who just signed off on it, the rule must check for this condition and return null when the new certifier matches the existing one. Otherwise, an endless loop could be created where the certification is repeatedly returned to the same certifier for another signoff, and the certification would never successfully complete.

Definition and Storage Location

The CertificationSignOffApprover rule is specified in the UI during creation of a new certification. It is selected on the Advanced page of any certification specification.

> Setup -> Certifications -> Create new certification (any type) -> Advanced -> Sign Off Approver Rule

The rule is recorded in the CertificationDefinition's attributes map.

```
<entry key="signOffApproverRuleName" value="[Cert Signoff Approver Rule Name]"/>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
certification	sailpoint.object.Certification	A reference to the Certification object being closed
certifier	sailpoint.object.Identity	Reference to the Identity who was assigned as the certifier for this certification
state	java.util.Map	A map of values that can be shared between rules; allows passing of data between rules

Outputs:

Argument	Type	Purpose
results	java.util.Map	Map containing either an Identity or Identity name with the key "identity" or "identityName", respectively. e.g.: "identity", identityObject *or* "identityName", "Adam.Kennedy"

Example

This example CertificationSignOffApprover rule forwards the certification to the certifier's manager for approval. This process continues with this rule until the certifier does not have a manager (e.g. all the way up the manager hierarchy).

```
import sailpoint.object.Identity;

// This requires approval all the up the manager hierarchy. Once we get
// to the most senior manager, approvals stop.
Identity identity = certifier.getManager();

if (identity != null) {
   Map results = new HashMap();
   results.put("identity", identity);
   return results;
} else {
   return null;
}
```

Since every signer is added to the certificationSignOffHistory immediately after the certificationSignOffApprover rule runs, this rule could be limited to only require one level of secondary signoff by checking the certification signoff history like this:

```
import sailpoint.object.Certification;
import sailpoint.object.Identity;

// if cert signoff history indicates it has already been signed off once,
// do not submit to any other levels of
approval
List history = certification.getSignOffHistory();

if (history == null || history.isEmpty()){
   Identity identity = certifier.getManager();
   Map results = new HashMap();
   results.put("identity", identity);
   return results;
}
else
    return null;
}
```

IdentityTrigger

Description

An IdentityTrigger rules apply to both Certification Events and Lifecycle Events; they determine whether the associated certification or business process (respectively) should be triggered for the Identity on which an action occurs. IdentityTrigger rules run anytime an Identity is changed in an Identity Refresh or Aggregation if the "Process Events" option is selected on the task, and they are passed the Identity as it existed before and after the change. They also run when an Identity is edited through the **Define** -> **Identities** administrator page. The rule's logic determines what attributes are evaluated, and the rule can return a True or False value; True fires the certification/business process associated with the rule and False does not.

When more than one trigger exists, they are retrieved from the database without regard to order, so their evaluation order depends on the database engine and possibly the order in which they were created in the database. Regardless, all are passed the same new and previous identity values (i.e. the effects of the one trigger's event do not feed into the next trigger's evaluation). Additionally, if multiple triggers' conditions are met in one Identity update, the events launched by the triggers are processed in the background and may occur concurrently.

Definition and Storage Location

The IdentityTrigger rule is specified in the UI during specification of a certification event or lifecycle event.

> Setup -> Certifications -> Certification Events -> New Certification Event -> Event Type: Rule -> Rule
>
> or
>
> Setup -> Lifecycle Events -> New Lifecycle Event -> Event Type: Rule -> Rule

The rule is referenced in the IdentityTrigger XML representing the event.

```
<TriggerRule>
<Reference class="sailpoint.object.Rule" id="402846023a660a1d013a8e3ba5ed12ca"
name="[IdentityTrigger Rule Name]"/>
</TriggerRule>
```

The Process Events option on the task is specified in the taskDefinition attributes map as the "processTriggers" key. This is selectable through the UI for Identity Refresh tasks but must be manually added to the taskDefinition XML for aggregation tasks.

```
<entry key="processTriggers" value="true"/>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
previousIdentity	sailpoint.object.Identity	Identity as it existed before it was updated
newIdentity	sailpoint.object.Identity	Identity as it existed after it was updated

NOTE: Either Identity can be void (previous is void on Identity creation and new is void on Identity deletion) and the rule must test for this to prevent a possible exception condition.

Outputs:

Argument	Type	Purpose
result	Boolean	True if the event should be triggered or false if it should not

Example

This example IdentityTrigger rule causes the certification or lifecycle event to fire only if the Identity's job title changes from "DBA" to "Production Manager".

```
// The instanceof operator returns false if the object is null or void
// as well as if it is a different object type.
if (!(previousIdentity instanceof Identity) || !(newIdentity instanceof Identity)) {
    return false;
}

String oldVal = previousIdentity.getAttribute("jobtitle");
String newVal = newIdentity.getAttribute("jobtitle");

return "DBA".equals(oldVal) && "Production Manager".equals(newVal);
```

IdentitySelector

Description

Like an IdentityTrigger, an IdentitySelector rule can apply to a Certification Event or a Lifecycle Event and determines whether the associated certification or business process should be run for the Identity on which an action occurs. The difference is that an IdentityTrigger rule defines the event itself whereas an IdentitySelector rule determines the set of Identities to which the event applies. Additionally, the IdentitySelector rule (or any identity selector filter) is evaluated *before* the action is examined, so if the Identity on which the action occurred is not part of the Identity selector filter, the action is ignored and the certification or business process is not fired.

Like IdentityTrigger rules, these rules only run during refresh or aggregation if the "process events" option is selected for the identity refresh or aggregation task.

IdentitySelector rules can also be used for specifying criteria for role assignment or for Advanced Policy detection. In the case of role assignment rules, if the rule returns "true", the role is assigned to the Identity. See the description of the *Policy* rule type for more information on the Policy usage of IdentitySelector rules. Role assignment and policy rules are also run by Identity Refresh tasks, though their execution is controlled by the "Refresh assigned, detected roles and promote additional entitlements" and "Check active policies" options, respectively.

Definition and Storage Location

The IdentitySelector rule is specified in the UI during specification of a certification event or lifecycle event.

Setup -> Certifications -> Certification Events -> New Certification Event -> Include Identities: Rule

or

Setup -> Lifecycle Events -> New Lifecycle Event -> Include Identities: Rule

The rule is referenced in the IdentityTrigger XML representing the event.

```
<Selector>
  <IdentitySelector>
    <RuleRef>
<Reference class="sailpoint.object.Rule" id="402846023a65e596013a65e6e0900267"
name="[IdentitySelector Rule Name]"/>
    </RuleRef>
  </IdentitySelector>
</Selector>
```

The Process Events option on the task is specified in the taskDefinition attributes map as the "processTriggers" key. This is selectable through the UI for Identity Refresh tasks but must be manually added to the taskDefinition XML for aggregation tasks.

```
<entry key="processTriggers" value="true"/>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
identity	sailpoint.object.Identity	Identity object on which the triggering action has occurred (post-change version unless change is a delete action, in which case pre-change version is passed to rule)

Outputs:

Argument	Type	Purpose
success	Boolean	True if the Identity meets the criteria for running the certification/business process or false if it does not

Example

This example IdentitySelector rule causes the event to be applied only to Identities assigned to the APAC region.

```
import sailpoint.object.Identity;

if ("APAC".equals(identity.getRegion()) {
    return true;
```

```
    } else {
        return false;
    }
```

Provisioning Rules

These rules run during the processing of provisioning requests. Some are connector specific and some apply for all connectors, as indicated in their descriptions.

BeforeProvisioning

Description

The BeforeProvisioning rule is executed immediately before the connector's provisioning method is called. This gives customer the ability to customize or react to anything in the ProvisioningPlan before the requests are sent to the underlying connectors used in provisioning. This rule is not connector-specific; it runs for all applications regardless of connector type.

Definition and Storage Location

This rule is associated to an application in the UI through the application definition.

> Applications -> Application Definition -> select an application or create a new application -> Rules ->
> Provisioning Rules section -> Before Provisioning Rule

The reference to the rule is recorded in the Application XML in the attributes map as:

```
<entry key="beforeProvisioningRule" value="[Rule Name]"/>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
plan	sailpoint.integration.ProvisioningPlan	Contains provisioning request details
application	sailpoint.object.Application	Application object containing this rule reference

Outputs: None. The rule should directly update the ProvisioningPlan passed to it.

Example

This example BeforeProvisioning Rule alters the "region" value in the plan being provisioned to change it from "Europe" to "EMEA".

```
import sailpoint.object.*;
import sailpoint.tools.*;
import sailpoint.object.ProvisioningPlan;
import sailpoint.object.ProvisioningPlan.AccountRequest;
import sailpoint.object.ProvisioningPlan.AttributeRequest;
import sailpoint.object.ProvisioningPlan.Operation;
```

```
AccountRequest acctReq = plan.getAccountRequest("TestApp");

boolean found = false;
List attributeRequests = acctReq.getAttributeRequests();
if ( attributeRequests != null ) {
    for ( AttributeRequest req : attributeRequests ) {
        String name = req.getName();
        if ( name != null && name.compareTo("region") == 0  ) {
            if ("Europe".equals(req.getValue())){
                req.setValue("EMEA");
            }
        }
    }
}
```

AfterProvisioning

Description

An application's AfterProvisioning rule is executed immediately after the connector's provisioning method is called, but only if the provisioning result is in a committed or queued state. This gives customers the ability to customize or react to anything in the ProvisioningPlan that has been sent out to specific applications after the provisioning request has been processed. This rule is not connector-specific; it runs for all applications regardless of connector type.

Definition and Storage Location

This rule is associated to an application in the UI through the application definition.

> Applications -> Application Definition -> select an application or create a new application -> Rules -> Provisioning Rules section -> After Provisioning Rule

The rule name is recorded in the attributes map of the application XML. as:

```
<entry key="afterProvisioningRule" value="[Rule Name]"/>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
plan	sailpoint.object.ProvisioningPlan	Contains provisioning request details
application	sailpoint.object.Application	Application object containing this rule reference
result	sailpoint.object.ProvisioningResult	Contains provisioning request result

Outputs: none; the rule's actions are outside the direct provisioning process so no return value is expected or used

Example

This example rule notifies the application owner if an Identity is assigned the Super User role in the application. Similar logic would apply to users being added to a specific Active Directory group, etc.

```
import sailpoint.object.*;
import sailpoint.object.ProvisioningPlan;
import sailpoint.object.ProvisioningPlan.AccountRequest;
import sailpoint.object.ProvisioningPlan.AttributeRequest;


// examine provisioning result to see if Identity has been added to Admin group
System.out.println("running after provisioning rule");

String requester;

if ( plan != null ) {
      List accounts = plan.getAccountRequests();
      if ( ( accounts != null ) && ( accounts.size() > 0 ) ) {
        for ( AccountRequest account : accounts ) {
            if (( account != null ) &&
          ( AccountRequest.Operation.Create.equals(account.getOperation())
       || AccountRequest.Operation.Modify.equals(account.getOperation()))) {
            //Check if adding someone to "super" role
                    AttributeRequest attrReq =
account.getAttributeRequest("role");
                 if (attrReq != null) {
              if ("super".equals(attrReq.getValue())) {
                   String nativeIdent = plan.getNativeIdentity();
                   List requesters = plan.getRequesters();
                   if (!(null == requesters || void == requesters)) {
                       Identity reqIdent = requesters.get(0);
                       requester = reqIdent.getName();
                   } else {
                        requester = "No requester recorded";
                   }
                   // email application owner if they find "super" role
                   Identity appOwner = application.getOwner();
                   System.out.println("owner:" + appOwner.toXml());
                   System.out.println("email:" + appOwner.getEmail());
                   String templateName = "NewSuperUser";
                   EmailTemplate template = (EmailTemplate)
context.getObject(EmailTemplate.class, templateName);
                   template.setTo(appOwner.getEmail());

                   EmailOptions options = new EmailOptions();
                   options.setSendImmediate(true);
                   options.setNoRetry(true);
                   options.setVariable("nativeIdentity", nativeIdent);
                   options.setVariable("requester", requester);

                   context.sendEmailNotification(template, options);

               }
            }
          }
        }
     }
}
```

JDBCProvision

Description

A JDBC Provision rule is only specified for an application that uses the JDBC connector and supports provisioning. It contains the application-specific provisioning logic for applications which use that connector. The JDBC connector is a generic connector that cannot know how to provision to the specific database except as instructed in custom-written logic provided a provisioning rule.

Definition and Storage Location

This rule is associated to an application in the UI through the application definition:

> Applications -> Application Definition -> Select application or create new application with Application Type: JDBC -> Rules -> Provision Rule Type: Global Provision Rule -> Provision Rule

The reference to the rule is recorded in the Application XML.

```
<entry key="jdbcProvisionRule" value="[JDBC Provision Rule Name]"/>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
application	sailpoint.object.Application	Reference to the application object
schema	sailpoint.object.Schema	Reference to the application schema
connection	java.sql.Connection	Connection object to connect to the JDBC database
plan	sailpoint.object.ProvisioningPlan	Provisioning plan containing the provisioning request(s)

Outputs:

Argument	Type	Purpose
result	sailpoint.object.ProvisioningResult	ProvisioningResult object containing the status (success, failure, retry, etc.) of the provisioning request

Example

This example JDBC rule can process account creation requests, deletion requests, and modification requests that pertain to the "role" attribute. It logs debug messages if other account request types are submitted.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.sql.Types;
import java.util.List;
import sailpoint.api.SailPointContext;
import sailpoint.connector.JDBCConnector;
```

```
import sailpoint.object.Application;
import sailpoint.object.ProvisioningPlan;
import sailpoint.object.ProvisioningPlan.AccountRequest;
import sailpoint.object.ProvisioningPlan.AttributeRequest;
import sailpoint.object.ProvisioningPlan.PermissionRequest;
import sailpoint.object.ProvisioningResult;
import sailpoint.object.Schema;
import sailpoint.tools.xml.XMLObjectFactory;
import org.apache.commons.logging.LogFactory;
import org.apache.commons.logging.Log;

Log _log = LogFactory.getLog("RuleProvisionSampleDB");

public String getAttributeRequestValue(AccountRequest acctReq, String attribute) {
    if ( acctReq != null ) {
        AttributeRequest attrReq = acctReq.getAttributeRequest(attribute);
        if ( attrReq != null ) {
            return attrReq.getValue();
        }
    }
    return null;
}


ProvisioningResult result = new ProvisioningResult();
if ( plan != null ) {
    _log.debug( "plan [" + plan.toXml() + "]" );

    List accounts = plan.getAccountRequests();
    if ( ( accounts != null ) && ( accounts.size() > 0 ) ) {
        for ( AccountRequest account : accounts ) {
          try {
            if ( AccountRequest.Operation.Create.equals( account.getOperation() ) ) {

//Ideally we should first check to see if the account already exists.
//As written, this just assumes it does not.
                _log.debug( "Operation [" + account.getOperation() + "] detected." );
                PreparedStatement statement = connection.prepareStatement( "insert into
users (login,first,last,role,status) values (?,?,?,?,?)" );
                statement.setString ( 1, (String) account.getNativeIdentity() );
                statement.setString ( 2, getAttributeRequestValue(account,"first") );
                statement.setString ( 3, getAttributeRequestValue(account,"last") );
                statement.setString ( 4, getAttributeRequestValue(account,"role") );
                statement.setString ( 5, getAttributeRequestValue(account,"status") );
                statement.executeUpdate();
                result.setStatus( ProvisioningResult.STATUS_COMMITTED );

          } else if ( AccountRequest.Operation.Modify.equals( account.getOperation() )
) {
                // Modify account request -- change role

                _log.debug( "Operation [" + account.getOperation() + "] detected." );
                PreparedStatement statement = connection.prepareStatement( "update users
set role = ? where login = ?" );
                statement.setString ( 2, (String) account.getNativeIdentity() );
                if ( account != null ) {
                    AttributeRequest attrReq = account.getAttributeRequest("role");
                    if ( attrReq != null &&
ProvisioningPlan.Operation.Remove.equals(attrReq.getOperation()) ) {
                        statement.setNull ( 1, Types.NULL );
                        _log.debug( "Preparing to execute:"+statement.toString() );
                        statement.executeUpdate();
                    } else {
```

```
                        statement.setString(1,attrReq.getValue());
                        _log.debug( "Preparing to execute:"+statement.toString() );
                        statement.executeUpdate();
                    }
                }

                result.setStatus( ProvisioningResult.STATUS_COMMITTED );

            } else if ( AccountRequest.Operation.Delete.equals( account.getOperation() )
    ) {
                    _log.debug( "Operation [" + account.getOperation() + "] detected." );
                    PreparedStatement statement = connection.prepareStatement( (String)
    application.getAttributeValue( "account.deleteSQL" ) );
                    statement.setString ( 1, (String) account.getNativeIdentity() );
                    statement.executeUpdate();
                    result.setStatus( ProvisioningResult.STATUS_COMMITTED );
            } else if ( AccountRequest.Operation.Disable.equals( account.getOperation()
    ) ) {
                    // Not supported.
                    _log.debug( "Operation [" + account.getOperation() + "] is not
    supported!" );
            } else if ( AccountRequest.Operation.Enable.equals( account.getOperation() )
    ) {
                    // Not supported.
                    _log.debug( "Operation [" + account.getOperation() + "] is not
    supported!" );
            } else if ( AccountRequest.Operation.Lock.equals( account.getOperation() ) )
    {
                    // Not supported.
                    _log.debug( "Operation [" + account.getOperation() + "] is not
    supported!" );
            } else if ( AccountRequest.Operation.Unlock.equals( account.getOperation() )
    ) {
                    // Not supported.
                    _log.debug( "Operation [" + account.getOperation() + "] is not
    supported!" );
            } else {
                    // Unknown operation!
                    _log.debug( "Unknown operation [" + account.getOperation() + "]!" );
            }

        }
        catch( SQLException e ) {
          _log.error( e );
          result.setStatus( ProvisioningResult.STATUS_FAILED );
          result.addError( e );
        }
      }
    }
}

_log.debug( "result [" + result.toXml(false)+ "]");
return result;
```

JDBCOperationProvisioning

Description

A JDBC Operation Provisioning rule is only specified for an application that uses the JDBC connector and supports provisioning. It contains application- and operation-specific provisioning logic for the application. The

JDBC connector is a generic connector that cannot know how to provision to the specific database except as instructed in custom-written logic provided a provisioning rule.

Separate JDBCOperationProvisioning rules are created for account enabling, account disabling, account deletion, account unlocking, account creation, and account modification. This rule type was introduced in IdentityIQ version 6.1 as an alternative to specifying a single JDBCProvision rule which performs all of these operations for the application.

Definition and Storage Location

This rule is associated to an application in the UI through the application definition:

> Applications -> Application Definition -> Select application or create new application with Application Type: JDBC -> Rules -> Provision Rule Type: By Operation Rules -> Enable Provision Rule or Disable Provision Rule, etc.

The reference to the rule is recorded in the Application XML.

```
<entry key="jdbcEnableProvisioningRule" value="[JDBCOperationProvision Rule Name]"/>
<entry key="jdbcDisableProvisioningRule" value="[JDBCOperationProvision Rule Name]"/>
<entry key="jdbcCreateProvisioningRule" value="[JDBCOperationProvision Rule Name]"/>
<entry key="jdbcDeleteProvisioningRule" value="[JDBCOperationProvision Rule Name]"/>
<entry key="jdbcModifyProvisioningRule" value="[JDBCOperationProvision Rule Name]"/>
<entry key="jdbcUnlockProvisioningRule" value="[JDBCOperationProvision Rule Name]"/>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
application	sailpoint.object.Application	Reference to the application object
schema	sailpoint.object.Schema	Reference to the application schema
connection	java.sql.Connection	Connection object to connect to the JDBC database
plan	sailpoint.object.ProvisioningPlan	Provisioning plan containing the provisioning request(s) to be processed
request	sailpoint.object.ProvisioningPlan.AbstractRequest	AbstractRequest object containing the account request (or object request, in the case of group provisioning) to be processed

Outputs:

Argument	Type	Purpose
result	sailpoint.object.ProvisioningResult	ProvisioningResult object containing the status (success, failure, retry, etc.) of the provisioning request

Example

This example JDBC Operation Provisioning rule can process an account creation request.

NOTE: This is the same rule code found above in the JDBC Provisioning Rule within the account create operation code block. Separate rules would then be created for the account modify, delete, unlock, etc. operations.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.sql.Types;
import java.util.List;
import sailpoint.api.SailPointContext;
import sailpoint.connector.JDBCConnector;
import sailpoint.object.Application;
import sailpoint.object.ProvisioningPlan;
import sailpoint.object.ProvisioningPlan.AccountRequest;
import sailpoint.object.ProvisioningPlan.AttributeRequest;
import sailpoint.object.ProvisioningResult;
import sailpoint.object.Schema;
import sailpoint.tools.xml.XMLObjectFactory;
import org.apache.commons.logging.LogFactory;
import org.apache.commons.logging.Log;


public String getAttributeRequestValue(AccountRequest acctReq, String attribute) {
      if ( acctReq != null ) {
            AttributeRequest attrReq = acctReq.getAttributeRequest(attribute);
            if ( attrReq != null ) {
          return attrReq.getValue();
      }
    }
    return null;
}


AccountRequest acctRequest = (AccountRequest) request;
ProvisioningResult result = new ProvisioningResult();
try {
            //Ideally we should first check to see if the account already exists.
            //As written, this just assumes it does not.

            log.debug( "Operation [" + acctRequest.getOperation() + "] detected." );
            PreparedStatement statement = connection.prepareStatement( "insert into
users (login,first,last,role,status) values (?,?,?,?,?)" );
            statement.setString (1, (String) acctRequest.getNativeIdentity() );
            statement.setString (2, getAttributeRequestValue(acctRequest,"first") );
            statement.setString (3, getAttributeRequestValue(acctRequest,"last") );
            statement.setString (4, getAttributeRequestValue(acctRequest,"role") );
            statement.setString (5, getAttributeRequestValue(acctRequest,"status") );
            statement.executeUpdate();
            result.setStatus( ProvisioningResult.STATUS_COMMITTED );


 }
 catch( SQLException e ) {
     log.error( e );
            result.setStatus( ProvisioningResult.STATUS_FAILED );
            result.addError( e );
 }

log.debug( "result [" + result.toXml(false)+ "]");
return result;
```

SapHrProvision

Description

The SAP HR/HCM connector was introduced in version 7.0, so this rule applies only to versions 7.0+.

Two options are available for configuring provisioning to SAP HR/HCM applications:

1. A single rule (of type SapHrProvision) which contains all supported provisioning operations, or
2. A collection of operation-specific rules (of type SapHrOperationProvisioning, discussed below), one per supported provisioning operation

An SAP HR Provision rule is specified for an application that uses the SAP HR/HCM connector if it will support provisioning. It contains the installation-specific provisioning logic for provisioning to the SAP HR application.

Definition and Storage Location

This rule is associated to an application in the UI through the application definition:

> Applications -> Application Definition -> select application or create new application with Application Type: SAP HR/HCM -> Rules -> Provision Rule Type: Global Provision Rule -> Provision Rule

The reference to the rule is recorded in the Application XML.

```
<entry key="saphrProvisionRule" value="[Provision Rule Name]"/>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
application	sailpoint.object.Application	Reference to the application object
schema	sailpoint.object.Schema	Reference to the application schema
destination	com.sap.conn.jco.JCoDestination	A connected and ready to use SAP destination object that can be used to call BAPI function modules and call to SAP tables.
plan	sailpoint.object.ProvisioningPlan	Provisioning plan containing the provisioning request(s)
request	Sailpoint.object.ProvisioningPlan. AbstractRequest	AccountRequest being processed; always null for this global rule; only set for SapHrOperationProvisioning
connector	Sailpoint.connector.SAPHRConnector	Application connector being used for the operation

Outputs:

Argument	Type	Purpose
result	sailpoint.object.ProvisioningResult	ProvisioningResult object containing the status (success, failure, retry, etc.) of the provisioning request

Example

This example rule shows how to parse a provisioningPlan to determine the appropriate operation. The details within each operation would then be similar to the logic shown in the SapHrOperationProvisioning rule example below.

```
ProvisioningResult result = new ProvisioningResult();
if ( plan != null ) {
      _log.debug( "plan [" + plan.toXml() + "]" );

      List accounts = plan.getAccountRequests();
      if ( ( accounts != null ) && ( accounts.size() > 0 ) ) {
        for ( AccountRequest account : accounts ) {
            try {
                  if ( AccountRequest.Operation.Create.equals(
account.getOperation() ) ) {
              // Process Create request
          } else if ( AccountRequest.Operation.Create.equals( account.getOperation())) {
            // Process Modify request as illustrated in SapHrOperationProvisioning
            // rule below
          …
          }
        }
      }
    }
}
return result;
```

SapHrOperationProvisioning

Description

The SAP HR/HCM connector was introduced in version 7.0, so this rule applies only to versions 7.0+.

An SAP HR Operation Provisioning rule is an alternative to the SAP HR Provision rule when the installation wants to partition their provisioning logic into individual rules per operation. It contains the installation-specific provisioning logic for each specific provisioning operation for the SAP HR application.

Separate SapHrOperationProvisioning rules can be created for account enabling, account disabling, account deletion, account unlocking, account creation, and account modification.

Definition and Storage Location

This rule is associated to an application in the UI through the application definition:

Applications -> Application Definition -> select application or create new application with Application Type: SAP HR/HCM -> Rules -> Provision Rule Type: By Operation Rules -> [Operation] Provision Rule

The reference to the rule is recorded in the Application XML.

```
<entry key="saphrEnableProvisioningRule" value="[OperationProvision Rule Name]"/>
<entry key="saphrDisableProvisioningRule" value="[OperationProvision Rule Name]"/>
<entry key="saphrCreateProvisioningRule" value="[OperationProvision Rule Name]"/>
<entry key="saphrDeleteProvisioningRule" value="[OperationProvision Rule Name]"/>
<entry key="saphrModifyProvisioningRule" value="[OperationProvision Rule Name]"/>
```

```
<entry key="saphrUnlockProvisioningRule" value="[OperationProvision Rule Name]"/>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
application	sailpoint.object.Application	Reference to the application object
schema	sailpoint.object.Schema	Reference to the application schema
destination	com.sap.conn.jco.JCoDestination	A connected and ready to use SAP destination object that can be used to call BAPI function modules and call to SAP tables.
connection	java.sql.Connection	Connection object to connect to the JDBC database
plan	sailpoint.object.ProvisioningPlan	Provisioning plan containing the provisioning request(s) to be processed
request	sailpoint.object.ProvisioningPlan.AbstractRequest	AbstractRequest object containing the account request (or object request, in the case of group provisioning) to be processed

Outputs:

Argument	Type	Purpose
result	sailpoint.object.ProvisioningResult	ProvisioningResult object containing the status (success, failure, retry, etc.) of the provisioning request

Example

An example rule of this type is provided in the [IdentityIQ installation directory]/WEB-INF/config/examplerules.xml file. A simplified version of that rule is provided here for illustration of the key components of this rule. This example is for an account modify operation, and this simplified version only updates the email address attribute. Error handling and other auxiliary logic have been omitted for brevity. The rule in examplerules.xml should be used as a model for writing your actual production rule.

In summary, this rule should iterate through the AccountRequests in the (modify) ProvisioningPlan. For each AccountRequest, if there is a request to set the email address to a new value, it should issue the appropriate BAPI calls to update that email address. Additional logic in this rule illustrates calls to retrieve date range parameter and other attributes which can affect the request.

This rule modifies only the email attribute ("0010"), but the same logic could also be applied to modify phone ("0020") or SY-USERNAME ("0001"), as shown in the example in examplerules.xml.

```
        boolean isEmailInPlan = false;
        String SUBTYPE_EMAIL = "0010";
        String errorText = "";

        // This function will iterate through account requests in plan, retrieve email
```

```
    // address attribute request updates, and process them
public void doProvision() throws Exception {
    List<AccountRequest> accReqList = plan.getAccountRequests();
    String accNativeIdentity = null;
    if (!Util.isEmpty(accReqList)) {
        int accReqListSize = accReqList.size();
        for( AccountRequest accReq : accReqList ) {
            if ( accReq.getApplication().equals( application.getName() ) ) {
                accNativeIdentity = accReq.getNativeIdentity();
                AttributeRequest emailAttrib = accReq.getAttributeRequest("Email");

                //Finding the email attribute in provisioning plan and
                //trying to modify the account's email id
                if ( null != emailAttrib &&
                    emailAttrib.getOp().toString().equalsIgnoreCase("Set") ) {
                    isEmailInPlan = true;
                    modifyCommunicationData(accNativeIdentity,
                        emailAttrib.getValue(), SUBTYPE_EMAIL);
                }
            }
        }
    }
}

    // function modifies the email address of SAP HR record
    // Email must be present(assigned) in order to modify it
    // BAPI used - BAPI_EMPLCOMM_CHANGE
public void modifyCommunicationData( String userId, String parValue, String type)
    throws Exception {
    // set effective date range from today till end of year 9999
     String endDateStr = "99991231";
     SimpleDateFormat formatter = new SimpleDateFormat("yyyyMMdd");
     String beginDateStr = formatter.format(new Date());
     // Bapi locks the record for processing
     JCoFunction functionEnqueue = destination.getRepository().getFunction(
            "BAPI_EMPLOYEE_ENQUEUE");
     functionEnqueue.getImportParameterList().setValue("NUMBER", userId);

     // Bapi to modify Communication data - email and phone
     JCoFunction functionCommunicationChange =
connector.getFunction(destination,"BAPI_EMPLCOMM_CREATE");
     if ( functionCommunicationChange == null )
          throw new RuntimeException("BAPI_EMPLCOMM_CREATE not found in SAP.");
     String returnPersonnelID = null;
     functionCommunicationChange.getImportParameterList().setValue("EMPLOYEENUMBER",
userId); // Personal Number
     functionCommunicationChange.getImportParameterList().setValue("SUBTYPE", type); //
SubType 0010/0020 - Email/Phone
     functionCommunicationChange.getImportParameterList().setValue("VALIDITYBEGIN",
beginDateStr); // Begin Date
     functionCommunicationChange.getImportParameterList().setValue("VALIDITYEND",
endDateStr); // End Date
     functionCommunicationChange.getImportParameterList().setValue("COMMUNICATIONID",
parValue); // Email Address to modify

     // Bapi unlocks the record after processing
     JCoFunction functionDequeue = destination.getRepository().getFunction(
         "BAPI_EMPLOYEE_DEQUEUE");
     functionDequeue.getImportParameterList().setValue("NUMBER", userId);

     try {
          // executing Bapis
          JCoContext.begin(destination);
```

```
            functionEnqueue.execute(destination);
            functionCommunicationChange.execute(destination);
            functionDequeue.execute(destination);
        } catch (Exception e) {

        } finally {
            JCoContext.end(destination);
        }
    }

    doProvision();
    return result;
```

PeopleSoftHRMSProvision

Description

The PeopleSoft HCM Database connector was introduced in version 7.0, so this rule applies only to versions 7.0+.

Two options are available for configuring provisioning to PeopleSoft HCM Database applications:

1. A single rule (of type PeopleSoftHRMSProvision) which contains all supported provisioning operations, or
2. A collection of operation-specific rules (of type PeopleSoftHRMSOperationProvisioning, discussed below), one per supported provisioning operation

A PeopleSoft HRMS Provision rule is specified for an application that uses the PeopleSoft HCM Database connector if it will support provisioning. It contains the installation-specific provisioning logic for provisioning to the PeopleSoft HCM database application.

Definition and Storage Location

This rule is associated to an application in the UI through the application definition:

> Applications -> Application Definition -> select application or create new application with Application Type: PeopleSoft HCM Database -> Rules -> Connector Rules section -> select Provision Rule Type: Global Provision Rule -> Provision Rule

The reference to the rule is recorded in the Application XML.

```
<entry key="peoplesofthrProvisionRule" value="[Provision Rule Name]"/>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
application	sailpoint.object.Application	Reference to the application object
schema	sailpoint.object.Schema	Reference to the application schema
plan	sailpoint.object.ProvisioningPlan	Provisioning plan containing the provisioning request(s)
request	Sailpoint.object.ProvisioningPlan.	AccountRequest being processed; always null

| | AbstractRequest | for this global rule; only set for PeopleSoftHRMSOperationProvisioning rules |
| connector | Sailpoint.connector. PeopleSoftHRMSConnector | Application connector being used for the application |

Outputs:

Argument	Type	Purpose
result	sailpoint.object.ProvisioningResult	ProvisioningResult object containing the status (success, failure, retry, etc.) of the provisioning request

Example

This example rule shows how to parse a provisioningPlan to determine the appropriate operation. The details within each operation would then be similarto the logic shown in the PeopleSoftHRMSOperationProvisioning rule example below.

```
ProvisioningResult result = new ProvisioningResult();
if ( plan != null ) {
      _log.debug( "plan [" + plan.toXml() + "]" );

      List accounts = plan.getAccountRequests();
      if ( ( accounts != null ) && ( accounts.size() > 0 ) ) {
        for ( AccountRequest account : accounts ) {
             try {
                  if ( AccountRequest.Operation.Create.equals(
account.getOperation() ) ) {
             // Process Create request
          } else if ( AccountRequest.Operation.Create.equals( account.getOperation())) {
             // Process Modify request as illustrated in the
             // PeopleSoftHRMSOperationProvisioning rule below
          …
          }
        }
      }
    }
}
return result;
```

PeopleSoftHRMSOperationProvisioning

Description

The PeopleSoft HCM Database connector was introduced in version 7.0, so this rule applies only to versions 7.0+.

A PeopleSoft HRMS Operation Provisioning rule is an alternative to the PeopleSoft HRMS Provision rule when the installation wants to partition their provisioning logic into individual rules per operation. It contains the installation-specific provisioning logic for each specific provisioning operation for the PeopleSoft HCM Database application.

Separate PeopleSoftHRMSOperationProvisioning rules can be created for account enabling, account disabling, account deletion, account unlocking, account creation, and account modification.

Definition and Storage Location

This rule is associated to an application in the UI through the application definition:

> Applications -> Application Definition -> select application or create new application with Application Type: PeopleSoft HCM Database -> Rules -> Connector Rules section -> select Provision Rule Type: By Operation Rules -> [Operation] Provision Rule

The reference to the rule is recorded in the Application XML.

```
<entry key="peoplesofthrEnableProvisioningRule" value="[OperationProvision Rule
Name]"/>
<entry key="peoplesofthrDisableProvisioningRule" value="[OperationProvision Rule
Name]"/>
<entry key="peoplesofthrCreateProvisioningRule" value="[OperationProvision Rule
Name]"/>
<entry key="peoplesofthrDeleteProvisioningRule" value="[OperationProvision Rule
Name]"/>
<entry key="peoplesofthrModifyProvisioningRule" value="[OperationProvision Rule
Name]"/>
<entry key="peoplesofthrUnlockProvisioningRule" value="[OperationProvision Rule
Name]"/>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
application	sailpoint.object.Application	Reference to the application object
session		Session connection to PeopleSoft server
schema	sailpoint.object.Schema	Reference to the application schema
plan	sailpoint.object.ProvisioningPlan	Provisioning plan containing the provisioning request(s) to be processed
request	sailpoint.object.ProvisioningPlan. AbstractRequest	AbstractRequest object containing the account request (or object request, in the case of group provisioning) to be processed
connector	Sailpoint.connector. PeopleSoftHRMSConnector	Application connector being used for the application

Outputs:

Argument	Type	Purpose
result	sailpoint.object.ProvisioningResult	ProvisioningResult object containing the status (success, failure, retry, etc.) of the provisioning request

Example

An example rule of this type is provided in the [IdentityIQ installation directory]/WEB-INF/config/examplerules.xml file. A simplified version of that rule is provided here for illustration of the key components of this rule. This example is for an account modify operation, and this simplified version only updates the email address attribute. Error handling, import statements, and other auxiliary logic have been omitted for brevity. The rule in examplerules.xml should be used as a model for writing your actual production rule.

In summary, this rule iterates through the AccountRequests in the (modify) ProvisioningPlan. For each AccountRequest, if there is a request to set the email address to a new value, it calls the appropriate method in the connector to update the email attribute.

```
//Variables to read Plan operation
AccountRequest req = null;
Operation operation = null;

ProvisioningResult result = new ProvisioningResult();
List<AccountRequest> accountRequests = plan.getAccountRequests();
int size = accountRequests.size();
ComponentInterface ci = null;

 //This method is used for initializing Component Interface
 public boolean initCI(String accNativeIdentity) {
    //provide CI name to get handle of the CI
    //In this example 'CI_PERSONAL_DATA' is component interface provided out of box
by PeopleSoft HRMS
    //which is used for updating personal data.
     ci = connector.getCIHandle("CI_PERSONAL_DATA");

   //first set the property 'KEYPROP_EMPLID' with the corresponding EMPLID
     ci.setPropertyByName("KEYPROP_EMPLID", accNativeIdentity);
     boolean userExists = false;

   // Get the employee record
    if (null != ci) {
       userExists = ci.get();
    }
    return userExists;
 }

 //This function will modify the email address if received in the plan
 public void doProvision() {
    HashMap emailObj = new HashMap();
    List<AccountRequest> accReqList = plan.getAccountRequests();
    String accNativeIdentity = null;
    String emailCollAttribute = null;

    if (!Util.isEmpty(accReqList)) {
        int accReqListSize = accReqList.size();
        for( AccountRequest accReq : accReqList ) {
            if ( accReq.getApplication().equals( application.getName() ) ) {
                accNativeIdentity = accReq.getNativeIdentity();
                //Get the requests by passing schema attributes
                AttributeRequest emailAttribReq =
accReq.getAttributeRequest("EMAIL_ADDR");

                try {
                 boolean userExists = initCI(accNativeIdentity);
```

```
                   if(userExists) {
                        if ( null != emailAttribReq) {
                            Object emailValue = emailAttribReq.getValue();
                            String attrName = emailAttribReq.getName();
                            if(null != emailValue){

updateEmail(emailCollAttribute,emailObj,ci,emailValue,emailAttribReq);
                            }
                            //Reset the component interface. This is required between some
operations
                            //to make sure old data is not in the component interface.
                            connector.resetCI();
                            isReset = true;
                         }
                    } else {
                          ProvisioningResult result = new ProvisioningResult();
                          result.setStatus(ProvisioningResult.STATUS_FAILED);
                          result.addError("User does not exist " + accNativeIdentity );
                     }
                 } catch (Exception e) {
                   result.setStatus(ProvisioningResult.STATUS_FAILED);
                   result.addError(e.getMessage());
                 }
             }
          }
       }
    }

    // This function will update the email address
     /* 'COLL_EMAIL_ADDRESSES' is a collection attribute (of 3 sub attributes) needed to
support tracking multiple
        email addresses per person. Sub attributes:
PROP_EMAIL_ADDR,KEYPROP_E_ADDR_TYPE,PROP_PREF_EMAIL_FLAG
     */
    public void updateEmail(String emailCollection,HashMap emailObj,ComponentInterface
ci, Object emailValue, AttributeRequest req) {

            String type = "KEYPROP_E_ADDR_TYPE";
            boolean isUpdated = false;
            ProvisioningResult provisioningResult = new ProvisioningResult();
            emailCollection = "COLL_EMAIL_ADDRESSES";
            emailObj.put("PROP_EMAIL_ADDR", emailValue);  // email address
            emailObj.put("KEYPROP_E_ADDR_TYPE", "BUSN");  // address type = business
            emailObj.put("PROP_PREF_EMAIL_FLAG", "Y");     // marked as primary
            //calling updateCollectionAttributes method of connector
            try {
               isUpdated = connector.updateCollectionAttributes(emailCollection,
emailObj, ci, type );
            } catch(Exception e) {
                provisioningResult.setStatus(ProvisioningResult.STATUS_FAILED);
                provisioningResult.addError(e.getMessage());
                req.setResult(provisioningResult);
            }
            if(isUpdated) {
                provisioningResult.setStatus( ProvisioningResult.STATUS_COMMITTED );
            }
    }

     // Logic to Read operation from Plan
    for ( int i=0;i < size;i++ ) {
         req=accountRequests.get(i);
         operation=req.getOperation();
```

```
            if( operation.toString().equals("Modify") ) {
                // call doProvision method
                doProvision();
            }
        }
    return result;
```

Integration

Description

An Integration rule is the rule type for a plan initializer rule, which contains custom logic that is executed immediately before the provisioning plan is sent to a writeable connector or PIM/SIM to be executed.

This rule can be used to economize what data gets passed across to the integration or connector, instead of sending lots of unneeded data (e.g. - loading just the name of the person being remediated or of the requester, instead of passing the entire Identity object to the integration).

Definition and Storage Location

There is no UI option for specifying an Integration rule. It can only be specified through the XML of an IntegrationConfig or within the ProvisioningConfig in an Application definition. It is referenced within a <PlanInitializer> element.

```
<PlanInitializer>
    <Reference class="sailpoint.object.Rule" name="[Integration Rule Name]"/>
</PlanInitializer>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
identity	sailpoint.object.Identity	Reference to the Identity object for which the provisioning request has been made
integration	sailpoint.object.Integration Config	Reference to the integrationConfig (or ProvisioningConfig cast as an integrationConfig) that defines provisioning to the application
plan	sailpoint.object.Provisionin gPlan	Reference to the ProvisioningPlan object containing the requested provisioning action

Outputs:

Argument	Type	Purpose
result	sailpoint.object.Provision ingResult	Result indicating success or failure; failure halts the provisioning action Any other return type (including no return value) allows provisioning processing to continue

Example

This example Integration rule retrieves the requester from the plan and loads just the name into the plan arguments map. The integration executor or connector is eventually given a simplified version of the provisioningPlan object; this simpler form does not contain a Requester list, so that information must be passed through the arguments map if it is needed for the final provisioning action.

```
import java.util.ArrayList;
import java.util.List;
import sailpoint.object.Attributes;
import sailpoint.object.Identity;
import sailpoint.object.ProvisioningPlan;

/**
 * Get plan arguments into a map
 */
Map map = (Map) plan.getIntegrationData();

/* Retrieve an Identity from the plan's Requesters list and save
 * the Identity's name into the arguments map (usually only one in list)*/

String name = null;
if ( plan.getRequesters() != null ) {
   for ( Identity requester : plan.getRequesters() ) {
       name = requester.getName();
   }
   map.put("requester", name );
}
```

Notification/Assignment Rules

These rules are used in determining the recipient Identity for email notifications, escalations, approvals, etc. These apply to different types of system objects, as noted in each rule description.

EmailRecipient

Description

An EmailRecipient Rule is used to specify additional email recipients for certification reminder notifications, escalation notifications, and escalation reminder notifications.

Definition and Storage Location

This rule is associated to a certification in the UI through the Certification Definition.

> Setup -> Certifications -> Create new certification of any type -> Notifications -> Notify before Certification Expires -> Add a Reminder or Add Escalation -> Additional Email Recipients -> Recipient(s) Rule

The rule name is recorded in a NotificationConfig within the CertificationDefinition XML. Email Recipient Rules are recorded as the additionalRecipientsRuleName in a ReminderConfig for certification reminders and in an EscalationConfig for certification escalations.

```
<entry key="certification.remindersAndEscalations">
```

```
 <value>
  <NotificationConfig enabled="true" escalationEnabled="true" remindersEnabled="true">
    <Configs>
<ReminderConfig additionalRecipientsPresent="true"
additionalRecipientsRuleName="[Email Recipient Rule Name]" … />
<EscalationConfig additionalRecipientsPresent="true"
additionalRecipientsRuleName="[Email Recipient Rule Name]" … />
    </Configs>
  </NotificationConfig>
 </value>
</entry>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
item	sailpoint.object.Notifiable	The Notifiable interface for objects that can be reminded, escalated, and expired

Outputs:

Argument	Type	Purpose
identity	String, List (of strings)	The identity name or names to whom the email should be sent

Example

This example EmailRecipient rule returns the name of the item owner's manager as the email recipient.

```
import sailpoint.object.Identity;

Identity manager = new Identity();

Identity owner = item.getOwner();

if (null != owner) {
    manager = owner.getManager();
    if (null != manager)
        return manager.getName();
}
```

Escalation

Description

An Escalation rule identifies a new owner for a workItem or certification when an "escalation" of the item is triggered. For a certification, this occurs when the certification has not been signed off and a triggering time period or number of reminder notices is reached. For a workItem, this can be when an inactive owner is detected or according to the notification schedule specified in the workItem's notificationConfig.

NOTE: The notification configuration mechanism for certifications was updated in version 6.0 to allow more flexibility in reminder notifications and escalations. Beginning with 6.0, each certification escalation only runs once at the prescribed date and time, so the rule only needs to return one escalation recipient. Subsequent escalations can be configured to return a different recipient using the same or a different rule. This can simplify the logic required in any given certification escalation rule, since a single escalation rule is no longer required to manage escalation through a chain of people.

Definition and Storage Location

Escalation rules can be associated to workItems and certifications in a few places in the UI. For certifications, this is set in the Certification specification:

> Setup -> Certifications -> create new certification (any type) -> Notifications -> Notify Before Certification Expires -> Add Escalation -> Escalation Rule

An escalation rule can also be associated with certification revocations in the certification specification:

> Setup -> Certifications -> create new certification (any type) -> Notifications -> Escalate Revocations -> Escalation Rule

The references to the escalation rules for a certification are recorded in the CertificationDefinition XML in an EscalationConfig (within a NotificationConfig).

```
<entry key="certification.remindersAndEscalations">
  <value>
    <NotificationConfig enabled="true" escalationEnabled="true">
      <Configs>
<EscalationConfig before="true" emailTemplateName="Work Item Escalation"
enabled="true" escalationRuleName="[Escalation Rule Name]" millis="604800000"/>
      </Configs>
    </NotificationConfig>
  </value>
</entry>
<entry key="remediation.remindersAndEscalations">
  <value>
<NotificationConfig enabled="true" escalationEnabled="true"
escalationMaxReminders="5">
      <Configs>
<EscalationConfig before="true" emailTemplateName="Work Item Escalation"
enabled="true" escalationRuleName="[Escalation Rule Name]" maxReminders="5"
millis="604800000"/>
      </Configs>
    </NotificationConfig>
  </value>
</entry>
```

The inactive owner workItem escalation rule is a system configuration option:

> Gear menu -> Global Settings -> IdentityIQ Configuration -> WorkItem -> WorkItem Rules section -> Inactive user work item escalation rule

That escalation rule name is recorded in the System Configuration XML.

```
<entry key="inactiveOwnerWorkItemForwadRule" value="escalate to spadmin"/>
```

There are many types of workItems that may be created in IdentityIQ – policy violations, certification escalations, workflow approvals or provisioning forms, etc. Any of these workItems can contain notificationConfigs that include an escalationConfig as shown on the CertificationDefinition above.

```
<NotificationConfig enabled="true" escalationEnabled="true"
escalationMaxReminders="5">
   <Configs>
<EscalationConfig before="true" emailTemplateName="Work Item Escalation"
enabled="true" escalationRuleName="[Escalation Rule Name]" maxReminders="5"
millis="604800000"/>
   </Configs>
</NotificationConfig>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
item	sailpoint.object.Notifiable	The Notifiable interface for the object (work item or certification) being escalated

Outputs:

Argument	Type	Purpose
newOwner	String	The identity name to whom the item is being assigned in escalation

Examples

This example Escalation rule escalates the item to the current owner's manager. If that manager is an inactive Identity, it continues up the manager chain until it finds an active Identity. If no new owner can be found (or if the current owner value is null), it escalates to a default Identity – in this case, the Administrator.

```
import sailpoint.object.Identity;

// method returns owner of item (workItem)
Identity owner = item.getNotificationOwner(context);

// if no owner, escalate to spadmin
if (owner == null)
    return "spadmin";
else {
    // escalate to owner's manager; if manager is inactive, keep
    // escalating until find active manager
    Identity newOwner = owner.getManager();
    while (newOwner != null && newOwner.isInactive()) {
        newOwner = newOwner.getManager();
    }
}

if (newOwner == null)
    return "spadmin";
```

```
    else
        return newOwner.getName();
```

This example Escalation rule is intended for an inactive workItem owner rule; it assumes the current owner is inactive, since the rule would only be called in that case, and it selects managers up the corporate hierarchy until it finds an active manager. If no new owner can be found, it escalates to a default Identity – in this case, the Administrator.

```
    import sailpoint.object.Identity;

    Identity newOwner = item.getNotificationOwner(context);

    while (newOwner != null &amp;&amp; newOwner.isInactive()) {
        newOwner = newOwner.getManager();
    }

    if (newOwner == null)
        return "spadmin";
    else
        return newOwner.getName();
```

Approver

Description

An Approver rule once was called when a role or profile change was submitted for approval from the modeler or when a candidate role was submitted for approval from a certification or role mining action. This rule has been ignored by recent versions of IdentityIQ, having been replaced by the **Role create, update, and delete** business process, but was inadvertently left in the System Configuration UI pages (**Gear menu -> Global Settings -> IdentityIQ Configuration** -> **Roles** -> **Rules** section -> **Role and profile change approver rule**) until version 6.3. It should not be used, and will not be run even if specified, in any version of IdentityIQ covered by this document.

ApprovalAssignment

Description

The ApprovalAssignment rule type was introduced in version 6.2. It is called during the approval generation process in a workflow – specifically the Provisioning Approval Subprocess that ships with IdentityIQ versions 6.2+. It is passed the approval list as it has been built based on the the approval step specification, but it provides one last hook where custom logic can be infused into the approval creation process. It could, for example, change who is responsible for completing the approval process based on some attribute about the request, the workItem, or the target Identity.

The main purpose of this rule is to allow approval ownership to be calculated based on extended attribute or some other criteria that falls outside the scope of the default mechanisms for deriving the approval owner. It could also be used to alter the approval scheme according to non-standard criteria, or even to bypass approval entirely based on certain criteria.

Definition and Storage Location

Typically, this rule is specified as an argument to the Approve step of the LCM Provisioning workflow, which invokes the Provisioning Approval Subprocess workflow, passing the rule name to it. It is run by the buildCommonApprovals workflow library method, so it can be set in any workflow which invokes that method to build the approval object.

The ApprovalAssignment Rule is specified as an argument to the workflow approval step which launches the Provisioning Approval Subprocess workflow or as an argument to the workflow step which invokes the buildCommonApprovals library method directly.

```
<arg name="approvalAssignmentRule" value="[Rule Name]"/>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
approvals	List of sailpoint.object.Workflow.Approval objects	List of approval objects as created based on the approval step specification in the workflow; contains the definition and the current state of the approval
approvalSet	sailpoint.object.ApprovalSet	Contains all the items to be approved in the set

NOTE: This rule is also passed the entire list of args provided to the approval step of the workflow, each named according to its name in the args map for the approval step.

Outputs:

Argument	Type	Purpose
approvals	List of sailpoint.object.Workflow.Approval objects	The final approval list to use for this approval proess

Example

This example rule retrieves the target Identity and redirects the approval ownership to a workgroup (called "Security Team") for all users whose location is "Zurich". (IdentityName is an argument passed to the approval step in the default LCM approval process. Any identity attribute could be chosen as a differentiating attribute which should cause the approval ownership to be modified.)

```
import sailpoint.object.Workflow;
import sailpoint.object.Workflow.Approval;
import sailpoint.object.Identity;

Identity targetUser = context.getObjectByName(Identity.class, identityName);

if ("Zurich".equals(targetUser.getAttribute("location"))) {
    List newApprovals = null;
    if ( approvals != null ) {
```

```
            newApprovals = new ArrayList();
            for ( Approval approval : approvals ) {
               if ( approval != null ) {
                   // update the approver/owner to the Security Team
                   approval.setOwner("Security Team");
                   newApprovals.add(approval);
               }
            }
         }
         return newApprovals;
      } else
         return approvals;
```

FallbackWorkItemForward

Description

The FallbackWorkItemForward rule is used to select a fallback owner for a certification work item to prevent self-certification. This runs during certification creation when a predelegation rule in a certification is attempting to assign an item to an owner that will result in self-certification, as well as any time an existing certification work item is forwarded to a different user through automated forwarding (e.g. to the user configured as Forwarding User on the User Preferences page or through execution of the inactive user work item escalation rule or global work item forwarding rule). Of course, this does not apply for users who have been allowed to self-certify (per the allowSelfCertification option in the system configuration).

NOTE: To allow self-certification, you can choose the configuration option to allow self certification for all certifiers (or for one of the supported subsets of certifiers). This can be set through the UI here:

Gear menu -> Compliance Manager -> Behavior -> Allow Self Certifications for: All Certifiers / System and Certification Administrators / System Administrators only

Definition and Storage Location

The default fallback forwarding rule is set in the system configuration through the UI's System Setup menu.

Gear menu -> Global Settings -> IdentityIQ Configuration -> WorkItems -> Self-Certification Work Item Forwarding Rule

The reference to the rule is recorded in the System Configuration XML.

```
<entry key="fallbackWorkItemForwardRule" value="[FallbackWorkItemForward Rule Name]"/>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
item	sailpoint.object.WorkItem	Reference to the workItem (some workItem arguments may not yet be set)
owner	sailpoint.object.Identity	Reference to the Identity who currently owns the work item
creator	String	Name of Identity who created the certification belonging

		to this workItem
certifiers	java.util.List	List of certifier names for the certification belonging to the workItem
name	String	Name of the certification belonging to the workItem (may be null if not created yet)
type	sailpoint.object.Certificatio n.Type enumeration	Type of the certification belonging to the workItem

Outputs:

Argument	Type	Purpose
newOwner	String or sailpoint.object.Identity	Identity object or name of Identity object who should be the new owner of theworkItem

Example

This example FallbackWorkItemForward rule first tries to forward the item to the certification owner. If the certification does not yet exist so its owner cannot be determined, it iterates through the certifiers list and sends the item to the first certifier who is not the current workItem owner. If none of these successfully identifies a certifier, it sends the workItem to the Administrator.

```
import sailpoint.object.Certification;
import sailpoint.object.Identity;

string approver = null;
if (null != name) {
    Certification cert = getObjectByName(Certification.class, name);
    approver = cert.getOwner();
}
if (null == approver) {
for ( string certifier : certifiers ) {
    if (certifier != owner.getName()
        return certifier;
}

return "spadmin";
```

WorkItemForward

Description

A WorkItemForward rule examines a WorkItem and determines whether or not it needs to be forwarded to a new owner for further analysis or action. Only one WorkItemForward rule can be in use at any time for an installation; it is selected in the system configuration and is called every time a WorkItem is opened and any time it is forwarded through the user interface.

Definition and Storage Location

The WorkItemForward rule for the installation is set through the UI in the System Setup options.

Gear menu -> Global Settings -> IdentityIQ Configuration -> WorkItems -> Global WorkItem Forwarding Rule

The rule name is recorded as the value for the workItemForwardRule key in the System Configuration XML.

```
<entry key="workItemForwardRule" value="forward to Spadmin"/>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
item	sailpoint.object.WorkItem	Reference to the workItem being opened (some workItem arguments may not yet be set)
owner	sailpoint.object.Identity	Reference to the Identity who currently owns the work item
identity	sailpoint.object.Identity	Reference to the same Identity object as owner (provided for backward compatibility to older versions of this rule)

Outputs:

Argument	Type	Purpose
newOwner	String or sailpoint.object.Identity	Identity object or name of Identity object who should receive the workItem

Example

This example WorkItemForward rule attempts to find an Identity with an email address by examining the owner first and then checking up the manager chain for an Identity with an email address. The first Identity in the hierarchy found to have an email address is assigned as the workItem owner. If none is found with an email address, the original owner is left as the workItem owner.

```
import sailpoint.object.Identity;

Identity newOwner = owner;

String email = owner.getEmail();
if ( email == null || email.length() == 0 ) {
    newOwner = owner.getManager();
    while ( newOwner != null ) {
        email = newOwner.getEmail();
        if ( email != null && email.length() > 0 )
            break;
        newOwner = newOwner.getManager();
    }
}

if ( email == null || email.length() == 0 ) {
    // This defaults to not changing the owner,
    // but it could alternatively assign it to a fixed user.
    newOwner = owner;
    log.warn("no owner with email found");
```

```
        }

        return newOwner;
```

Owner Rules

Owner rules assign ownership of certain system objects to a given Identity. Their usage locations are included in the Description section of each rule type.

Owner

See *Owner* in Form/Provisioning Policy-related Rules section.

Policy Owner

See *PolicyOwner* in Policy/Violation Rules section.

GroupOwner

Description

The GroupOwner rule is used to assign group owners for the groups created from a GroupFactory.

Definition and Storage Location

The GroupOwner rule is set for a GroupFactory through the UI on the Groups page.

> Setup -> Groups -> select or create a Group -> Group Owner Rule

The reference to the rule is recorded in the GroupFactory XML.

```
<GroupOwnerRule>
<Reference class="sailpoint.object.Rule" id="402846023a65e596013a65e6e8020274"
name="Group Ownership Rule – Highest Ranking Member of Sub-Group"/>
</GroupOwnerRule>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
factory	sailpoint.object.GroupFactory	Reference to the groupFactory object from which the groups are generated
group	sailpoint.object.GroupDefinition	Reference to a single GroupDefinition from the factory

Outputs:

Argument	Type	Purpose
owner	sailpoint.object.Identity	Identity object or name of the Identity assigned as the

or string	group owner

Example

This example GroupOwner rule assigns group ownership to the employee in the group with the lowest employee ID (the employee with the most seniority at the company).

```
import sailpoint.object.QueryOptions;
import sailpoint.object.Identity;

QueryOptions qo = new QueryOptions();
// Group defined as a filter so add
// filter to queryOptions to get members list
qo.addFilter(group.getFilter());
Iterator identities = context.search(Identity.class, qo);

//Find the employee with the lowest employee ID.
Identity emp = null;
String empId = null;
Identity owner = null;
String ownerEmpId = null;

while (identities.hasNext()) {
  emp = identities.next();
  empId = emp.getAttribute("empId");

  if (empId != null && (ownerEmpId == null ||
      empId.compareTo(ownerEmpId) < 0)) {
    owner = emp;
    ownerEmpId = empId;
  }
}

//When all of the employee IDs in the subgroup are null, default to spadmin.
if (owner == null) {
  return "spadmin";
}

return owner;
```

Scoping Rules

Scoping rules are used to assign scopes to Identities when scoping is enabled. An Identity's assigned scope determines whether other users can see and make requests for that Identity. Controlled, or authorized, scopes determine what objects each Identity can see and make requests around; controlled scopes are *not* determined by the scoping rules.

ScopeCorrelation

Description

The ScopeCorrelation rule evaluates one or more Identity attributes to select a scope or list of scopes that applies to the Identity. If it returns multiple scopes, the ScopeSelection rule chooses which of the scopes to assign. There is only one ScopeCorrelation rule per IdentityIQ installation.

Definition and Storage Location

The ScopeCorrelation rule is set through the UI on the Configure Scoping page.

> Gear menu -> Global Settings -> Scopes -> Configure Scoping -> Scope Correlation Rule

The reference to the rule is recorded in the Identity ObjectConfig XML.

```
<entry key="scopeCorrelationRule" value="[Scope Correlation Rule Name]"/>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
identity	sailpoint.object.Identity	Reference to the identity being assigned a scope
scopeCorrelationAttribute	String	Name of the scope correlation attribute specified in the scoping configuration
scopeCorrelationAtributeValue	String	The value for the correlation attribute on the Identity

Outputs:

Argument	Type	Purpose
scopes	sailpoint.object.Scope or java.util.List<Scope>	One or more scopes that meet the rule's criteria for assignment to the Identity

Example

This example ScopeCorrelation rule assigns Identities with a job title of "Administrator" to the "All" scope, which is the highest level scope in this company's scope hierarchy. Otherwise, it assigns the Identity to the scope whose name corresponds to the Identity's "region" attribute, creating a new scope if no matching scope exists.

```
import sailpoint.object.Scope;

Scope all = context.getObjectByName(Scope.class, "All");
// if scope "All" doesn't exist yet, create it
if (all == null) {
    all = new Scope("All");
    context.saveObject(all);
    String allId = all.getId();
    all.setDisplayName("All");
    all.setPath(allId);
    all.setAssignedScope(all);

    context.saveObject(all);
    context.commitTransaction();
}

String jobTitle = identity.getStringAttribute("jobTitle");
// Assign scope "all" to any Identity with the jobTitle of "Administrator"
if ("Administrator".equals(jobTitle)) {
```

```
      return all;
}
// Since the user's scope isn't "all", get region and check if it exists as scope
String region = identity.getStringAttribute("region");
if (region == null) {
return null;
}

try {
    Scope scope = context.getObjectByName(Scope.class, region);

    if (scope == null) {
      // If it doesn't exist, then we need to create it as a child of the All scope
      scope = new Scope(region);
      context.saveObject(scope);
      all.addScope(scope);
      scope.setDisplayName(region);
      scope.setAssignedScope(scope);
      context.saveObject(scope);
      context.saveObject(all);
      context.commitTransaction();
    }

    return scope;

} catch (GeneralException e) {
    log.error("Error creating scope.", e);
    return null;
}
```

ScopeSelection

Description

The ScopeSelection rule runs to select a single scope to assign to an Identity when the scope attribute correlation or scopeCorrelation rule have identified multiple possible scopes for the Identity. There is only one scopeSelection rule per IdentityIQ installation.

Definition and Storage Location

The ScopeSelection rule is set through the UI on the Configure Scoping page.

> Gear menu -> Global Settings -> Scopes -> Configure Scoping -> Scope Selection Rule

The reference to the rule is recorded in the Identity ObjectConfig XML.

> <entry key="scopeSelectionRule" value="[Scope Selection Rule Name]"/>

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
identity	sailpoint.object.Identity	Reference to the identity being assigned a scope
scopeCorrelation Attribute	String	Name of the scope correlation attribute specified in the scoping configuration

scopeCorrelation AtributeValue	String	The value for the correlation attribute on the Identity
candidateScopes	java.util.List <sailpoint.object.Scope>	List of scopes identified as candidates for assignment to the Identity; rule should select one of these and return it as the scope to assign

Outputs:

Argument	Type	Purpose
scope	sailpoint.object.Scope	Scope to be assigned to the Identity

Example

This example ScopeSelection rule looks at the scope hierarchy above the identified candidate scopes to find a scope that matches the identity's "department" attribute. This rule could be used for an installation where the scope hierarchy is based on a combination of department and location, as illustrated here:

- ¬ Accounting
    - ¬ Dallas
    - ¬ New York
- ¬ Marketing
    - ¬ Los Angeles
    - ¬ New York

If the scope correlation attribute is location, an Identity who works in New York would have both New York scopes identified as candidate scopes. This ScopeSelection rule would choose the first one if he were in the Accounting department in New York.

```
import sailpoint.object.Scope;

Scope selected = null;

// Use the identity's department to select the correct subscope.
String dept = identity.getAttribute("department");
if (null != dept) {
    for (Iterator it=candidateScopes.iterator(); it.hasNext(); ) {
        Scope current = (Scope) it.next();

        // If any of the ancestor scopes have this user's department
        // name, then use it.
        Scope parent = null;
        while (null != (parent = current.getParent())) {
            if (dept.equals(parent.getName())) {
                selected = current;
                break;
            }
        }
    }
}

return selected;
```

Identity and Account Mapping Rules

There are three rules that can be set in the Identity Mapping windows:

- IdentityAttribute for specifying the identity attribute source when it is not mapped from a single application attribute
- IdentityAttributeTarget for specifying transformations on attributes being pushed to targets
- Listener for responding to value changes on an attribute

A LinkAttribute rule can specify the source mapping for link attributes on the Account Mapping window.

IdentityAttribute

Description

When identity attribute mapping depends on multiple application attributes or other complex evaluations, an IdentityAttribute rule can be specified to control that mapping. IdentityAttribute rules can be specified as application-specific or global rules.

Definition and Storage Location

An IdentityAttribute rule is connected to the Identity ObjectConfig in the UI through the Identity Mapping Sources.

> Gear menu -> Global Settings -> Identity Mappings -> Add New Attribute (or edit existing attribute) -> Add Source (or click an existing source to edit it) -> Application Rule or Global Rule -> Rule

A reference to the rule gets stored in the Identity ObjectConfig XML within the AttributeSource element; if the rule is application-specific, an ApplicationRef is also recorded within the AttributeSource.

```
<AttributeSource name="[System-assigned name for mapping source]">
   <!—ApplicationRef only here if rule is app-specific, not global -->
   <ApplicationRef>
<Reference class="sailpoint.object.Application" id="402846023a65e596013a65e7acaa0506"
name="[application name]"/>
   </ApplicationRef>
   <RuleRef>
<Reference class="sailpoint.object.Rule" id="402846023a65e596013a65e7838704d3"
name="[IdentityAttribute Rule Name]"/>
   </RuleRef>
</AttributeSource>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
environment	java.util.Map	Map of arguments to the aggregation or refresh task that is executing the rule in attribute promotion
identity	sailpoint.object.Identity	Reference to identity object that represents the user being aggregated/refreshed

attributeDefinition	sailpoint.object.Attribute Definition	Reference to the attributeDefinition object for this attribute
link	sailpoint.object.Link	Only included as an argument for application rules, not global rules
attributeSource	sailpoint.object.Attribute Source	Attribute source definition (see AttributeSource object XML above for an example)
oldValue	java.lang.Object	Attribute value of target identity attribute before the rule runs

Outputs:

Argument	Type	Purpose
attributeValue	java.lang.Object	Value to record for the attribute

Example

This example IdentityAttribute rule examines the "userCode" link attribute from the application schema and sets the isContractor Identity attribute (custom attribute) to true when the userCode is 4300.

```
import sailpoint.object.Link;
import sailpoint.object.Attributes;

String isContractor = "false";
Attributes attrs = link.getAttributes();
if ( attrs != null ) {
    int userCode = attrs.getInt("userCode");
    if ( userCode == 4300 ) {
        isContractor = "true";
    }
}
return isContractor;
```

IdentityAttributeTarget

Description

Identity mapping targets are defined when attribute changes are to be propagated to accounts on other applications. If any manipulation or transformation is required on the attribute value before it can be written to the target application, an IdentityAttributeTarget rule is used to perform that action.

Definition and Storage Location

An IdentityAttributeTarget rule is connected to the Identity ObjectConfig in the UI through the Identity Mapping Targets.

> Gear menu -> Global Settings -> Identity Mappings -> Add New Attribute (or edit existing attribute) -> Add Target (or click an existing Target to edit it) -> Transformation Rule

A reference to the rule gets stored in the Identity ObjectConfig XML within an AttributeTarget element; since targets are always application-specific, an ApplicationRef is also recorded within the AttributeTarget.

```
<AttributeTarget name="status">
  <ApplicationRef>
    <Reference class="sailpoint.object.Application"
id="402846023a65e596013a65e7b2e4050b" name="XYZ Application"/>
  </ApplicationRef>
  <RuleRef>
    <Reference class="sailpoint.object.Rule" id="402846023ab1fc5e013ab3bccce40197"
name="[IdentityAttributeTarget Rule Name]"/>
  </RuleRef>
</AttributeTarget>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
value	java.lang.Object	Value of the Identity attribute (can be single value or list)
sourceIdentityAttribute	sailpoint.object.objectAttribute	Reference to the source objectAttribute for this target
sourceIdentityAttributeName	string	Name of the identity attribute for this target
sourceAttributeRequest	sailpoint.object.ProvisioningPlan.AttributeRequest	Reference to the ProvisioningPlan AttributeRequest that is setting the attribute on the identity
target	sailpoint.object.AttributeTarget	Reference to the AttributeTarget that is being processed
identity	sailpoint.object.Identity	Reference to the Identity being processed
project	sailpoint.object.ProvisioningProject	Reference to the ProvisioningProject that contains the changes being requested

Outputs:

Argument	Type	Purpose
attributeValue	java.lang.object	Transformed value that will be pushed to the target

Example

This example IdentityAttributeTarget rule transforms a Boolean inactive flag to a string value "inactive" for the application attribute. This can be important when different applications record related values in different formats.

```
import sailpoint.tools.Util;

if (Util.otob(value) == true)
    return "inactive";
else
    return "active";
```

Listener

Description

A Listener rule is triggered when the value of an attribute changes and performs logic in response to that value change.   The rule is called during aggregation or refresh when the attribute value changes.

Definition and Storage Location

A Listener rule is connected to the Identity ObjectConfig in the UI through the Identity Mapping Sources.

> Gear menu -> Global Settings -> Identity Mappings -> Add New Attribute (or edit existing attribute) -> Value Change Rule

A reference to the rule gets stored on the ObjectAttribute in the Identity ObjectConfig XML.

```
<ListenerRule>
      <Reference class="sailpoint.object.Rule" id="402846023a65e596013a65e7853c04d5"
name="Example Change Notification Rule"/>
</ListenerRule>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
environment	java.util.Map	Task arguments for the task that invoked the rule
identity	sailpoint.object.Identity	Reference to the Identity to whom the attribute applies
object	sailpoint.object.Identity	Reference to the Identity to whom the attribute applies; passed in both variables for compatibility with generic rules
attributeDefinition	sailpoint.object.objectAttribute	Definition of the ObjectAttribute
attributeName	String	Name of the ObjectAttribute
oldValue	java.lang.Object	Original (pre-change) value of the attribute
newValue	java.lang.Object	New (post-change) value of the attribute

Outputs: None; the rule performs actions that are outside of the attribute modification process so IdentityIQ does not expect or act upon a return value from this rule.

Example

This example Listener rule sends an email to the Identity's manager if the Identity's UserType attribute changes.

```
import sailpoint.object.Identity;
import sailpoint.object.Certification;
import sailpoint.object.EmailOptions;
import sailpoint.object.EmailTemplate;
import sailpoint.object.Configuration;
import sailpoint.api.ObjectUtil;
import java.util.List;
```

```
import java.util.ArrayList;
import java.util.Map;
import java.util.HashMap;


// Send a mail to the manager
Identity manager = identity.getManager();
if (manager != null) {
      try {
              HashMap args = new HashMap(identity.getAttributes());
              args.put("attributeName", attributeName);
              args.put("oldValue", oldValue);
              args.put("newValue", newValue);

      EmailTemplate template = context.getObjectByName(EmailTemplate.class, "Value
Change Notification");
      List emailRecipientAddresses = ObjectUtil.getEffectiveEmails(context, manager);
      EmailOptions ops = new EmailOptions(emailRecipientAddresses, args);
      context.sendEmailNotification(template, ops);

      } catch( Exception e ) {
              log.error( "Error occurred trying to send an email to the manager."
       + e.getMessage() );
      }
} else {
      log.warn("UserType ValueChange Rule: "
                      + "Identity " + identity.getName() + " has no manager");
}
```

LinkAttribute

Description

A LinkAttribute rule can be used as the source for an Account Mapping activity, promoting account attributes from Links during aggregation. LinkAttribute rules can be specified as application-specific rules or as a global rule.

Definition and Storage Location

A LinkAttribute rule is connected to the Link ObjectConfig in the UI through the Account Mapping Sources.

Gear menu -> Global Settings -> Account Mappings -> Add New Attribute (or edit existing attribute) -> Add Source (or click an existing source to edit it) -> Application Rule or Global Rule -> Rule

A reference to the rule gets stored in the Link ObjectConfig XML within the AttributeSource element; if the rule is application-specific, an ApplicationRef is also recorded within the AttributeSource.

```
<AttributeSource name="[System-assigned name for source]">
   <!-- ApplicationRef only here if rule is app-specific, not global -->
   <ApplicationRef>
<Reference class="sailpoint.object.Application" id="402846023a65e596013a65e7acaa0506"
name="[application name]"/>
   </ApplicationRef>
   <RuleRef>
<Reference class="sailpoint.object.Rule" id="402846023a65e596013a65e7838704d3"
name="[LinkAttribute Rule Name]"/>
   </RuleRef>
</AttributeSource>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
environment	java.util.Map	Map of the task arguments from the aggregation task
link	sailpoint.object.Link	Reference to the link object from which the account attribute value is being extracted/manipulated

Outputs:

Argument	Type	Purpose
value	java.lang.object	Contains the value for the account attribute

Example

This example LinkAttribute rule transforms a string date read from the Link to store it in an account attribute of type Date.

```
import sailpoint.object.Identity;
import sailpoint.tools.Util;

String acctValue = link.getAttribute("acct_lastLogin");

return Util.stringToDate(acctValue);
```

Form/Provisioning Policy-related Rules

These rules are used to set various fields or options on Forms (workflow, reporting) or Templates (provisioning policies). The rule types that relate to these objects are:

- AllowedValues: determines values displayed in drop-down list boxes on workflow forms, provisioning policies, and report forms
- FieldValue: rule for determining field value; on provisioning policy Templates, the calculated value for the field is used *instead* of presenting the field to a user for data input, whereas on Forms, this populates a default value for the field but does not prevent field presentation to a user
- Validation: rule for validating the contents of a field on a Template or Form; runs on submission of the form and prevents data from being submitted if the field fails validation (redisplays form to user displays error message)
- Owner: rule for determining field owner; only applies to Templates and is used to determine which user will be presented each field for data gathering

NOTE: Most of these rules are passed an Identity object. In provisioning policies, this is the Identity to whom the provisioning request pertains. An Identity may or may not be relevant to forms, so this Identity field may sometimes be null.

FieldValue

Description

The FieldValue rule sets the default value for a form field. In a provisioning policy, fields that are assigned a value with a rule (or any other method) are usually not presented to a user on a data-gathering form, so this becomes the defined value for the field, not a just default that can be overridden.

Definition and Storage Location

FieldValue rules are specified in the UI in the field definition of a provisioning policy or form. From the form editor or provisioning policy editor UI in 7.0, follow this navigation:

> Add Field (or click an existing field to edit) -> Value: select Rule -> Default Value Rule

In 7.1, with the new form editor UI, the navigation is:

> Add or edit a Field -> Value Settings -> Value: select Rule

The rule is referenced within the Field element of the Form object's XML.

```
<Field name="Field1" type="string">
  <RuleRef>
     <Reference class="sailpoint.object.Rule" name="[FieldValue Rule Name]"/>
  </RuleRef>
</Field>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
identity	sailpoint.object.Identity	Reference to the Identity for whom the field value is being set

NOTE: FieldValue rules are called from several different places inside IdentityIQ, and the list of arguments provided by each call can vary. As an example, if a field has dependencies, those are included in the arguments map to the rule. This rule type is a good candidate for using the rule code provided in *Printing the Beanshell Namespace* to understand the full set of arguments available to the rule.

Outputs:

Argument	Type	Purpose
value	java.lang.object	The value to set for the field

Example

This FieldValue rule generates a password value based on the associated Identity and application password policy.

```
import sailpoint.api.PasswordGenerator;
import sailpoint.object.PasswordPolicy;
import sailpoint.object.Application;

PasswordGenerator psswdGen = new PasswordGenerator(context);

String appName = field.getApplication();
Application app = context.getObjectByName(Application.class,appName);

String psswd = psswdGen.generatePassword(identity,app);

return psswd;
```

AllowedValues

Description

An allowedValues rule specifies the set of values to display in the drop-down list in a listbox presented on a provisioning policy or other form.

Definition and Storage Location

AllowedValues rules are specified in the UI in the field definition of a provisioning policy or form. From the form editor or provisioning policy editor UI in 7.0, follow this navigation:

Add Field (or click an existing field to edit) -> Allowed Values: select Rule -> Allowed Values Rule

In 7.1, with the new form editor UI, the navigation is:

Add or edit a Field -> Value Settings -> Allowed Values: select Rule

The rule is referenced within the Field element of the Template or Form object's XML.

```
<Field displayName="Field Name" name="fieldname" … >
  <AllowedValuesDefinition>
    <RuleRef>
<Reference class="sailpoint.object.Rule" id="402846023ac1d3f6013b04b2acc6078a"
name="[Allowed Values Rule Name]"/>
    </RuleRef>
  </AllowedValuesDefinition>
</Field>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
identity	sailpoint.object.Identity	Reference to the Identity to whom the provisioning policy or form applies
form	sailpoint.object.Form	Reference to the form object holding the field where the allowed values are being set; for provisioning policies, this is a form built at run-time based on the Template
field	sailpoint.object.Field	Reference to the field object in which the allowed

		values are being set

NOTE: The list of arguments provided to this rule can vary based on the field for which it is defined (e.g. there may be dependencies which would be included as arguments to the rule). This rule type is a good candidate for using the rule code provided in *Printing the Beanshell Namespace* to understand the full set of arguments available inside it.

Outputs:

Argument	Type	Purpose
values	java.lang.Object	Object (possibly a collection) containing the allowed values for a given field

Example

This example AllowedValues rule populates a drop-down list with only the regions that are currently assigned to existing Identities, listing them in alphabetical order.

```
import java.util.List;
import java.util.ArrayList;
import sailpoint.object.QueryOptions;
import sailpoint.object.Identity;

List values = new ArrayList();

QueryOptions qo = new QueryOptions();
qo.setDistinct(true);
qo.setOrderBy("region");

Iterator regions= context.search(Identity.class, qo, "region");
while (regions.hasNext()) {
    String region = (String) regions.next()[0];
     values.add(region);
}
return values;
```

Validation

Description

A Validation rule examines a Field value and determines whether it is valid, as specified in the rule logic. If it is not valid, one or more messages are returned from the rule; if the field value is valid, the rule should return null. When messages are returned from the rule, the form is reloaded for the user to correct the error and the messages are displayed on it.

Definition and Storage Location

Validation rules are specified in the UI in the field definition of a provisioning policy or form. From the form editor or provisioning policy editor UI in 7.0, follow this navigation:

Add Field (or click an existing field to edit) -> Validation: select Rule -> Validation Rule

In 7.1, with the new form editor UI, the navigation is:

> Add or edit a Field -> Value Settings -> Validation: select Rule

The rule is referenced within the Field element of the Template or Form object's XML.

```
<Field displayName="Field Name" name="fieldname" … >
   <ValidationRule>
       <Reference class="sailpoint.object.Rule" id="402846023ac1d3f6013b0a3a2d0707d8"
name="[Validation Rule Name]"/>
   </ValidationRule>
</Field>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
identity	sailpoint.object.Identity	Identity to whom the field value relates
app	sailpoint.object.Application	Reference to the Application object to which the Form applies
form	sailpoint.object.Form	Reference to the Form object on which the Field exists
field	sailpoint.object.Field	Reference to the Field being validated
value	java.lang.Object	Object representing the field value

NOTE: Like the Field Value and Allowed Values rules, this rule may have different sets of arguments provided based on the field for which it is defined; specifically, field dependencies can impact the argument set. This rule type is a good candidate for using the rule code provided in *Printing the Beanshell Namespace* to understand the full set of arguments available inside it.

Outputs:

Argument	Type	Purpose
messages	sailpoint.tools.Message, String, or Collection (List) of Messages or strings	List of messages from the validation process; IdentityIQ can process a Message, string, or a collection of Messages or strings as the return value from this rule If any non-null value is returned, this means validation has failed.

Example

This example Validation rule checks that the Identity name entered corresponds to an existing Identity who is a Manager.

```
String name = (String) value;
Identity ident = context.getObject(Identity.class, name);
if (null == ident)
    return "Identity does not exist.";
else if (!(ident.isManager()))
```

```
            return "Identity is not a manager.";
        return null;
```

This example Validation rule checks that the email address entered is in a correct format (containing an @ and a period .) and that it is not already connected to an Identity in the system.

```
        import java.util.ArrayList;
        import sailpoint.object.*;
        import java.util.Iterator;

        ArrayList messages = new ArrayList();
        String inputVal = (String)value;

        if (inputVal.indexOf("@") &lt; 0) {
            messages.add("Need an @ sign in a valid email address.");  }

        if (inputVal.indexOf(".") &lt; 0) {
            messages.add("Need a . in a valid email address.");  }

        QueryOptions qo = new QueryOptions();
        qo.addFilter(Filter.eq("email",inputVal));
        List users  = context.getObjects(Identity.class,qo);
        if (!users.isEmpty()) {
             Iterator iter = users.iterator();
             while (iter.hasNext()) {
                Identity identity = (Identity)iter.next();
                messages.add("Email address already in use by " + identity.getName());
            }
        }
        return messages;
```

Owner

Description

Owner Rules are used by role or application provisioning policies to determine the owner of the provisioning policy or its policy fields. The owner of a field or policy is the Identity who will be asked to provide any input values for the provisioning activity that could not be identified or calculated automatically by the system.

NOTE: Fields have an Owner field whether they belong to provisioning policies or forms. However, for forms, the field owner value is ignored. Therefore an Owner rule is only useful for provisioning policy fields.

Definition and Storage Location

Owner rules are specified in the UI in the field definition of a provisioning policy or as the provisioning policy owner. When creating or editing a provisioning policy (in an Application or Role definition), an Owner rule can be specified for the policy owner or for the Field owner.

(Provisioning Policy) Owner: select Rule -> Owner Rule

or

(within Provisioning Policy) Add Field (or click an existing field to edit) -> Owner: select Rule -> Owner Rule

or

(within Provisioning Policy in 7.1) add or edit field-> Type Settings -> Owner: select Rule

The rule is referenced within the Bundle or Application object's XML in the Form element or in the Field element to which the rule relates.

```
<Form name="test prov policy">
  <OwnerDefinition>
    <RuleRef>
      <Reference class="sailpoint.object.Rule" id="402846023ac1d3f6013ae71128ba03dd"
name="[Owner Rule Name for Provisioning Policy]"/>
    </RuleRef>
  </OwnerDefinition>
  <Field displayName="Field Name" name="fieldname" type="string">
    <OwnerDefinition>
      <RuleRef>
        <Reference class="sailpoint.object.Rule" id="402846023ac1d3f6013ae71128ba03f7"
name="[Owner Rule Name for Field]"/>
      </RuleRef>
    </OwnerDefinition>
  </Field>
</Form>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
identity	sailpoint.object.Identity	Reference to the Identity being provisioned
role	sailpoint.object.Bundle	Reference to the role object involved in the provisioning process (if applicable)
application	sailpoint.object.Application	Reference to the application object to which the provisioning will occur
template	sailpoint.object.Template	Reference to the template object that defines the provisioning policy form
field	sailpoint.object.Field	Reference to the field object being assigned an owner (if any)

NOTE: This rule may have different sets of arguments provided based on the field for which it is defined; specifically, field dependencies can impact the argument set. This rule type is a good candidate for using the rule code provided in *Printing the Beanshell Namespace* to understand the full set of arguments available inside it.

Outputs:

Argument	Type	Purpose
identity	sailpoint.object.Identity or string	The rule returns an Identity object, an Identity name, or one of several special keywords that can be used to identify the appropriate owner based on the role or application to which the

		provisioning policy is attached. Those keywords are: • "IIQParentOwner": resolves to the owner of the application or role to which the policy belongs • "IIQRoleOwner": the owner of the role to which the policy belongs • "IIQApplicationOwner": the owner of the application to which the policy belongs

Example

This example Owner rule selects a different owner (in this case, a workgroup) for the provisioning form based on whether the Identity being provisioned for is an employee or a contractor.

```
import sailpoint.object.Identity;

String status = identity.getAttribute("status");

Identity provOwner = null;

if ("Contractor".equals(status)) {
provOwner = (Identity) context.getObject(Identity.class,
"ContractorAppoverWorkgroup");
} else {
provOwner = (Identity) context.getObject(Identity.class, "EmployeeAppoverWorkgroup");
}

return provOwner;
```

Workflow Rules

All rules specified within workflows (business processes) are rules of type Workflow. Workflow rules return an "object" which can be any value required for the functionality that invokes the rule. Workflow rules are used for initializing variables, controlling transitions between steps, and even performing the action within steps.

NOTE: Though they are not explicitly named in the rule signature, all workflow arguments and process variables are automatically available to all workflow rules.

Workflow

Description

All rules specified as part of workflows are rules of type Workflow. This includes rule that set values for workflow variables and step arguments, rules that determine transition conditions between steps, and rules that contain step execution instructions.

Definition and Storage Location

Workflow rules are defined in several places within the business process editor in the UI.

Setup -> Business Processes -> New Process (or edit an existing process) -> any of the options listed below

- Process Variables -> Initial Value Rule
- Transition Rule
- Step -> Details -> Action Rule
- Step -> Arguments -> Value Rule

References to the rules are stored in the workflow XML. The rule ID value for is recorded in the corresponding element.

Process Variable initialization rule:

```
<Variable initializer="rule:402846023ac1d3f6013ae1d2be300363" name="var1"/>
```

Transition rule:

```
<Step … >
  <Transition to="Process1" when="rule:402846023ac1d3f6013ae1d1cd280361"/>
</Step>
```

Step action rule:

```
<Step action="rule:402846023ac1d3f6013ae1d0b59e035f" name="Process1">
</Step>
```

Step Argument initialization rule:

```
<Step … >
  <Arg name="arg1" value="rule:402846023ac1d3f6013ae1d338800364"/>
</Step>
```

NOTE: Manually-created workflow XML can reference the rule by name (e.g value="rule:My Rule Name"); workflows created through the business process editor will use the rule ID as shown above.

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
wfcontext	sailpoint.workflow.workflow Context	Reference to the current workflowContext
handler	sailpoint.workflow.workflow Handler	Workflow handler connected to the current workflowContext
workflow	sailpoint.object.workflow	Current workflow definition
step	sailpoint.object.step	Current step in the workflow
approval	sailpoint.object.approval	Current approval being processed
item	sailpoint.object.workItem	workItem being processed

NOTE: step, approval, and/or item may be null for some usages of this rule type.

NOTE: Since the list of arguments provided to rules of this type can vary based on their usage, this rule type is a good candidate for using the rule code provided in *Printing the Beanshell Namespace* to understand the full set of arguments available inside each instance.

Outputs:

Argument	Type	Purpose
object	java.lang.object	Value to be returned from the rule (depends on the rule's usage)

Example

This example workflow rule is a step action rule that determines the approver for a workItem based on the approvalScheme process variable values. If no approvers are found, it uses the fallbackApprover. If the approver list contains the Identity who initiated the workflow (i.e. the user who made the request that invoked the workflow), that Identity is removed from the list.

```
import sailpoint.object.ApprovalSet;
import sailpoint.object.ProvisioningPlan;
import sailpoint.object.WorkItem.State;

List approvers = new ArrayList();
if ( approvalSet != null ) {
    List items = approvalSet.getItems();
    // By default there is one item for all of the edits
    ApprovalItem item = null;
    if ( Util.size(items) > 0 )
        item = items.get(0);

    if ( item != null ) {
        approvers = getApproverNames(approvalScheme, item, plan, identityName);
        if ( approvers != null && approvers.size() == 0 &&
fallbackApprover != null ) {
            if ( log.isDebugEnabled() ) {
                log.debug("Approver could not be resolved. Using fallbackApprover
'"+fallbackApprover+"'.");
            }
            approvers.add(fallbackApprover);
        }
        // If the launcher is an approver remove them from the list
        if ( approvers != null && approvers.contains(launcher) ) {
            approvers.remove(launcher);
            // If this is the only approver, automatically mark the item approved.
            if ( Util.size(approvers) == 0 ) {
                item.setState(WorkItem.State.Finished);
                item.setOwner(launcher);
            }
        }
    }
}
return approvers;
```

Policy/Violation Rules

These rules relate to policies and policy violations defined for the installation.

Policy

Description

The Policy rules (or constraints) for Advanced policies can be defined through a Policy rule. The rule specifies the conditions for determining when the policy has been violated.

NOTE: This is actually a special case of an IdentitySelector rule that is provided more arguments (the policy and constraint) than a normal IdentitySelector rule and can return a full PolicyViolation object, rather than just a "true" or "false" value. By returning a PolicyViolation, the rule can specify more details about the appearance and structure of the violation, but this is not strictly required. If the rule returns a PolicyViolation, that violation will be added for the Identity as returned. If the rule returns a "true" value, a PolicyViolation will be created using the information available on the policy itself.

Definition and Storage Location

A Policy rule is specified in the UI in an Advanced Policy definition.

> Setup -> Policies -> create or edit an Advanced Policy -> Policy Rules section -> Create New Rule -> Selection Method: Rule

A reference to the rule is recorded in the Policy XML within the GenericConstraint and IdentitySelector elements.

```
   <GenericConstraints>
     <GenericConstraint created="1352402035651" id="402846023ac1d3f6013ae17163c30346"
name="Policy Rule" violationOwnerType="None">
       <IdentitySelector>
         <RuleRef>
           <Reference class="sailpoint.object.Rule"
id="402846023a65e596013a65e7a55704ff" name="[Policy Rule Name]"/>
         </RuleRef>
       </IdentitySelector>
     </GenericConstraint>
   </GenericConstraints>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
identity	sailpoint.object.Identity	Reference to the Identity object being inspected
policy	sailpoint.object.Policy	Reference to the policy object
constraint	sailpoint.object.Constraint	Reference to the Constraint object that defines the policy rule

Outputs:

Argument	Type	Purpose
violation	sailpoint.object.PolicyViolation	PolicyViolation object if Identity is in violation of the policy;

		null if no violation is detected

Example

This example Policy rule defines a policy that Identities should be logging in to every account they own at least every 180 days. Accounts with no login activity for more than 180 days are in violation of the policy. This rule returns a complete PolicyViolation object.

```
import sailpoint.api.SailPointContext;
import sailpoint.object.Attributes;

import sailpoint.object.Custom;
import sailpoint.object.Filter;
import sailpoint.object.Identity;
import sailpoint.object.QueryOptions;
import sailpoint.object.Policy;
import sailpoint.object.PolicyViolation;
import sailpoint.object.Link;

import sailpoint.tools.GeneralException;
import sailpoint.tools.Message;

import java.text.SimpleDateFormat;
import java.text.DateFormat;
import java.util.*;

/**
 * Returns a date <n> days before today.
 */
private Date getDateNDaysAgo(int numDays) {
        Calendar cal = Calendar.getInstance();
        Date returnDate = null;

        cal.add(Calendar.DATE, -(numDays));
        returnDate = cal.getTime();
        return (returnDate);
}

/**
* Checks if the first date is before the second date ignoring time.
**/

public static boolean isBeforeDay(Date date1, Date date2) {
    if (date1 == null || date2 == null) {
        throw new IllegalArgumentException("The dates must not be null");
    }
    Calendar cal1 = Calendar.getInstance();
    cal1.setTime(date1);
    Calendar cal2 = Calendar.getInstance();
    cal2.setTime(date2);
    if (cal1 == null || cal2 == null) {
        throw new IllegalArgumentException("The dates must not be null");
    }
    if (cal1.get(Calendar.ERA) < cal2.get(Calendar.ERA)) return true;
    if (cal1.get(Calendar.ERA) > cal2.get(Calendar.ERA)) return false;
    if (cal1.get(Calendar.YEAR) < cal2.get(Calendar.YEAR)) return true;
    if (cal1.get(Calendar.YEAR) > cal2.get(Calendar.YEAR)) return false;
    return cal1.get(Calendar.DAY_OF_YEAR) < cal2.get(Calendar.DAY_OF_YEAR);
}
```

```
    // Start of main rule logic

    PolicyViolation v = null;
    Date lastLoginDate = identity.getLastLogin();
    if (lastLoginDate == null)
        lastLoginDate = new Date();

    Date testDate = getDateNDaysAgo(180);
    if (isBeforeDay(lastLoginDate, testDate)) {
        v = new PolicyViolation();
        v.setActive(true);
        v.setIdentity(identity);
        v.setPolicy(policy);
        v.setConstraint(constraint);
        v.setDescription("[Last Login Date [" + lastLoginDate.toString()  + "] is more
than 180 days ago.]");
        v.setStatus(sailpoint.object.PolicyViolation.Status.Open);
    }

    return v;
```

Violation

Description

A Violation rule specifies the formatting for a policy violation. This generally means that it alters the description attribute on the PolicyViolation object. This is often used to describe the violation in user-friendly terms. In the case of Role and Entitlement SOD policies, this can be used to summarize a set of violations detected into a multi-line string description.

Definition and Storage Location

The Violation rule is specified in the UI as the Policy's Violation Formatting Rule.

> Setup -> Policies -> Create new policy (or select existing policy) -> Violation formatting rule

The rule name is recorded in the Policy's Attributes map as the violationRule.

```
<Policy name="Policy Name" … >
  <Attributes>
    <Map>
      <entry key="violationRule" value="[Violation Rule Name]"/>
    </Map>
  </Attributes>
</Policy>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
identity	sailpoint.object.Identity	Reference to the Identity object to whom the violation applies
policy	sailpoint.object.Policy	Reference to the policy object that has been violated
constraint	sailpoint.object.Constraint	Reference to the constraint, or policy rule, with in the policy that has been violated

violation	sailpoint.object.PolicyViolation	Reference to the policyViolation object that records the violation
state	java.util.Map	A Map that can be used to store and share data between executions of this rule

Outputs:

Argument	Type	Purpose
violation	sailpoint.object.PolicyViolation	Rule returns the altered policyViolation object

NOTE: The rule may either return a violation or alter the violation passed as an argument to the rule. The calling method overwrites the violation with the returned violation if one is returned; it resumes processing with the passed-in violation if the rule returns anything other than a PolicyViolation object (including no return value).

Example

This example Violation rule formats the violation description for a policy rule that flags users who have more than a given number of application accounts and has a high risk score. It creates a user-friendly description of the violation, stating the number of accounts the user holds and the risk score calculated for the user.

```
import sailpoint.object.Identity;
import sailpoint.object.Link;

int score = identity.getScore();
int numberOfLinks = identity.getLinks().size();

violation.setDescription("User has accounts on " + numberOfLinks + " resources with a
composite score of " + score + ".");

return violation;
```

The exampleRules.xml file in the [IdentityIQ Install Directory]/web-inf/config directory includes an additional example Violation rule called "Render SOD Entitlements" that provides an example of how to format a Role or Entitlement SOD policy violation into a multi-line string description.

PolicyOwner

Description

The PolicyOwner rule is used to determine the owner of a Policy Violation. Policy violation owners can be set for the whole policy or for individual rules, or constraints, defined within the policy.

Definition and Storage Location

The PolicyOwner rule is set in the UI through the Policy Definition.

Setup -> Policies -> Policy Violation Owner -> Rule

or

Setup -> Policies -> Create or edit Policy Rule -> Policy Violation Owner -> Rule

The reference to the rule is recorded in the Policy XML. This can exist at the policy level or within each defined constraint, depending on the level at which the owner rule is specified.

```
<Policy … name="SOD Policy" … violationOwnerType="Rule">
  <ViolationOwnerRule>
    <Reference class="sailpoint.object.Rule" id="402846023a65e596013a65e7a6de0501"
name="[Policy Owner Rule Name]"/>
  </ViolationOwnerRule>

or

  <SODConstraint … name="Accounting SOD-762" violationOwnerType="Rule" … >
    …
    <ViolationOwnerRule>
      <Reference class="sailpoint.object.Rule" id="402846023ac1d3f6013ae178111d034a"
name="[Policy Owner Rule Name]"/>
    </ViolationOwnerRule>
  </SODConstraint>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
identity	sailpoint.object.Identity	Reference to the identity to whom the violation relates (the policy violating identity)
policy	sailpoint.object.Policy	Reference to the policy to which the violation relates
constraint	sailpoint.object.BaseConstraint	Reference to the policy constraint that the Identity has violated; only passed when assigning a violation owner per each specific constraint – this argument is null when the violation owner is set at the whole policy level

NOTE: The signature for this rule changed in version 6.2p4 and version 6.3 to add the Policy and Constraint arguments. In previous versions, the rule editor information showed that it receives 3 parameters in addition to the common arguments: Environment (the task arguments), Policy, and Violation, but that was not correct information; the rule was previously passed only the violating Identity. Since the new signature includes the same arguments as the old plus additional ones, this change is backward compatible and old policyOwner rules will still work in newer versions.

Outputs:

Argument	Type	Purpose
owner	sailpoint.object.Identity	The identity to which ownership of the violation (and therefore responsibility for addressing it) should be assigned

Example

This example PolicyOwner rule returns the manager of the policy-violating Identity; if the Identity does not have a manager, the violation is owned by a hypothetical service account Identity named PolicyReviewer.

```
import sailpoint.object.Identity;

Identity owner = identity.getManager();
if (null == owner){
    owner = context.getObject(Identity.class, "PolicyReviewer");
}
return owner;
```

PolicyNotification

Description

The PolicyNotification rule is used to specify additional people who should be notified when a policy violation is discovered. This is expressed in the policy definition as a PolicyAlert Owner rule, and the rule type PolicyNotification is never referenced explicitly by IdentityIQ.

Definition and Storage Location

The PolicyNotification rule is an XML-only attribute shown as an attribute of the PolicyAlert component with a Policy definition. This is an alternative to specifying a username or set of usernames through the UI to notify when policy violations are created.

```
<Policy name="SOD Policy" … >
…
  <PolicyAlert … >
    <OwnerRule>
     <Reference class="sailpoint.object.Rule" name="[Policy Notification Rule Name]"/>
    </OwnerRule>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
environment	sailpoint.object.Attributes	Map of arguments passed to the policy checking process
policy	sailpoint.object.Policy	Reference to the policy to which the violation relates
violation	sailpoint.object.PolicyViolation	Reference to the policy violation created based on this policy analysis

Outputs:

Argument	Type	Purpose
listener	sailpoint.object.Identity or List<Identity> or String or List<String> or Map	Specify the user or users who should be notified of the violation; Can be one identity, an identity name, a List of Identities, a List of identity names, or a Map containing "Identity" with an

		Identity object or "identityName" with a string identity name

Example

This example PolicyNotification rule returns the manager (identity) of the policy-violating Identity. If the user has no manager, it returns null and no additional people are notified.

```
import sailpoint.object.Identity;

Identity listener = null;
Listener = policyViolation.getIdentity.getManager();

return listener;
```

Login Configuration Rules

There are 4 rules which relate to IdentityIQ login and authentication.

- SSOAuthentication
- SSOValidation
- SAMLCorrelation
- IdentityCreation

Beginning in version 6.3, IdentityIQ supports two different types of single sign-on configurations: rule-based SSO and SAML SSO. The SSOAuthentication and SSOValidation rules apply to the rule-based SSO, while the SAMLCorrelation rule applies to SAML SSO. (Prior to version 6.3, only rule-based SSO was supported.)

The IdentityCreation rule only applies to IdentityIQ login/authentication in the case of a failed pass-through authentication attempt, as described below.

SSOAuthentication

Description

The SSOAuthentication rule specifies how the user is authenticated and matched to an Identity for sign-on to IdentityIQ. Writing this rule is the only action required to implement rule-based single sign-on with IdentityIQ. Version 6.1 introduced the option of returning a Link (account) from this rule instead of an Identity; this option *must* be used when implementing Electronic Signatures with SSO authentication because this rule is used to validate the user for recording their electronic signature as well as for initial sign-on.

Definition and Storage Location

The SSOAuthentication rule is connected to the instance through the Login Configuration page.

> Gear menu -> Global Settings -> Login Configuration-> SSO Configuration -> check Enable Rule Based Single Sign-On (SSO) -> Single Sign-On Rule

The name of this rule gets stored in the attributes map of the System Configuration XML.

```
<entry key="loginSSORule" value="[SSOAuthentication Rule]"/>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
httpRequest	string	Contains header information, including the user's token from the SSO system

Outputs:

Argument	Type	Purpose
identity or link	sailpoint.object.Identity or sailpoint.object.Link	Specifies the Identity or the Link matched to the information passed in the header

Example

This example SSOAuthentication rule validates the HTTP header and then extracts the username from it to correlate to an Identity.

```
import sailpoint.object.Application;
import sailpoint.object.Identity;
import sailpoint.object.Link;
import sailpoint.tools.GeneralException;
import sailpoint.api.Correlator;
import sailpoint.api.SailPointContext;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

private String COOKIE = "cookie";
private String TRANSACTION_ID = "smtransactionid";
private String SERVER_SESSION = "smserversessionid";
private String AUTHDIR_OID = "smauthdiroid";
private String AUTHDIR_SERVER = "smauthdirserver";
private String AUTHDIR_NAME = "smauthdirname";
private String USER_DN = "smuserdn";
private String USERNAME = "smuser";
private String[] HEADER_ATTRS = { TRANSACTION_ID, SERVER_SESSION, AUTHDIR_SERVER,
AUTHDIR_NAME, USER_DN, USERNAME, COOKIE };

/**
* Make sure we have the values we know about. May vary by
* version of SiteMinder.
*/
private void validateHeader() {
      for ( String header : HEADER_ATTRS ) {
String value = httpRequest.getHeader( header );
if ( value == null ) {
throw new GeneralException( "Invalid Site-Minder session." + " Missing variable [" +
header + "]" );
}
      }
}
```

```
// Rule processing starts here

validateHeader();

String username = httpRequest.getHeader( USERNAME );
Identity user = new Identity();

// Ask the correlator to find us the Link associated with the
// username we stripped from the header
Correlator correlator = new Correlator(context);

if ( username != null ) {
      user = correlator.findIdentityByAttribute("uid", username);
      if ( user == null ) {
            throw new GeneralException("Unable to find Link associated: " +
username);
      }
}
return user;
```

SSOValidation

Description

The SSO Validation rule, if defined, runs on every page change as a user navigates through IdentityIQ; it validates the session with the SSO provider by examining the headers through the httpRequest. This frequent validation provides an added measure of verification for those clients with extraordinary concerns for security, but it can impact system performance. If the SSO Validation Rule cannot verify a valid SSO session it logouts the user and displays an error message (as specified by the rule creator).

This rule was added in IdentityIQ version 6.0 to support a specific customer request and is not likely to be used in most installations.

Definition and Storage Location

The SSO Validation rule is connected to the instance through the Login Configuration page.

> Gear menu -> Global Settings -> Login Configuration-> SSO Configuration -> check Enable Rule Based Single Sign-On (SSO) -> Single Sign-On Rule

The name of this rule gets stored in the attributes map of the System Configuration XML.

```
<entry key="loginSSOValidationRule" value="[SSOValidation Rule]"/>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
httpRequest	string	Contains header information, including the user's token from the SSO system

Outputs:

Argument	Type	Purpose
resultMessage	string	Returns the error message to display, indicating that the validation failed; returns null if validation was successful

Example

This example SSO Validation rule validates the HTTP header to ensure it contains the expected elements.

```
import sailpoint.tools.GeneralException;

private String COOKIE_TOKEN = "CTINTERNAL";
private String COOKIE = "cookie";
private String REQUEST_ID = "ct_request_id";
private String SERVER_SESSION_TIME = "ct-session-init-time";
private String USER_DN = "ct-remote-user";
private String[] HEADER_ATTRS = { REQUEST_ID, SERVER_SESSION_TIME, USER_DN, COOKIE };

// Iterate through the header attributes
for ( String header : HEADER_ATTRS ) {
   String value = httpRequest.getHeader(header);
   // Check that none of these header values is null
   if ( value == null )
      return ("Invalid Clear Trust session."+" Missing variable ["    +header+"]");
   // Check that the "cookie" attribute contains required value
   if (header.contentEquals(COOKIE)){
      if (!value.contains(COOKIE_TOKEN))
   return ("Invalid Clear Trust session."+ " Missing CT Session cookie ["
+header+"]");
   }
}
```

SAMLCorrelation

The SAMLCorrelation rule provides the logic for mapping the assertion details provided by the identity provider to an Identity for sign-on to IdentityIQ. SAML SSO was introduced in version 6.3 of IdentityIQ, so this rule applies to 6.3+ versions.

Definition and Storage Location

The SAMLCorrelation rule is connected to the instance through the Login Configuration page.

> Gear menu -> Global Settings -> Login Configuration-> SSO Configuration -> check Enable SAML Based Single Sign-On (SSO) -> SAML Correlation Rule

The name of this rule gets stored in the Configuration object called SAML, as a rule reference within the SAMLConfig element.

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose

assertionAttributes	HashMap	A map of (string) key-value pairs provided by the Identity Provider; will always contain a key NameId (value is the name Id sent by the Identity Provider) and any other SAML assertion attributes

Outputs:

Argument	Type	Purpose
identity or link	sailpoint.object.Identity or sailpoint.object.Link	Specifies the Identity or the Link matched to the information passed in the assertionAttributes

NOTE: As with the SSOAuthentication rule, the rule must return a Link (account) instead of an Identity when implementing Electronic Signatures with SSO authentication because this rule is used to validate the user for recording their electronic signature as well as for initial sign-on.

Example

This example SAMLCorrelation rule takes the nameId attribute provided by the Identity Provider and looks up the identity which has that name in IdentityIQ. (This, of course, assumes that the identity name matches the value provided by the Identity Provider.)

```
import sailpoint.object.Identity;

// Get the nameId from the assertionAttributes
String nameId = (String)assertionAttributes.get("nameId");

Identity ident;

if(nameId != null) {
   // Lookup the identity based on nameId
   ident = context.getObject(Identity.class, nameId);
}

return ident;
```

IdentityCreation

Description

An IdentityCreation rule (also used in aggregation/refresh) can be specified as an Auto-Create User Rule in the IdentityIQ Login Configuration to automatically create an Identity following a failed attempt at pass-through authentication. When no Identity matching the entered username credential was found on the pass-through application, IdentityIQ creates an Identity for the user, and this rule can customize attributes for the Identity.

Definition and Storage Location

An Auto-Create User Rule is specified in the login configuration:

Gear menu -> Global Settings -> Login Configuration -> Login Settings -> Auto-Create User Rule

The Auto-Create User Rule is recorded in the RuleRegistry XML as a RuleCallout. The rule in its entirety is copied into the RuleRegistry as the value for this callout.

```xml
<RuleRegistry created="1350329289215" id="402846023a65e596013a65e5c9ff00eb" name="Rule
Registry">
  <Registry>
    <Map>
      <entry>
        <key>
          <RuleCallout>AUTO_CREATE_USER_AUTHENTICATION</RuleCallout>
        </key>
        <value>
           <!-- Whole rule is copied here automatically -->
        </value>
      </entry>
    </Map>
  </Registry>
```

Arguments and Example

See the *IdentityCreation* rule under Aggregation/Refresh Rules for the argument list for this rule type and an example rule.

Logical Application Rules

CompositeAccount

Description

The CompositeAccount Rule is used to generate logical application account links. This is an alternative to specifying a correlation strategy on the logical application's tiers. If the rule exists, it will be used in place of the tier definition correlation; if it does not exist, the Identity's logical application accounts are determined by matching the Identity to the tiers using the IdentitySelector object defined on the Tier. Given an Identity, the rule determines if the Identity should have an account on the logical application, and if so, creates and returns a Link object or list of Links.

Definition and Storage Location

A CompositeAccount Rule can be created and associated to a Logical application in the application definition.

> Applications -> Application Definition -> Create new or edit existing application of Application Type: Logical -> Tiers -> Account Rule

The rule name is recorded in the logical application's attributes map as the "accountRule" within a CompositeDefinition element.

```xml
<Application connector="sailpoint.connector.DefaultLogicalConnector" name="My Logical
App" type="Logical" … >
<Attributes>
    <Map>
      <entry key="compositeDefinition">
```

```
            <value>
              <CompositeDefinition accountRule="[CompositeAccount Rule Name]" … >
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
identity	sailpoint.object.Identity	Reference to the Identity to evaluate and determine whether or not it should have an account (link) on the logical application
application	sailpoint.object.Application	Reference to the application object representing the logical application

Outputs:

Argument	Type	Purpose
links	sailpoint.object.Link or List <sailpoint.object.Link>	Rule returns a single Link object or a list of one or more Links objects that will be connected to the Identity

Example

This example CompositeAccount rule determines that an Identity should have an account on the logical application if that Identity has accounts with the same nativeIdentity value on all of the tier applications defined in the logical application definition. A link is returned for each nativeIdentity value held by the Identity on all of the tier application (e.g. if the Identity has a JohnSmith and a JohnSmithAdmin account on all tier applications, the list returned from the rule will contain Links for the logical application for each of those nativeIdentity values).

```
import sailpoint.object.Application;
import sailpoint.object.Identity;
import sailpoint.object.Link;
import java.util.ArrayList;
import sailpoint.object.CompositeDefinition;

List composites = null;

// Get the tiers, the names are passed through the Application.
CompositeDefinition def = application.getCompositeDefinition();

if (def != null){
    List tiers = def.getTiers();
    if (tiers == null || tiers.size() < 2) {
        log.error("Must have two or more application names specified.");
        return null;
    }

    // Get Identity's current links for all tiers.
    List apps = new ArrayList(tiers.size());
    Map linksByApp = new HashMap(tiers.size());

    String primaryTierApp = def.getPrimaryTier() != null ? def.getPrimaryTier() : "";
```

```
Application primaryTier = null;

for (Iterator it=tiers.iterator(); it.hasNext(); ) {
    String appName = ((CompositeDefinition.Tier) it.next()).getApplication();
    Application app = context.getObject(Application.class, appName);

    if (primaryTierApp.equals(appName)){
        primaryTier = app;
    }
    if (null != app) {
        List links = identity.getLinks(app);
        if ((null != links) && !links.isEmpty()) {
            apps.add(app);
            linksByApp.put(app, links);
        }
    }
}

if (tiers.size() == linksByApp.size()) {
    List topTierLinks = (List) linksByApp.get(primaryTier);

    for (int i = 0; i < topTierLinks.size(); i++) {
        Link link1 = (Link) topTierLinks.get(i);
        String id = link1.getNativeIdentity();

        List componentLinks = new ArrayList();

        // Other tiers must have a link with the same name.  We're starting
        // at index 1 because we're already looking at the top tier app.
        for (int j = 1 ; j < apps.size() ; j++) {
            Application app = (Application) apps.get(j);
            List linksForTier = (List) linksByApp.get(app);
            boolean foundMatchInTier = false;
            for (Iterator it=linksForTier.iterator(); it.hasNext(); ) {
                Link link2 = (Link) it.next();

                if (id.equalsIgnoreCase(link2.getNativeIdentity())) {
                    foundMatchInTier = true;
                    componentLinks.add(link2);
                    break;
                }
            }

            // If we didn't find a match, quit looking.
            if (!foundMatchInTier) {
                break;
            }

            Link composite = null;

            // If we have components in all tiers, we found a composite account.
            if (componentLinks != null && apps != null && componentLinks.size() ==
apps.size() - 1) {

                composite = new Link();
                composite.setApplication(application);
                composite.setNativeIdentity(id);
                composite.addComponent(link1);

                for (Iterator it = componentLinks.iterator(); it.hasNext(); ) {
                    Link theLink = (Link) it.next();
```

```
                         if (theLink != null){
                            if (theLink.getAttributes() != null){
                                groupmbr = theLink.getAttributes().get("groupmbr");
                                if (null != groupmbr)  {
                                    composite.setAttribute("groupmbr", groupmbr);
                                }
                                directPermissions =
theLink.getAttributes().get("directPermissions");
                                if (null != directPermissions)  {
                                    composite.setAttribute("directPermissions",
directPermissions);
                                }
                            }
                            composite.addComponent(theLink);
                        }
                    }
                }

                // outer loop: continue processing top tier accounts in case
                // we have more than one stack
                if (composite != null) {
                    if (composites == null)
                        composites = new ArrayList();
                    composites.add(composite);
                }
            }
        }
    }
}

    return composites;
```

CompositeRemediation

Description

The CompositeRemediation rule is called when provisioning needs to be performed against logical accounts. It is passed the provisioning plan built by the plan compiler and alters that plan so the request is directed at the component applications, rather than the logical application. The rule is meant to build a separate modified provisioning plan and return that plan. If the rule returns null, IdentityIQ uses the plan passed to the rule in subsequent processing, so the rule author can choose to have the rule directly modify the plan passed to it instead.

Definition and Storage Location

The CompositeRemediation rule is specified in the application definition for a logical application.

> Applications -> Application Definition -> Create new or edit existing application of Application Type: Logical -> Tiers -> Provisioning Rule

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
identity	sailpoint.object.Identity	Reference to the Identity object for whom the provisioning request has been made

| plan | sailpoint.object.ProvisioningPlan | Reference to a provisioning plan against the logical application |
| application | sailpoint.object.Application | Reference to the (logical) application object on which the rule is defined |

Outputs:

Argument	Type	Purpose
provisionPlan	sailpoint.object.ProvisioningPlan	converted provisioning plan that targets the applications that make up the logical application.

Example

This example CompositeRemediation rule makes modifications to the AccountRequest in the plan that relates to the logical application FinanceApp. AccountRequests for FinanceApp will be sent to the CorpDirectory tier application. The group attribute on CorpDirectory is "group" instead of "groupMember" (as is reflected in the FinanceApp schema), so AttributeRequests referring to the attribute name "groupMember" must be changed to refer to "group". Any non-FinanceApp-related requests that exist in the plan are kept intact and copied into a new provisioning plan exactly as they came in from the original plan, along with the converted requests for FinanceApp/CorpDirectory.

```
import sailpoint.tools.GeneralException;
import sailpoint.tools.Util;
import sailpoint.object.Identity;
import sailpoint.object.ProvisioningPlan;
import sailpoint.object.ProvisioningPlan.AccountRequest;
import sailpoint.object.ProvisioningPlan.AttributeRequest;

ProvisioningPlan updatedPlan = null;
if ( plan != null ) {
    // Get the account request for the composite application from the plan by app name
    AccountRequest compositeRequest = plan.getAccountRequest("FinanceApp");
    if ( compositeRequest != null ) {
        List convertedAttributeRequests = new ArrayList();
        // Convert the attribute requests that reference groupMember to just groups
        List attributeRequests = compositeRequest.getAttributeRequests();
        if ( Util.size(attributeRequests) > 0 ) {
            for ( AttributeRequest request : attributeRequests ) {
                String attributeName = request.getName();
                if ( "groupMember".compareTo(attributeName) == 0 ) {
                    AttributeRequest req = new AttributeRequest(request);
                    req.setName("groups");
                    convertedAttributeRequests.add(req);
                } else {
                    convertedAttributeRequests.add(new AttributeRequest(req));
                }
            }
        }
        List updatedAccountRequests = new ArrayList();

        // add in any other request that are part of the plan if any
        List accountRequests = plan.getAccountRequests();
```

```
        if ( Util.size(accountRequests) > 0 ) {
            for ( AccountRequest accountRequest : accountRequests) {
                String appName = accountRequest.getApplication();
                if ( "FinanceApp".compareTo(appName) != 0 ) {
                    updatedAccountRequests.add(accountRequest);
                }
            }
        }
        // Convert the "FinanceApp" request to "CorpDirectory"
        // and add it to the updated account requests
        AccountRequest convertedRequest = new AccountRequest(compositeRequest);
        convertedRequest.setApplication("CorporateDirectory");
        convertedRequest.setAttributeRequests(convertedAttributeRequests);
        updatedAccountRequests.add(convertedRequest);

        // create new plan and add the list of account requests to it
        updatedPlan = new ProvisioningPlan(plan);
        updatedPlan.setAccountRequests(updatedAccountRequests);
    }
}
return updatedPlan;
```

CompositeTierCorrelation

Description

The CompositeTierCorrelation rule correlates tier accounts to the primary application account for a given Identity. This rule is only specified when simple attribute matching is insufficient to identify the correct tier account (e.g. when an Identity has multiple accounts on a tier application and only a subset of them should be correlated to the tier as part of the logical application.

Definition and Storage Location

The CompositeTierCorrelation rule must be specified in the Application XML as an attribute on the Tier. There is no UI option for specifying this rule.

```
<Application connector="sailpoint.connector.DefaultLogicalConnector" … name="MyApp"
type="Logical">
  <Attributes>
    <Map>
      <entry key="compositeDefinition">
        <value>
          <CompositeDefinition … >
            <Tiers>
<Tier application="TierAppName" correlationRule="[CompositeTierCorrelation Rule]/>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
identity	sailpoint.object.Identity	Reference to the Identity object for which the tier correlation is being done
tierApplication	sailpoint.object.Application	Reference to the application object represented by the tier being correlated
primaryLink	sailpoint.object.Link	Reference to the link object (account) held

		by the Identity on the primary application

Outputs:

Argument	Type	Purpose
links	sailpoint.object.Link or list of links	One or more links on the tier application that correlate to the tier

Example

This CompositeTierCorrelation rule retrieves all of the Identity's links associated with the tierApplication and examines attributes on them to determine which should be the correlated links.

```
Filter f = Filter.and(Filter.eq("application", tierApplication),
            Filter.eq("identity", identity));

QueryOptions qo = new QueryOptions();
qo.addFilter(f);
List appLinks = Context.getObjects(Link.class,qo);
List corrLinks = new ArrayList();

if (null != appLinks) {
    for (Link appLink : appLinks) {
        // Only add active accounts to the correlation link list
        String inactiveAttr = (String) appLink.getAttribute("inactive");
        if (inactiveAttr.equals("false")) {
            corrLinks.add(appLink);
        }
    }
}
return corrLinks;
```

Unstructured Targets Rules

A few application types (e.g. Active Directory, SharePoint) have the capacity to track access that is not readily discoverable in a centralized location but must be gathered by traversing directory trees or sites. These permissions are gathered using unstructured configs that define how to access their data. These rules are used to manipulate the collected data in various ways.

TargetCreation

Description

This rule is called when a Target is created, allowing manipulation of the target before it is saved. It is most commonly used to filter out unwanted targets before they are created (e.g. objects that have only "system access" and therefore will not correlate to anything or targets that do not need to be tracked because they are unrestricted).

Definition and Storage Location

The TargetCreation rule is connected to the application in the UI through the application definition.

Applications -> Application Definition -> Edit or Create new application of Application Type: Active Directory, Active Directory Full, Microsoft SharePoint, or Microsoft SharePoint Online -> Unstructured Targets -> New Unstructured Data Source (or click existing) -> Create Target Source (or select existing) -> Creation Rule

In IdentityIQ 7.1, the SecurityIQ application type defines unstructured targets as a schema, so the TargetCreation rule for that application is specified here:

Applications -> Application Definition -> Edit or Create new application of Application Type: SecurityIQ -> Rules -> Schema Rules: unstructured -> Creation Rule

The reference to the rule is recorded in the TargetSource XML, which is in turn referenced by the Application object.

```
<TargetSource collector="sailpoint.unstructured.SharePointRWTargetCollector" …>
  <CreationRule>
    <Reference class="sailpoint.object.Rule" id="402846023a65e596013a65e789c004ff"
name="[TargetCreation Rule Name]"/>
  </CreationRule>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
application	sailpoint.object.Application	Reference to the Application object that owns the Target
target	sailpoint.object.Target	Reference to the Target object being created
targetSource	sailpoint.object.TargetSource	Source of the configuration for the collector

Outputs:

Argument	Type	Purpose
target	sailpoint.object.Target	The Target object as modified by the rule

NOTE: The rule can modify the target that is passed in as a parameter. If the rule returns a Target object, that object will replace the target passed to the rule. If the rule returns anything else (including no return value), the target passed to the rule is used in subsequent processing. As a result, if the target was modified directly by the rule, those modifications are applied even if the rule does not explicitly return it.

Example

This example TargetCreation rule prevents an unwanted target from being created. It examines the target's Name attribute for the value "C:\tmp\" and returns a null Target when that is found; returning a null Target object causes it to be filtered and therefore not created.

```
import sailpoint.object.Target;
```

```
import sailpoint.tools.Util.

String targetName = target.getName();
if ( Util.compareTo(targetName, "C:\tmp\") == 0 )  {
    Target nullTarget = new Target();
    return nullTarget;
}
```

TargetCorrelation

Description

This rule determines how the permission data gathered through unstructured configs is correlated to a link or group in IdentityIQ. The correlation is done by IdentityIQ based on the attribute name and value returned from this rule.

Definition and Storage Location

The TargetCorrelation rule is connected to the application in the UI through the application definition.

> Applications -> Application Definition -> Edit or Create new application of Application Type: Active Directory, Active Directory Full, Microsoft SharePoint, or Microsoft SharePoint Online -> Unstructured Targets -> New Unstructured Data Source (or click existing) -> Create Target Source (or select existing) -> Correlation Rule

In IdentityIQ 7.1, the SecurityIQ application type defines unstructured targets as a schema, so the TargetCreation rule for that application is specified here:

> Applications -> Application Definition -> Edit or Create new application of Application Type: SecurityIQ -> Rules -> Schema Rules: unstructured -> Correlation Rule

The reference to the rule is recorded in the TargetSource XML, which is in turn referenced by the Application XML.

```
<TargetSource collector="sailpoint.unstructured.SharePointRWTargetCollector" …>
  <CorrelationRule>
    <Reference class="sailpoint.object.Rule" id="402846023a65e596013a65e789c004db"
name="[TargetCorrelation Rule Name]"/>
  </CorrelationRule>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
application	sailpoint.object.Application	Reference to the application object on which the targets exist
nativeId	String	The group or user's native identity on the target
target	sailpoint.object.Target	Reference to the collected target
targetSource	sailpoint.object.TargetSour	Reference to the config that drives the

	ce	collection process
isGroup	boolean	Flag indicating whether the target represent a group (as opposed to a user account)

Outputs:

Argument	Type	Purpose

result	java.util.Map	Map containing the appropriate group or link attribute name and value to correlate the discovered permission to an entity:
		If permission belongs to a group:
		  "groupAttributeName","[attribute name]"
		  "groupAttributeValue","[value that identifies the group]"
		or
		  "group","[ManagedAttribute object for the group]"
		
		If permission belongs an account:
		  "linkIdentity","[GUID for the correlated Link]"
		or
		  "linkDisplayName","[display name attribute value for the Link]"
		or
		  "linkAttributeName","[attribute name]"
		  "linkAttributeValue","[value that identifies the account]"
		
		NOTE: If Link is on an application that includes Instances, a "linkInstance" attribute should be included in the map to specify the application instance to match to as well.

Example

This TargetCorrelation rule identifies "objectSid" as the correlation attribute and the nativeId value from the target as the correlation value.

```
import sailpoint.api.Correlator;
import sailpoint.tools.xml.XMLObjectFactory;

private String ATTR_OBJECT_SID = "objectSid";
Map returnMap = new HashMap();
if ( isGroup ) {
   returnMap.put(Correlator.RULE_RETURN_GROUP_ATTRIBUTE,ATTR_OBJECT_SID);
   returnMap.put(Correlator.RULE_RETURN_GROUP_ATTRIBUTE_VALUE,
     nativeId);
} else {
   returnMap.put(Correlator.RULE_RETURN_LINK_ATTRIBUTE, ATTR_OBJECT_SID);
   returnMap.put(Correlator.RULE_RETURN_LINK_ATTRIBUTE_VALUE, nativeId);
}
return returnMap;
```

TargetRefresh

Description

This rule allows the installation to manipulate attributes of an Unstructured Target during a Target Aggregation task if that target has previously been created. The TargetCreation rule is run when new targets are being created while this one runs for existing targets. This rule type was introduced in version 7.1.

Definition and Storage Location

The TargetRefresh rule is connected to the application in the UI through the application definition.

> Applications -> Application Definition -> Edit or Create new application of Application Type: Active Directory, Active Directory Full, Microsoft SharePoint, or Microsoft SharePoint Online -> Unstructured Targets -> Add New Unstructured Data Source (or click existing) -> Create Target Source (or select existing) -> Refresh Rule

In IdentityIQ 7.1, the SecurityIQ application type defines unstructured targets as a schema, so the TargetCreation rule for that application is specified here:

> Applications -> Application Definition -> Edit or Create new application of Application Type: SecurityIQ -> Rules -> Schema Rules: unstructured -> Refresh Rule

The reference to the rule is recorded in the TargetSource XML, which is in turn referenced by the Application XML.

```
<TargetSource collector="sailpoint.unstructured.SharePointRWTargetCollector" …>
  <RefreshRule>
    <Reference class="sailpoint.object.Rule" name="TargetRefresh Rule Name"/>
  </RefreshRule>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
application	sailpoint.object.Application	Reference to the application object on which the targets exist
target	sailpoint.object.Target	Reference to the collected target
targetSource	sailpoint.object.TargetSource	Reference to the config that drives the collection process

Outputs:

Argument	Type	Purpose
target	Sailpoint.object.Target	The Target object as modified by the rule

Example

This example TargetRefresh rule set the displayName of the target to match the filename, rather than the full path of the resource.

```
import sailpoint.object.Target;
import sailpoint.tools.Util.

String targetName = target.getName();
int index = targetName.lastIndexOf("\\");
String fileName = targetName.substring(index + 1);
target.setDisplayName(fileName);
return target;
```

TargetTransformer

Description

This rule is a seldom-used rule for manipulating a target object during target collection. It is similar to a Build Map rule for a connector schema, but applies specifically to target collection/manipulation and is only available in the WindowsTargetCollector and the original (pre-7.2) SecurityIQTargetCollector.

Definition and Storage Location

The TargetTransformer rule is not available in the IdentityIQ UI. The reference to the rule is recorded in the TargetSource XML, which is in turn referenced by the Application XML.

```
<TargetSource collector="sailpoint.unstructured.WindowsTargetCollector" …>
  <TransformationRule>
    <Reference class="sailpoint.object.Rule" name="Windows Transform Rule"/>
  </TransformationRule>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
collector	sailpoint.unstructured.Windows TargetCollector or SecurityIQTargetCollector	Reference to the target collector from which the targets are being read
target	sailpoint.object.Target	Reference to the collected target
targetSource	sailpoint.object.TargetSource	Reference to the config that drives the collection process

Outputs:

Argument	Type	Purpose
target	Sailpoint.object.Target	The Target object as modified by the rule

Alert Processing Rules

IdentityIQ version 7.1p1 introduced alert processing functionality. This was added to allow IdentityIQ to trigger processes based on alerts aggregated from SailPoint's unstructured data management solution, SecurityIQ, though it could also be used for handling alerts from other sources. IdentityIQ 7.2 made some improvements to this alert processing functionality, including definition of some new rule types around alert creation/correlation.

Applications which define an Alert schema (which includes only SecurityIQ, out of the box) will support creation and execution of an AlertCreation and AlertCorrelation rule.

NOTE: Alert aggregation can also execute a customization rule, just like any other aggregation process for any other schema type. The usage of a customization rule in alert aggregation is generally redundant with the AlertCreation rule because existing alerts are never updated in place (unlike accounts and groups) - meaning that the Creation rule will run for every alert processed and can do any customization required. Consequently, it is unlikely that most customers will use that rule type in this specialized aggregation process. Refer to the *ResourceObjectCustomization* content for more information if you want to use this rule type in alert processing.

AlertCreation

The AlertCreation rule runs during alert aggregation, as the alert record is being created in IdentityIQ. It is primarily used for populating attributes of the Alert record that are not auto-populated by the connector. Common examples include the Display Name or Alert Date attributes which will be included in the summary data shown on the Alerts list in the UI if populated.

NOTE: If the AlertCreation rule returns null, this will cause the alert aggregation process to skip the alert (i.e. not create it in IdentityIQ).

Definition and Storage Location

An AlertCreation rule is associated to the application which contains the Alert schema (e.g. SecurityIQ) through the Rules tab and can be defined through the rule editor UI.

> Applications -> Application Definition -> choose/create application with alert schema (e.g. SecurityIQ) -> Rules -> Schema Rules: alert -> Creation Rule

In the Application XML, the rule reference is stored within the alert schema:

```
<Application connector="sailpoint.connector.SecurityIQConnector" name="SecurityIQ"
profileClass="" type="SecurityIQ">
…
<Schemas>
    <Schema … objectType="alert">
    …
      <CreationRule>
        <Reference class="sailpoint.object.Rule" name="Alert Creation"/>
      </CreationRule>
    </Schema>
…
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
application	sailpoint.object. Application	The application object from which the alert was aggregated
alert	sailpoint.object. Alert	The alert object to be evaluated against the rule's conditions to determine if the alertDefinition's response should be executed or not

Outputs:

Argument	Type	Purpose
alert	sailpoint.object.Alert	The alert object being created (with any updates made by the rule)

Example

This example AlertCreation rule populates the alert's displayName attribute from other data that is included in the alert. Specifically, this rule sets the displayname to username: action.

```
import sailpoint.object.Alert;

String user = alert.getAttribute("userFullName");
String action = alert.getAttribute("actionType");

alert.setDisplayName(user + ":" + action);

return alert;
```

AlertCorrelation

The AlertCorrelation rule allows the alert to be associated to some other object in IdentityIQ. IdentityIQ allows an alert to be correlated to any object, though this correlation typically maps to the identity who owns the account which performed the action that generated the alert or to the account (Link) itself. The correlation records the object name and ID on the alert, rather than attaching the alert to the object. This correlation is used in the alert response – the object is provided to the email, workflow, or certification event that is triggered for the alert by an AlertDefinition.   The AlertCorrelation rule runs just before the AlertCreation rule.

Definition and Storage Location

An AlertCorrelation rule is associated to the application which contains the Alert schema through the Rules tab and can be defined through the rule editor UI.

Applications -> Application Definition -> choose/create application with alert schema (e.g. SecurityIQ) -> Rules -> Schema Rules: alert -> Correlation Rule

In the Application XML, the rule reference is stored within the alert schema:

```
<Application connector="sailpoint.connector.SecurityIQConnector" name="SecurityIQ"
profileClass="" type="SecurityIQ">
```

```
…
<Schemas>
    <Schema … objectType="alert">
     …
      <CorrelationRule>
        <Reference class="sailpoint.object.Rule" name="Alert Correlation"/>
      </CorrelationRule>
</Schema>
…
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
source	sailpoint.object. Application	The application object from which the alert was aggregated
alert	sailpoint.object. Alert	The alert object to be evaluated against the rule's conditions to determine if the alertDefinition's response should be executed or not

Outputs:

Argument	Type	Purpose
object	sailpoint.object.SailPointObject	Returns the object which should be correlated to the alert

Example

This example AlertCorrelation rule identifies the AD account which contains the matching account name (specified in a Link attribute called siqAccountName) and returns the identity which owns that account, thereby correlating the alert to that identity.

```
import sailpoint.object.Alert;
import sailpoint.object.Link;
import sailpoint.object.QueryOptions;
import sailpoint.object.Filter;

String username = alert.getAttribute("userFullName");
QueryOptions qo = new QueryOptions();
qo.addFilter(Filter.eq("siqAccountName",username));

List accts = context.getObjects(Link.class, qo);

if (null != accts) {
   if (accts.size() > 1) {
       log.warn("too many matched accounts: " + accts.size());
       return null;
   }
   for (Link ADAcct : accts) {
        return ADAcct.getIdentity();

   }
}
```

AlertMatch

An AlertDefinition specifies the desired response in IdentityIQ for a certain set of alerts read from an alert source system (like SecurityIQ), and it includes logic for identifying which alerts should trigger the defined response. An AlertMatch rule is one way to define that identifying logic (a set of Match Terms is the other).

Definition and Storage Location

An AlertMatch rule can be associated to an AlertDefinition on the Alert Definition page. There is no rule editor available on this page, however, so the rule must be written in XML and imported into IdentityIQ.

> Setup -> Alerts -> Alert Definitions -> create/edit Alert Definition -> Selector -> Match Rule

In the AlertDefinition XML, this is stored within the AlertMatchConfig:

```
<AlertDefinition name="Financials Run Cert">
…
  <AlertMatchConfig>
    <MatchRule>
      <Reference class="sailpoint.object.Rule" name="financial alert match"/>
    </MatchRule>
  </AlertMatchConfig>
…
</AlertDefinition>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
alert	sailpoint.object.Alert	The alert object to be evaluated against the rule's conditions to determine if the alertDefinition's response should be executed or not

Outputs:

Argument	Type	Purpose
status	Boolean	Returns true if the alert should execute the alertDefinition's response (i.e. it meets the identifying criteria for the specified response)

Example

This example AlertMatch rule examines the alertRuleNames attribute of the Alert object to determine if the alert was created in response to the Alert on Financials rule. If it was, it should trigger this alert response in IdentityIQ (return true); if not, it should not.

```
List alertRuleNames = alert.getAttribute("alertRuleNames");

if (null != alertRuleNames) {
```

```
    for (String alertRule : alertRuleNames) {

        if ("Alert on Financials".equals(alertRule))
        {
          return true;
        }
      }
  }
    else
      log.warn("no rule names retrieved");

    return false;
```

Activity Data Source Rules

These rules relate to the processing of activity data for various applications. The first two --ActivityCorrelation and ActivityTransformer -- manage activity manipulation for all systems on which activity data can be collected. The last two -- ActivityConditionBuilder and ActivityPositionBuilder --relate only to JDBC activity data sources.

ActivityTransformer

Description

The ActivityTransformer rule manipulates the activity data read from the activity data source to transform it into the ApplicationActivity format that is required for IdentityIQ to record it. The activity data is passed to this rule in different ways, depending on the activity data source and activity collector used.

Definition and Storage Location

The ActivityTransformer rule is connected to the application in the UI through the application definition.

> Applications -> Application Definition -> Edit or Create new application -> Activity Data Sources -> New Activity Data Source (or click existing) -> Transformation Rule

The reference to the rule is recorded in the ActivityDataSource XML, which is in turn referenced by the Application XML (in its <ActivityDataSources> element).

```
<TransformationRule>
    <Reference class="sailpoint.object.Rule" id="ff8080813b712ea0013b71c4d873004f"
name="testActivityTransform"/>
</TransformationRule>
```

Arguments

Inputs (in addition to the common arguments): Variable based on the collector type

The JDBC Collector passes in a nested hashmap structure called rowColumns and an ApplicationActivity object called activity.

Argument	Type	Purpose
rowColumns	java.util.HashMap	A hashmap of column names and values
activity	sailpoint.object.ApplicationActivity	Reference to an ApplicationActivity object that has been

		partially completed from key values in the record

The LogFile Collector passes in each attribute as a separate name, value pair. The names of the arguments are completely dependent on the column names in the log file; all are added as strings. It also passes an ApplicationActivity object to the rule. An example argument list could look like this:

Example Argument	Type	Purpose
username	String	Name of the user performing the activity
actionDate	string	Date the activity occurred
…		Other fields here
activity	sailpoint.object.App licationActivity	Reference to an ApplicationActivity object that has been partially completed from key values in the log file record

The Windows Event Log Collector passes these arguments to the rule:

Argument	Type	Purpose
datasource	Sailpoint.object.ActivityD ataSource	A reference to the ActivityDataSource object that defines the source from which the application's activity data is read
event	sailpoint.object.Windows EventLogEntry	A reference to the WindowsEventLogEntry object that represents an entry in the Windows Event Log

The RACF Activity Collector does not use this rule.

NOTE: Because of the variable nature of the arguments to this rule type, these are good candidates for using the procedures outlined in *Printing the Beanshell Namespace* to understand all of the available parameters in each rule.

Outputs:

Argument	Type	Purpose
activity	ApplicationActivity	The ApplicationActivity object that represents the activity

Example

This example activityTransformer rule reads the individual parameters passed to the rule and builds them into an ApplicationActivity object. This example relates to a Log File Collector that has sent this set of arguments:

Argument	Description
ActionDateTime	Date/Time the activity occurred
SystemName	The computer on which the activity occurred
UserID	The username of the account performing the action
Status	Completion status of the action
activity	Reference to an ApplicationActivity object that has been partially completed from key values in the log file record

```
import sailpoint.object.ApplicationActivity.Action;
import sailpoint.object.ApplicationActivity.Result;

activity.setTimeStamp(ActionDateTime);
activity.setTarget(SystemName);
activity.setAction(Action.Create);
activity.setUser(SystemName + "\\" + UserID);

if ( "Complete".equals(Status) ) {
    activity.setResult(Result.Success);
} else if ("Error".equals(Status) ) {
    activity.setResult(Result.Failure);
}

return activity;
```

ActivityCorrelation

Description

This rule is used to correlate the activity record to a user in IdentityIQ; in other words, this rule associates the activity record to the Identity who performed the activity.

Definition and Storage Location

The ActivityCorrelation rule is defined in the UI within the application definition.

> Applications -> Application Definition -> Create new or select existing application -> Activity Data Sources -> New Activity Data Source (or edit existing) -> Correlation Rule

The rule is referenced as the CorrelationRule in the ActivityDataSource XML.

```
<CorrelationRule>
    <Reference class="sailpoint.object.Rule" id="402846023a65e596013a65e7860704d6"
name="Example Activity Correlation Rule"/>
</CorrelationRule>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
application	sailpoint.object.Application	Reference to the application on which the activity occurred
datasource	sailpoint.object.ActivityDataSource	Reference to the datasource from which the activity was collected
activity	sailpoint.object.ApplicationActivity	Reference to the ApplicationActivity object representing the collected activity record

Outputs:

Argument	Type	Purpose
result	java.util.Map or sailpoint.object.Link	A map that provides identifying information from which IdentityIQ can select the appropriate link and ultimately the appropriate Identity. The map will contain any of these key/value sets: Key/Values for correlating to a Link: • "linkIdentity", nativeIdentity value or • "linkDisplayName", displayName value or • "linkAttributeName", identifying Link attribute and "linkAttributeValue", Link attribute value NOTE: When correlating to a link through the map attributes, if the link is on an application defined with instances, a "linkInstance" attribute specifying the instance name should also be included in the map Key/Values for correlating to an Identity: • "identityAttributeName", identifying Identity attribute for the Identity and "identityAttributeValue", Identity attribute value • "identityName", Identity name • "identity", Identity object Alternatively the rule can return a Link object.

Example

This ActivityCorrelation rule indicates that the name in the activity object's "user" attribute matches to the samAccountName attribute on the link to which the activity should be correlated.

```
import sailpoint.object.ApplicationActivity;

String user = activity.getUser();
Map returnMap = new HashMap();
if ( user != null ) {
    returnMap.put("linkAttributeName", "samAccountName");
    returnMap.put("linkAttributeValue", user);
}
return returnMap;
```

ActivityPositionBuilder

Description

The ActivityPositionBuilder rule applies only to JDBC Activity Collectors. It runs at the end of the activity-data-gathering process and uses the current position in the activity collector resultSet to build a Map<String,String> that can be saved to the SailPoint database. This map will be retrieved and passed to the ActivityConditionBuilder rule to build the where clause in the next incremental call to this collector. These two

rules, together, function as a placeholder for activity collection to identify which records in the datasource have already been collected by IdentityIQ and which still need to be read.

Definition and Storage Location

The ActivityPositionBuilder rule can be associated to an application on the Activity Data Sources tab within the application definition. There is no rule editor available on this page, however, so the rule must be written in XML and imported into IdentityIQ.

> Applications -> Application Definition -> Create new or select existing application -> Activity Data Sources -> New activity data source (or edit existing) -> Activity Data Source Type: JDBC Collector -> Query Settings -> Position Builder.

The rule name is recorded as the value for the positionConfigBuilderRule in the attributes map of the ActivityDataSource XML.

```
<entry key="positionConfigBuilderRule" value="[ActivityPositionBuilder Rule Name]"/>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
row	java.sql.ResultSet	Current position in the result set

Outputs:

Argument	Type	Purpose
returnMap	java.util.Map	Map that can be used to build the where clause for the next call to the activity collector

Example

This example rule gets the timestamp for the request from the resultSet and records it in the returnMap. When this map is fed into the ActivityConditionBuilder rule in the next activity collection, only activity that has occurred after that date-time will be collected.

```
import java.util.Map;
import java.util.HashMap;

Map returnMap = new HashMap();

String lastTimeStamp = row.getString("REQUEST_TIME");
returnMap.put("lastTimeStamp", lastTimeStamp);

return returnMap;
```

ActivityConditionBuilder

Description

The ActivityConditionBuilder rule works in conjunction with the ActivityPositionBuilder rule, as described above. It, too, only applies to JDBC Activity Collectors. It uses the map created by the ActivityPositionBuilder rule to build the where clause for retrieving the next set of activity data from the data source.

This rule is only applied if the sql statement that defines where and how activity data is read includes a reference variable $(positionCondition). The ActivityConditionBuilder rule specifies the condition that is substituted for that variable.

Definition and Storage Location

The ActivityConditionBuilder rule can be associated to an application on the Activity Data Sources tab within the application definition. There is no rule editor available on this page, however, so the rule must be written in XML and imported into IdentityIQ.

> Applications -> Application Definition -> Create new or select existing application -> Activity Data Sources -> New activity data source (or edit existing) -> Activity Data Source Type: JDBC Collector -> Query Settings -> Condition Builder.
>
> … Query Settings -> SQL Statement example: Select * from activity where $(positionCondition)

The rule name is recorded as the value for the conditionBuilderRule in the attributes map of the ActivityDataSource XML.

```
<entry key="conditionBuilderRule" value="[ActivityConditionBuilder Rule Name]"/>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
config	java.util.Map	Map of values that can be used to identify where in the datasource the collector should resume activity collection

Outputs:

Argument	Type	Purpose
condition	string	String value for where clause in activity lookup

Example

This example activityConditionBuilder rule uses the lastTimeStamp value recorded in the config map (by the ActivityPositionBuilder rule) to determine the earliest request date to retrieve from the activity source to update activity data in IdentityIQ.

```
String condition = "";
```

```
String lastTimeStamp = (String)config.get("lastTimeStamp");
if ( lastTimeStamp != null )
    condition = "REQUEST_TIME &gt; '" + lastTimeStamp + "'";

return condition;
```

Report Rules

This section describes rules that relate to running or configuring reports in IdentityIQ.

ReportCustomizer

Description

A ReportCustomizer rule runs during report-UI rendering to create a dynamically-built form for report-filter specification. Out of the box, many of SailPoint's out of the box reports allow filtering based on any editable identity attribute, including extended attributes. Since extended attributes are customer-defined, the form for specifying these attributes as filters necessarily cannot be shipped complete and therefore must be built dynamically.

Definition and Storage Location

The rule is specified in the report's TaskDefinition XML as part of the LiveReport definition.

```
<LiveReport>
…
  <InitializationRule>
    <Reference class="sailpoint.object.Rule" name="Identity Report Form Customizer"/>
  </InitializationRule>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
taskDefinition	sailpoint.object.TaskDefinition	Reference to the taskDefinition object that contains the report definition
report	sailpoint.object.LiveReport	The report definition itself
locale	java.util.Locale	Represents a specific geographical, political, or cultural region; used for locale-specific rendering/calculations (e.g. number formatting)

Outputs: None; the rule's logic is should directly perform the necessary updates to objects; in the out of the box examples, the rules add attributes for the ReportingLibrary class to use in rendering the filter form.

NOTE: The signature of this rule in the rules that ship with IdentityIQ indicates that it returns a Map, but that is not so. No returned value from this rule will be processed.

Example

This example rule is the Identity Report Form Customizer rule that ships with the product and runs as part of initialization of several of the out of the box reports. It adds all searchable attributes to the ReportingLibrary for use in rendering the report filter form.

```
import sailpoint.object.*;
import sailpoint.reporting.ReportingLibrary;

ObjectConfig identityConfig = ObjectConfig.getObjectConfig(Identity.class);
List standardAttributes = new ArrayList();

standardAttributes.add(identityConfig.getObjectAttributeMap().get("firstname"));
standardAttributes.add(identityConfig.getObjectAttributeMap().get("lastname"));
standardAttributes.add(identityConfig.getObjectAttributeMap().get("displayName"));
standardAttributes.add(identityConfig.getObjectAttributeMap().get("email"));
standardAttributes.add(identityConfig.getObjectAttributeMap().get("manager"));
standardAttributes.add(identityConfig.getObjectAttributeMap().get("inactive"));

ReportingLibrary.addAttributes(context, report, IdentityEntitlement.class,
standardAttributes, "identity","Identity Attributes", locale, "id");

List extendedAttrs = new ArrayList();
for(ObjectAttribute att : identityConfig.getSearchableAttributes()){
    if (!att.isStandard())
        extendedAttrs.add(att);
}

for(ObjectAttribute att : identityConfig.getMultiAttributeList()){
    extendedAttrs.add(att);
}

ReportingLibrary.addAttributes(context, report, IdentityEntitlement.class,
extendedAttrs, "identity","Identity Extended Attributes", locale, "id");
```

ReportValidation

Description

A ReportValidation rule is used to perform report-specific form validation in case where the single field validation option available in forms does not meet the needs of the report. This can be specified as a rule (ValidationRule) or as a script (ValidationScript) embedded within the LiveReport definition.

Definition and Storage Location

The rule is specified in the report's TaskDefinition XML as part of the LiveReport definition.

```
<LiveReport>
…
  <ValidationRule>
    <Reference class="sailpoint.object.Rule" name="Report Validation Rule Name"/>
  </ValidationRule>
…
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
report	sailpoint.object.LiveReport	Report definition being validated
form	sailpoint.object.Form	Form for rendering parameter UI, requiring validation of user input
locale	java.util.Locale	Represents a specific geographical, political, or cultural region; used for locale-specific rendering/calculations (e.g. number formatting)

Output:

Argument	Type	Purpose
errors	List<Message>	Returns a List of messages indicating where/why validation has failed; list should be returned as null or empty if the form entries pass validation

Example

This example validation rule checks that an email file format is specified when email recipients are specified for the report. This validation rule could actually be specified as a field validation rule on the file format field or as a validation rule on the report itself. In general, a report-level ValidationRule will only be used when the form definition cannot achieve the required result.

```
import java.util.*;
import sailpoint.tools.Util;
import sailpoint.web.messages.MessageKeys;

List errors = null;
List emailRecips = form.getField("emailIdentities").getValue();
List fileTypes = form.getField("emailFileFormat").getValue();

if (!Util.isEmpty(emailRecips) && Util.isEmpty(fileTypes)){
    errors = new ArrayList();
    errors.add(MessageKeys.REPT_FORM_ERR_NO_EMAIL_FORMAT);
}

return errors;
```

ReportParameterQuery

Description

A ReportParameterQuery rule is used to specify any custom filter for the report and add it into the queryOptions object that is used in the datasource filter. This is an alternative to a QueryScript (embedded within the report definition). Parameters using a QueryRule do not need to specify a property because the queryRule overrides

any property on the parameter; the argument specified on the parameter can be accessed within the rule through the "value" variable.

Definition and Storage Location

The rule is specified in the report's TaskDefinition XML as part of a parameter definition in the LiveReport.

```
<Parameter argument="groupDefinitions">
  <QueryRule>
    <Reference class="sailpoint.object.Rule" name="Group Query Rule"/>
  </QueryRule>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
value	java.lang.Object	Parameter value, as specified in report form
arguments	sailpoint.object.Attributes	Report arguments map
queryOptions	sailpoint.object.QueryOptions	Current QueryOptions object for report filter

Output:

Argument	Type	Purpose
queryOptions	sailpoint.object.QueryOptions	Updated QueryOptions object to use as report filter

Example

Group and populations are stored in groupDefinitions objects as a filter, so this example shows how a group or population selected as a report parameter can be built into the datasource filter. This example comes from a QueryScript in the Identity Forwarding Report, but extracting that script into a rule object would turn this into a QueryRule.

```
import sailpoint.object.*;
import sailpoint.reporting.*;

Filter f = ReportingLibrary.getGroupDefinitionFilter(context, value, false);
if (f != null) {
    queryOptions.addFilter(f);
}
return queryOptions;
```

ReportParameterValue

Description

A ReportParameterValue rule is processed as "property = return value from ValueRule". It performs processing based on the argument's value to return a different value that should be used in the criterion. In a ValueRule,

the argument is accessed through the variable name "value". This is an alternative to a ValueScript embedded within the report definition.

Definition and Storage Location

The rule is specified in the report's TaskDefinition XML as part of the LiveReport definition.

```
<Parameter argument="applications" property="parent.application">
   <ValueRule>
       <Reference class="sailpoint.object.Rule" name="App Value Rule"/>
   </ValueRule>
</Parameter>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
value	java.lang.Object	Parameter value, as specified in report form
arguments	sailpoint.object.Attributes	Report arguments map

Output:

Argument	Type	Purpose
result	java.lang.Object	Value or list of values to use in query with the specified parameter

Example

This example rule retrieves the application name that corresponds to the applicationID in the "application" report argument. This would be the App Value Rule referenced in the example Definition and Storage Location above.

```
import sailpoint.object.*;
import sailpoint.api.ObjectUtil;
if (value != null){
    return ObjectUtil.convertIdsToNames(context, Application.class, value);
}
return null;
```

Miscellaneous Rules

This section is a catch-all grouping for rules that do not fall into any of the other rule categories discussed in this document.

RiskScore

Description

When risk is associated with specific Identity attributes, custom risk score components can be created to reflect that risk in the identity risk scores. Scoring for these Identity-attribute-based components can either be based on a score assigned to a particular Identity attribute value or be calculated through a RiskScore rule. The resultant score is factored into the identity risk score based on the "weight" assigned to the component in the composite score for the Identity.

Definition and Storage Location

The custom risk score component must be added to the ScoreConfig XML as a new ScoreDefinition element within the <IdentityScores> element; this is how it is added into the identity risk score computation. Depending on how it is specified, there may or may not be a UI component for specifying scorer details. To specify it without creating an interface window, record the ScoreDefinition like this:

```
<ScoreDefinition component="true" displayName="Accounting User Risk Score"
name="acctUser" scorer="sailpoint.score.IdentityAttributeScorer"
shortName="AccountingUser" weight="10">
   <Attributes>
     <Map>
       <entry key="rule" value="Accounting Risk Score Rule"/>
     </Map>
   </Attributes>
</ScoreDefinition>
```

This adds Accounting User Risk Score to the Composite Scoring page for Identity Risk Score (visible via **Identities -> Identity Risk Model** -> **Composite Scoring**) and allows its score weight to be adjusted through the UI but does not allow the rule to be changed or any other basis for score calculation to be specified through the UI. To expose those additional configuration options in the UI, the ScoreDefinition must include a configPage (gotoCustomScorePage is the default configuration page provided with IdentityIQ), a set of input arguments in a signature element (determines the fields displayed on the default configuration page), and an attributes map (prepopulates fields with values).

```
<ScoreDefinition component="true" configPage="gotoCustomScorePage"
displayName="Inactive User Score" name="inactiveUser"
scorer="sailpoint.score.IdentityAttributeScorer" shortName="Inactive" weight="25">
   <Attributes>
     <Map>
       <entry key="attribute" value="inactive"/>
       <entry key="score" value="500"/>
       <entry key="value" value="true"/>
       <entry key="rule" value="Inactive User Scoring Rule"/>
     </Map>
   </Attributes>
   <Description>This is a custom scorer. It looks for inactive users, and if an
identity is found to be inactive we assign the risk score specified for this score
component.  If a rule is specified, the attribute-score-value combination will be
ignored in favor of the rule. </Description>
   <Signature>
     <Inputs>
       <Argument helpKey="help_risk_custom_attribute" name="attribute" type="string">
         <Prompt>Attribute name:</Prompt>
```

```
      </Argument>
      <Argument helpKey="help_risk_custom_value" name="value" type="string">
        <Prompt>Attribute value:</Prompt>
      </Argument>
      <Argument helpKey="help_risk_custom_score" name="score" type="int">
        <Prompt>Risk Score:</Prompt>
      </Argument>
      <Argument helpKey="Rule to control scoring" name="rule" type="string">
        <Prompt>Scorer Rule:</Prompt>
      </Argument>
    </Inputs>
  </Signature>
</ScoreDefinition>
```

As before, the weight assigned to this component in the composite score is still modifiable in the UI on the Composite Score page. When specified this way, the rule name can be modified from the UI within the Identity risk model definition.

> Identities -> Identity Risk Model -> Composite Scoring -> click the scoring component name

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
identity	sailpoint.object.Identity	Reference to the Identity being scored

Outputs:

Argument	Type	Purpose
score	Integer or string	Value of the risk score to assign for the Identity Attribute score

Example

This example RiskScore rule assigns a 500 score for this identity attribute component score to any Identity who works in the Accounting or Finance department.

```
if ("Accounting".equals(identity.getAttribute("department") ||
    "Finance".equals(identity.getAttribute("department")) {
    return 500;
}
return 0;
```

RequestObjectSelector

Description

The RequestObjectSelector rule specifies a filter that is used in determining the objects that a given user can request for the population of users over which he has request authority in the Lifecycle Manager component of

IdentityIQ. These are specified as the "Object Request Authority" rules that determine the list of Roles, Applications, and Managed Entitlements visible to a user in the LCM access request windows.

The scopeService class offers convenience methods for creating QueryOptions objects that filter the object lists by matching an Identity's assigned scope or controlled scopes. These are accessible by to rules that import the sailpoint.api.ScopeService class. The methods are:

```
QueryInfo getAssignedScopeQueryInfo(Identity)
QueryInfo getControlledScopesQueryInfo(Identity)
```

Definition and Storage Location

RequestObjectSelector rules can be selected and specified through the Lifecycle Manager Configuration page in System Setup.

> System Setup -> Lifecycle Manager Configuration -> Object Request Authority section under any of the four request categories (Self Service, Managers, Help Desk, All Users) -> Roles, Applications, or Managed Entitlements

The rules are recorded in the System Configuration XML in an attributes map belonging to one of these entries (entry designates the request category to which the object request authority rules apply):

```
<entry key="selfServiceRequestControls">
<entry key="managerRequestControls">
<entry key="helpDeskRequestControls">
<entry key="generalPopulationRequestControls">
```

They are noted by rule ID, rather than rule name, and appear like this:

```
<entry key="helpDeskRequestControls">
   <value>
     <Map>
       …
       <entry key="applicationSelectorRule" value="402846023a65e596013a65e5d4ae0133"/>
       <entry key="managedAttributeSelectorRule"
value="402846023a65e596013a65e5d6b10137"/>
       <entry key="roleSelectorRule" value="402846023a65e596013a65e5d4ae0133"/>
     </Map>
   </value>
</entry>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
requestor	sailpoint.object.Identity	Identity object for the user who is making LCM access requests
requestee	sailpoint.object.Identity	Identity object representing the user on whose behalf the request is being made (the person to whom the requested access will be granted); only applicable when the request is being processed for a single user; null when multiple users are specified

Outputs:

Argument	Type	Purpose
filter	sailpoint.object.QueryInfo	The QueryInfo object containing the filter to be applied to the object list

NOTE: Older versions of this rule may return a Filter object instead of a QueryInfo object and IdentityIQ can handle this return value. In some cases, multiple object request authority rules may apply to a given user (e.g. if one rule is set for Managers and another for Help Desk and the Identity in question falls into both categories). By default, all of these filters are or'd together so the least restrictive gets applied, but a null filter would just be ignored. By returning a QueryInfo object, the rule can specify that when the filter is null, the Identity should be able to see *all* objects of the given type, regardless of any other applicable filters – in effect, overriding any other filters. Returning a Filter object does not permit this option.

Example

This example RequestObjectSelector rule returns a filter that restricts the set of objects to those with the same scope as the requestee's assigned scope. It further restricts the set of objects to those with the custom attribute "requestable" set to true.

```
import sailpoint.api.ScopeService;
import sailpoint.object.Identity;
import sailpoint.object.Scope;
import sailpoint.object.QueryOptions;
import sailpoint.object.QueryInfo;
import sailpoint.object.Filter;

ScopeService scopeService = new ScopeService(context);
QueryInfo scopeQueryInfo;
if (requestee == null) {
    scopeQueryInfo = new QueryInfo(new QueryOptions());
} else {
    scopeQueryInfo = scopeService.getAssignedScopeQueryInfo(requestee);
}
Filter requestable = Filter.eq("requestable",true);
Filter assignedScope = scopeQueryInfo.getFilter();
Filter f = Filter.and(requestable, assignedScope);

QueryInfo finalQueryInfo = new QueryInfo(f, false);

return finalQueryInfo;
```

This example RequestObjectSelector rule gives the Identity access to all objects of the applicable type, regardless of any other request authority filters that might apply to the user. For example, if all Managers have access to all Roles and a user falls under the Manager and Help Desk categories, this rule, if connected to the Managers object request authority settings for LCM, forces an override of whatever Help Desk filters would be applied and grants the user access to all Roles.

```
import sailpoint.object.QueryInfo;

QueryInfo scopeQueryInfo = new QueryInfo(null,false);
```

```
       return scopeQueryInfo;
```

TaskEventRule

Description

The TaskEventRule is a rule type created in IdentityIQ 6.0. It is used to inject logic at a particular stage in the Task execution process; currently the only stage supported is task completion. This rule type was created to allow reporting tasks to notify the requesting user when the report has been completed.   When the user clicks **Email Me When Done** on the Task Result, a TaskEvent is created with an attached TaskEventRule that sends an email message to the requester when the task reaches the completion stage.

NOTE: Because custom tasks do not modify the Task Result UI and TaskEvents can only be created with a connection to an in-progress TaskResult, this rule type is not currently useful for custom coding.

Definition and Storage Location

At present, there is a single TaskEventRule active in IdentityIQ and it is connected to a TaskEvent when that event is created by the IdentityIQ reporting API. The TaskEventRule name is hard-coded there and there is no UI option for creating any other type of TaskEvent or TaskEventRule.

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
taskResult	sailpoint.object.TaskResult	Reference to the current taskResult object from the task execution
event	sailpoint.object.TaskEvent	TaskEvent object to which the rule is connected

Outputs:

Argument	Type	Purpose
newTaskResult	java.util.Map	Contains key "taskResult" and a taskResult object modified by the rule (or null if no update to taskResult is required as a result of the rule's execution)

Example

This is the TaskEventRule defined in IdentityIQ 6.0+ to send an email to a report requester when the report task reaches the Completion stage.

```
import sailpoint.object.*;
import java.util.*;

String identity = (String)event.getAttribute(TaskEvent.ATTR_EMAIL_RECIP);
if (identity != null){
```

```
    Identity identity = context.getObjectByName(Identity.class, identity);
    if (identity == null)
        return result;

    List emailAddresses = new ArrayList();
    emailAddresses.add(identity.getEmail());
    EmailOptions options = new EmailOptions(emailAddresses, null);
    options.setSendImmediate(true);

    Map emailVars = new HashMap();
    emailVars.put("reportName", taskResult.getName());
    options.setVariables(emailVars);

    String templateName =
(String)context.getConfiguration().get(Configuration.REPORT_COMPLETION_EMAIL_TEMPLATE)
;
    EmailTemplate et = context.getObjectByName(EmailTemplate.class, templateName);

    context.sendEmailNotification(et, options);
}
return null;
```

TaskCompletion

Description

The TaskCompletion rule is a rule type created in IdentityIQ 6.3 to support sending an email message to a specified recipient when task execution completes (either in all cases or with an error condition or a warning condition). Task notification is a new feature in version 6.3, and the logic for handling the notification resides in the Task Completion rule specified for the installation. A default rule, called Task Completion Email Rule, ships with the product and contains all the logic necessary to send an email to the recipient designated in the UI, using the email template specified in the UI. Task completion notification can be configured at the task level or at the system level; the task-level configuration takes precedence over the system-level configuration.

NOTE: Many customers will never change this rule or create another rule of this type. Only one can be used in each installation, and the provided rule contains the logic most customers will want to use to manage task completion notifications.

Definition and Storage Location

The rule to run on task completion is specified in the SystemConfiguration XML as the taskCompletionRule. This is an XML-only configuration (no UI component).

```
<entry key="taskCompletionRule" value="Task Completion Email Rule"/>
```

Arguments

Inputs (in addition to the common arguments):

Argument	Type	Purpose
result	sailpoint.object.TaskResult	Reference to the taskResult object from the current task execution

Outputs: None; the rule's logic is intended to send an email notification and return nothing to the system.

Example

The source for the default TaskCompletion rule is included below. That rule contains a main section of code (at the bottom) which executes other methods also defined within the rule source. It determines whether email notification has been configured for the task (for the type of result returned: success, failure, or warning) and, if so, sends an email message to the chosen recipient using the chosen email template.

```java
import java.util.*;
import sailpoint.tools.Util;
import sailpoint.tools.GeneralException;
import sailpoint.object.Configuration;
import sailpoint.object.EmailOptions;
import sailpoint.object.EmailTemplate;
import sailpoint.object.TaskResult;
import sailpoint.object.Identity;
import sailpoint.object.TaskDefinition;
import sailpoint.api.MessageRepository;
import sailpoint.api.Emailer;
import sailpoint.api.BasicMessageRepository;
import sailpoint.api.ObjectUtil;
import sailpoint.api.SailPointContext;

public Boolean sendEmailNotify = false;
public Boolean taskLevelEnabled = false;
public Boolean systemLevelEnabled = false;
MessageRepository _errorHandler;

/**
 * Method to send email
 */
private void sendEmailOnTaskCompletion(String emailTemplate, ArrayList recipients,
TaskResult result, SailPointContext context) {
    String message = "";
    String status = "";
    TaskDefinition def;
    Configuration sysConfig;

    def = result.getDefinition();
    EmailTemplate notifyEmail =  context.getObjectByName(EmailTemplate.class,
emailTemplate);
    if (null == notifyEmail) {
        log.error ("From Task Completion Email Rule: ERROR: could not find email
template [ " + emailTemplate + "]");
        return;
    }
    notifyEmail = (EmailTemplate) notifyEmail.deepCopy(context);
    if (null == notifyEmail) {
        log.error ("From Task Completion Email Rule: ERROR: failed to deepCopy template
[ " + emailTemplate + "]");
        return;
    }
    // For now, we'll just use a map with a few pre-selected properties.
    Map mArgs = new HashMap();

    mArgs.put("taskResult", result);
    mArgs.put("taskName", def.getName());
    mArgs.put("taskDesc", def.getDescription());
    if (result.isError()) {
        status = "Error";
```

```
    }
    else  if (result.isWarning()) {
        status = "Warning";
    }
    else if (result.isSuccess()) {
        status = "Success";
    }

    mArgs.put("taskStartTime", result.getLaunched() );
    mArgs.put("taskEndTime", result.getCompleted() );
    mArgs.put("status", status);
    if (result.getMessages() != null) {
        mArgs.put("message", result.getMessages());
    }
    mArgs.put ("resultId", result.getId());

    EmailOptions ops = new EmailOptions(recipients, mArgs);
    new Emailer(context, _errorHandler).sendEmailNotification(notifyEmail , ops);
}

private Boolean isEmailNotificationEnabled(TaskResult result, SailPointContext
context) {
    String notifyStr = null;
    Boolean sendEmail = false;
    TaskDefinition def;
    Configuration sysConfig;

    def = result.getDefinition();

    notifyStr = (String) def.getArgument (Configuration.ATT_EMAIL_NOTIFY);
    // if it is disabled at Task level, chk for system level settings
    if (notifyStr == null || (notifyStr.equals("Disabled")) ) {
        sysConfig = context.getConfiguration();
        notifyStr = sysConfig.getString(Configuration.ATT_EMAIL_NOTIFY);
        if (notifyStr == null || (notifyStr.equals("Disabled"))) {
            sendEmail = false;
            return (sendEmail);
        }
        else {
            systemLevelEnabled = true;
        }
    }
    else
    {
        taskLevelEnabled = true;
    }

    if (notifyStr.equals("Always")) {
        sendEmail = true;
    }

    if(((notifyStr.equals("Failure")) &amp;&amp; result.isError() == true) ||
        ((notifyStr.equals("Warning")) &amp;&amp; result.isWarning() == true
&amp;&amp; result.isError() == false)) {
        sendEmail = true;
    }
    return (sendEmail);
}

private List getEmailAddress (String identityName, SailPointContext context) {
    Identity identity = context.getObjectByName(Identity.class, identityName);
    if (identity != null)
    {
```

```
        List addresses = ObjectUtil.getEffectiveEmails(context, identity);
        if (!Util.isEmpty(addresses)) {
            return(addresses);
        }
        else
        {
           if(log.isWarnEnabled()) {
              log.warn("From Task Completion Email Rule: Missing Email Address for
Email Recipient: " + identityName );
           }
        }
     }
     return (null);
}

private ArrayList getEmailRecipient (Object identityNames, SailPointContext context) {
    List recipients;
    String val = null;
    StringTokenizer st = null;
    if (identityNames != null) {
        recipients = new ArrayList ();
        // From Task definition, single identity
        if (identityNames instanceof String  &amp;&amp; !identityNames.contains(","))
{
            List addresses = getEmailAddress (identityNames.toString(), context);
            if (addresses != null) {
                recipients.addAll (addresses);
            }
        }
        // From Task definition, multiple identities
        else if (identityNames instanceof String  &amp;&amp;
identityNames.contains(",") == true) {
            List nameList = Util.csvToList(identityNames);
            for (String identityName : nameList) {
                List addresses = getEmailAddress (identityName, context);
                if (addresses != null) {
                    recipients.addAll (addresses);
                }
            }
        }
        // From system configuration single or multiple identities it comes as list
        else if (identityNames instanceof List) {
            for (String identityName : identityNames) {
                List addresses = getEmailAddress (identityName, context);
                if (addresses != null) {
                    recipients.addAll(getEmailAddress (identityName, context));
                }
            }
        }
    }
    return (recipients);
}

// Main
String emailTemplate = "";
TaskDefinition def;
Configuration sysConfig;
Object identityNames;
sendEmailNotify = isEmailNotificationEnabled (result, context);

if (sendEmailNotify) {
    _errorHandler = new BasicMessageRepository();
    if (taskLevelEnabled) {
```

```
        //take template and recipient from task level settings
        def = result.getDefinition();
        Map mArgs = def.getEffectiveArguments();
        identityNames = mArgs.get(Configuration.ATT_IDENTITIES);
        emailTemplate = mArgs.get(Configuration.ATT_EMAIL_TEMPLATE);
    }
    else if (systemLevelEnabled) {
        //take template and recipient from system level settings
        sysConfig = context.getConfiguration();
        emailTemplate = sysConfig.getString(Configuration.ATT_EMAIL_TEMPLATE);
        identityNames = sysConfig.get(Configuration.ATT_IDENTITIES);
    }
    List recipients = getEmailRecipient(identityNames,context);
    if (recipients != null &amp;&amp; !Util.isEmpty(recipients)) {
        // Send Email
        sendEmailOnTaskCompletion(emailTemplate, recipients, result, context);
    }
    else {
        if(log.isWarnEnabled()) {
            log.warn("From Task Completion Email Rule: Cannot send task completion email
Notification. Reason : Missing Email Address for Email Recipients");
        }
    }
}
```

Non-Standard Rules

There are two additional sets of items that are stored in IdentityIQ as Rule objects but are not traditional rules as described in this document. These are rule libraries and Before and After Scripts for direct connectors.

Rule Libraries

Rule libraries are collections of methods that have been grouped together and stored in IdentityIQ as a Rule object. They contain sets of related but unconnected methods that can be invoked directly by workflow steps or other rules. These are stored as Rule objects, rather than in the compiled Java classes, so that their functionality can be easily modified to suit the needs of each installation.

IdentityIQ ships with a few rule libraries that are used by the default workflows. Examples of rule libraries are Workflow Library, Approval Library, and LCM Workflow Library, any of which can be viewed through the debug pages or the IIQ console. Customers can create their own custom libraries to provide additional functionality as needed.

To reference a rule library from another rule, include a <ReferencedRules> element in the rule XML, naming the rule library in the <Reference>. The methods within the library can then be invoked from within the rule's Source element.

```
<Rule…>
    <ReferencedRules>
        <Reference class='Rule' name='My Library'/>
    </ReferencedRules>
    <Source>
        doSomething();  //invokes the doSomething method in My Library
    </Source>
</Rule>
```

Refer to the Workflows chapter of the IdentityIQ Administration Guide for details on how to reference Rule Libraries in workflow steps.

Before/After Scripts

Before and After Scripts, also called Native Rules, are scripts that are sent through the connector to the IQService host machine to run before and after provisioning. Before Scripts can modify the request object (containing the provisioning request) and After Scripts can modify the result object (containing the provisioning result); both can perform custom actions or manipulations on those objects. Scripts can be written in any scripting language, including both object-oriented languages like PowerShell and non-object-oriented languages like Perl.

Native Rules are recorded as Rule objects in IdentityIQ and are assigned a type value that determines their usage.

- Before Script Rule Types: ConnectorBeforeCreate, ConnectorBeforeModify, ConnectorBeforeDelete
- After Script Rule Types: ConnectorAfterCreate, ConnectorAfterModify, ConnectorAfterDelete

The rule names are included in the attributes map of the Application XML for the application to which they apply; they are listed within the "nativeRules" entry.

```
<entry key="nativeRules">
   <value>
      <List>
         <String>AfterCreate-Powershell</String>
         <String>BeforeCreate-Powershell</String>
         <String>BeforeModify-Batch</String>
      </List>
   </value>
</entry>
```

Refer to the IQService Before/After Scripts section (in the Appendix) of the Sailpoint IdentityIQ Direct Connector Administration and Configuration Guide, which ships with IdentityIQ (versions 6.0+), for the complete documentation on Native Rules.

Appendix A: Loading Rules

Rules can be loaded into IdentityIQ through the IIQ console or through the **Import From File** option in the Gear menu -> Global Settings menu.

To import a rule from the console:

1. Launch the console by entering "iiq console" at a command prompt in the [IdentityIQ Install Directory]\WEB-INF\bin directory. The ">" prompt indicates that the console is running.

   ```
   C:\IdentityIQ\WEB-INF\bin> iiq console
   >
   ```

2. Use the import command to import the xml file containing the rule or rules.

   ```
   > import myrules.xml
   ```

3. The console lists all the objects in the file as they are imported.

   ```
   > import myrules.xml
   Rule:Test Rule 1
   Rule:Test Rule 2
   >
   ```

To import a rule through the UI:

1. Navigate to Gear menu -> **Global Settings** -> **Import From File**.
2. Click **Browse…**, select a filename from the file system, and click **Open**.
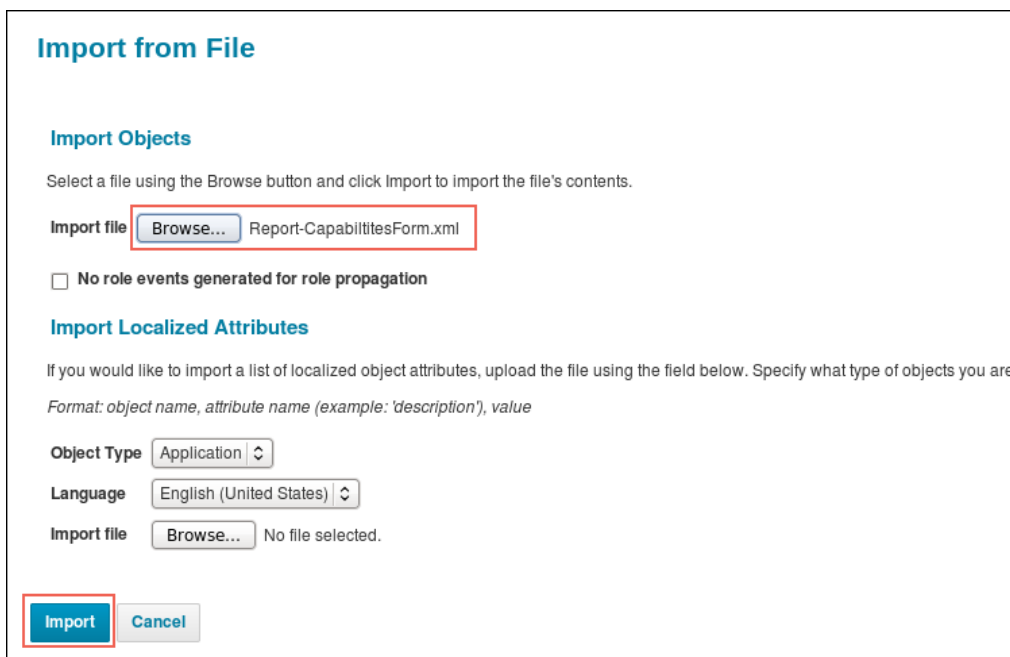3. Click **Import**.

Figure 8: **UI Import From File**