IDENTITYIQ:
RULES, SCRIPTS, AND API
EXERCISES

www.sailpoint.com

Table of Contents

Course Overview

This course teaches how to use the SailPoint API to write rules and scripts in BeanShell to control and constrain IdentityIQ system behaviors to meet business-specific use cases. The exercises guide students through building various rules, some of which are run independent of core processes and some of which are invoked by rule hooks built into the product.

The course is not intended to teach Java programming. It does require you write some BeanShell (Java syntax) but the artifacts are designed to provide enough content and focused guidance so you can practice specific method calls in the IdentityIQ API, rather than general Java programming. The examples are designed to model good practices, though some examples may omit nuances of best practices that are emphasized in later lessons and exercises.

Virtual Machine and Development Environment

The course includes a Skytap-hosted Virtual Machine (VM) which contains an IdentityIQ installation populated with a small set of test data. The VM is set up so you can do the entire set of labs inside the VM.

Exercise Tools

Various steps in the rule/script creation and deployment process will require you to access a web browser (Firefox), a file browser, a terminal window, and a text editor (gedit). Those tools are provided in the training VM and can be accessed from shortcuts in the VM header bar.

VM Credentials

Usage	Username	Password
VM Login	spadmin	admin
IdentityIQ Login	spadmin - or – any other user (first.last)	admin xyzzy
MySQL command line utility	root	root

Exercise Artifacts

The framework for the various rules you will be writing in the course exercises are provided in XML files inside the VM, in the /home/spadmin/APITraining/config folder. In most cases, parts of the logic are written for you, with comments and dashed lines indicating where you should be adding your own code, following the instructions provided in the exercises. Many of the code lines are built as fill-in-the-blank statements to guide your code writing.

Additionally, the /home/spadmin/APITraining/solutions folder contains completed versions of the course artifacts, which can be helpful as a reference, or as an imported artifact, if you get stuck.

Exercise Code Insertion Instructions

The code writing you will do in this course will involve inserting lines into existing BeanShell code. The lines where you need to insert the code are offset with comments like this:

```
// [1] Insert code below
//------------------------------

//------------------------------
```

The code lines may be empty, as shown above, or may be partially complete, with blank lines to indicate where you need to substitute items, like method names, arguments, or whole method calls.

The numbers in the comment, for example [1], are referenced in the exercise instructions. For example, "At [1], add the code …" to help you find the right place for each exercise instruction. Note that these numbers will *not* match the exercise instruction number but will just be sequential with the steps to complete within each code file.

Note on Variable Names

Where subsequent code relies on a specific variable name, the provided code snippets, where you fill in blanks, prescribe the variable for consistency. You may choose different variable names and alter the provided code to match.

Using this Document

Many of the exercises include questions; you should answer these questions to get the maximum learning from the activities. This exercise guide file is an editable PDF so you can record your answers.

The answer key is provided as a separate PDF document. It includes answers to all fill-in-the-blank questions, as well as the code lines to be written into the rules.

Exercise #1: Explore and Execute Rules and Scripts

Objective

Explore several of the rule and script hooks available in IdentityIQ and discover rule execution options.

Overview

In the first two activities of this exercise, you will explore some rule and script hooks in the product and see how rules and scripts are attached to core processes. Then, you'll practice some of the ways to execute rules *outside* the flow of built-in processes, as ad-hoc or scheduled actions.

Explore the Rule Editor

Most rule hooks in IdentityIQ allow you to select the rule to be executed from a dropdown list in the UI. In many cases, the rule can be created, viewed, or modified through the rule editor from the same UI page where the list is presented. Here, you will examine the Rule Editor for a Creation rule.

1. Log in to IdentityIQ as **spadmin / admin**.
2. Navigate to **Applications** - **Application Definition** and choose the **HR System - Employees** application.
3. Click the **Rules** tab.
4. Explore the Creation rule.
    a. Click the ellipses button (…) to the right of the **Creation Rule** to open the rule editor.
    b. Read the rule **Description** and complete this statement.

       This rule runs when a new _____ is created during aggregation.

    c. Look at the **Arguments** list. What variables are passed to this rule?

       _____

       _____

    d. What is this creation rule doing?

       _____

Explore Provisioning Policy Scripts/Rules

BeanShell code can be embedded as scripts inside some other objects, like Role and Application Provisioning Policies, Forms, and Workflows, or they can be created as Rule objects for reference by those objects.

1. Navigate to **Applications** - **Application Definition** and choose the **PRISM** application.
2. Click **Configuration** - **Provisioning Policies** and click the account Create provisioning policy, called **PRISM Account Create**.
3. View and edit the Login ID attribute's details.
    a. Click the pencil icon on the **Login ID** row.
    b. Expand the **Value Settings** section and find the **Value** field.
    c. Is the field's value currently set by a rule or script? _____
    d. Record the field's logic here:

    _____

    e. If you had more than one provisioning policy (such as on a different application) that needed to use this same logic, what would be a better way to store this logic?

    _____

4. Move the Login ID attribute's logic into a Rule.
    a. Select and copy the script contents to the clipboard.
    b. Click the **Value** dropdown and choose **Rule**.
    c. Click the ellipses button (…) to open the rule editor.
    d. Paste the script logic into the editor.
    e. Enter **Rule Name**: Return Identity Name
    f. What rule type is this rule? _____
    g. Click **Save**.
    h. Ensure that your **Return Identity Name** rule is selected in the rule dropdown and click **Apply** at the bottom of the field definition pane.
    i. Scroll to the top of the form editor and click **Save** to save the provisioning policy changes.

j. **Save** the application definition changes.

5. Use Lifecycle Manager to request an entitlement on the PRISM application that will create a new user account, to test your provisioning policy change and ensure it still works as expected.

   a. On the **Home** page, click the **Manage User Access** Quicklink card to initiate an access request through LCM.

   b. Enter the request details:

      i. **Select User**: choose **Amy.Cox**

      ii. **Manage Access** - **Add Access**: search for **PRISM** and choose the **User** entitlement (not the **PRISM User** role) on the PRISM application. Click **Next**.

      iii. **Review and Submit**: click **Submit**

   c. Log in as **Walter.Henderson / xyzzy** and approve the request.

   d. Log in as **spadmin** to verify account creation through IdentityIQ.

      i. Navigate to **Identities** - **Identity Warehouse** and choose **Amy.Cox** from the list.

      ii. Click the **Application Accounts** tab and locate her **PRISM** account.

         (1) Her account name should match her identity name. (Login ID is named as the identity attribute for the PRISM application and therefore sets the account name.)

         (2) Expand the account details and see that she is now in the User group, per your provisioning request.

   e. **Optional**: If you want to verify provisioning by checking the PRISM database itself, take the following steps:

      i. Open a terminal window.

      ii. At the command prompt, enter **mysql -u root -p**

iii. At the password prompt, enter **root**

iv. Enter these SQL commands:

(1) **use prism;**

(2) **select * from users where first="Amy";**

```
mysql> select * from users where first="Amy";
+---------+-------------+-------+------+--------+----------+--------+--------+--
---------+
| login   | description | first | last | groups | password | status | locked | l
astLogin |
+---------+-------------+-------+------+--------+----------+--------+--------+--
---------+
| Amy.Cox | NULL        | Amy   | Cox  | User   | NULL     | A      | N      | N
ULL      |
+---------+-------------+-------+------+--------+----------+--------+--------+--
---------+
1 row in set (0.00 sec)
```

Run a Rule from the IdentityIQ Console

In this activity, you will import and execute a simple rule. The rule accepts an identity name as an argument, looks up that identity, then prints and returns information about that identity.

1. Launch the IIQ Console command line utility using one of the following.

   a. Click the **IIQ Console** desktop shortcut

   OR

   b. Open a terminal window and navigate to **/home/spadmin/tomcat/webapps/identityiq/WEB-INF/bin**, then enter

   ```
   ./iiq console -j
   ```

2. Use the following command in the IdentityIQ Console to import the rule from **Rule-TRNG-IdentityLookup.xml**:

   ```
   import /home/spadmin/APITraining/config/Rule-TRNG-
   IdentityLookup.xml
   ```

   The **import** command reads an XML representation of a SailPoint object from a file into IdentityIQ and creates an object in the database.

3. View the rule in the IdentityIQ Console with this command:

   ```
   get rule TRNG-IdentityLookup
   ```

   The **get** command displays the XML representation of the SailPoint object in the console.

4. Examine the rule's <Source> block.

   a. What attribute values will be printed to Standard Out when the requested identity is found?

   _____

   b. What attribute value will be returned by the rule when the requested identity is found?

   _____

5. This rule requires an argument to specify the name of the identity to look up. When you run a rule from the console, arguments are provided in a file which contains a map. For this exercise, you will use the file LookupRuleArgs.xml.

   a. Examine the **/home/spadmin/APITraining/config/LookupRuleArgs.xml** file to familiarize yourself with its structure.

   b. Run the rule in the IdentityIQ Console with this command:

```
rule TRNG-IdentityLookup
/home/spadmin/APITraining/config/LookupRuleArgs.xml
```

```
> rule TRNG-IdentityLookup /home/spadmin/APITraining/config/LookupRuleArgs.xml
Found the Identity: Name = Adam.Kennedy, Email address = Adam.Kennedy@demoexample.com
Adam.Kennedy@demoexample.com
>
```

Add a Rule Library Function Call to the Rule

1. Import the Rule-TRNG-UtilRuleLibrary.xml file through the IdentityIQ Console:

```
import /home/spadmin/APITraining/config/Rule-TRNG-
UtilRuleLibrary.xml
```

2. View the rule library through the console:

```
get rule TRNG-UtilRuleLibrary
```

3. The rule library contains a method called getManagerEmail(). Find that method and note its signature here.

   _____

4. Examine the reference to the rule library into the TRNG-IdentityLookup rule.

   a. Open the file **/home/spadmin/APITraining/config/Rule-TRNG-IdentityLookup.xml** in a text editor.

b. Review the following ReferencedRules block to the Rule object between the Description and the Signature elements.

```
<ReferencedRules>
  <Reference class='Rule' name='TRNG-UtilRuleLibrary'/>
</ReferencedRules>
```

5. At [1], verify the IdentityLookup rule logic invokes the rule library method and returns the manager's email instead of the user's email.

```
import sailpoint.object.Identity;

Identity identity = context.getObjectByName(Identity.class, config.get("identityName"));
if (null != identity) {
    System.out.println("Found the Identity: Name = " + identity.getName() + ", Email address = " + identity.getEmail());
    // [1] Replace with code to return manager email address
    //--------------------------------------
    return getManagerEmail(identity.getName());
    //--------------------------------------
}
```

6. Re-import the **Rule-TRNG-IdentityLookup.xml** file to update the rule in your IdentityIQ instance.

   NOTE: Alternatively, you could complete steps 4 and 5 in the already-imported rule object through the Debug pages. There is no way to add a ReferencedRules element through the IdentityIQ UI for any rule, as it is an attribute of the object, not of the BeanShell code.

7. Re-run the rule in the IdentityIQ Console to test your change.

   ```
   rule TRNG-IdentityLookup
   /home/spadmin/APITraining/config/LookupRuleArgs.xml
   ```

   TIP: If you remain logged into the IdentityIQ Console, you can use the up arrow to recall previous commands and run them again without retyping them. This is a time-saver in the debugging process.

Create a Run Rule Task to Execute the Rule

1. Navigate to **Setup** - **Tasks**.
2. Click **New Task** and choose the **Run Rule** task type.
3. Configure the task arguments.
   a. **Name**: Run Identity Lookup Rule
   b. **Rule**: TRNG-IdentityLookup
   c. **Rule Config**: identityName,Adam.Kennedy

TIP: The task expects no space after the comma in this RuleConfig string. RuleConfig can either be a comma separated list of name-value pairs or can be specified as a Map. Specifying it as a Map is easier to do in the TaskDefinition's XML representation, since the UI field is a simple text field.

4. Click **Save** to save the task configuration.

5. You need to make an adjustment to the **Run Rule** TaskDefinition object to make the results appear in the Task Results UI. This step is required because the template task omits the required Returns element for displaying its results in the Task Result UI.

   a. Navigate to the Debug page using the **Debug Pages** bookmark in the bookmark toolbar.

   b. Select Object type: **TaskDefinition** and open the **Run Rule** task in the Object Editor.

   c. Add a Returns section to the Signature block, just above the </Signature> line, and save your changes.

```
<Returns>
   <Argument name="Result: " type="string">
      <Prompt>Result</Prompt>
   </Argument>
</Returns>
```

   NOTE: The return argument name needs to match this exactly since the Run Rule task returns a variable named "Result: ".

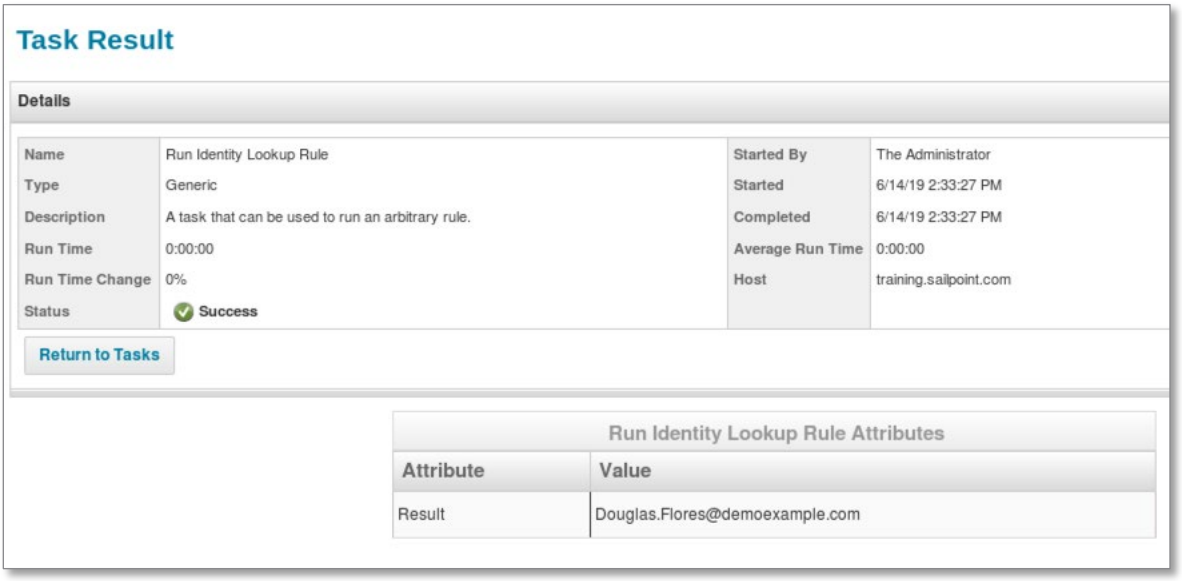
   d. Save the task definition.

6. Run the task.

   a. Navigate to **Setup** - **Tasks**.

   b. Find your **Run Identity Lookup Rule** task. Right-click it and click **Execute in Background**. Click **OK** to acknowledge the task execution.

7. Click the **Task Results** tab on this page. Examine the task results for the **Run Identity Lookup Rule** task. Adam Kennedy's manager's email address should be reflected in the **Result** entry.



NOTE: The print statements in the rule are still written to standard out. For the IdentityIQ Console, the console is its own standard out, but for the IdentityIQ web application, see the application server's standard out: in this case, catalina.out. Use the desktop shortcut **Tail Tomcat Standard Out** to view the end of the catalina.out log file and see these statements.

Tip: Rule Arguments for Console vs. Run Rule Task

Examine the arguments file you used when you ran this rule in the IdentityIQ Console (/home/spadmin/APITraining/config/LookupRuleArgs.xml). Note that it contains a Map of a Map.

This is a slightly more complex structure than is truly required for running a rule with arguments from the console. The console is looking for a simple Map of arguments (name-value pairs in key-value relationships).

However, rules run from the Run Rule task are passed arguments in a single map variable called "config". The rules must therefore be written to extract those arguments from that config map. To reuse a rule that works with the Run Rule task as a rule that you can run from the console, you must provide those arguments in a config map from both places.

```
<!-- For Run Rule Task -->
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE Map PUBLIC "sailpoint.dtd" "sailpoint.dtd">
<Map>
   <entry key="config">
    <value>
     <Map>
       <entry key="identityName" value="Adam.Kennedy"/>
     </Map>
    </value>
   </entry>
</Map>
```

If you create a rule that you will *only* ever be run from the IdentityIQ Console, you can eliminate this extra map layer in both your rule code and your argument file.

```
<!-- For IIQ Console rule execution -->
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE Map PUBLIC "sailpoint.dtd" "sailpoint.dtd">
<Map>
    <entry key="identityName" value="Adam.Kennedy"/>
</Map>
```

Exercise #2: Using the API

Objective

Use the object model to retrieve data from objects and to update the data recorded in them.

Overview

This exercise contains two activities to help you explore the IdentityIQ JavaDoc and practice writing common object method calls.

- You will write a custom rule for an ad-hoc data management operation to populate an extended attribute on Applications. The rule can be run from the console or the debug page.
- You will also modify the HR Employee application's Creation rule to add some conditional logic based on the value of a specific user record attribute.

Populate an Application Extended Attribute

Your installation has specified an extended attribute on the Application object which should be populated based on the owner specified for the application. You will create and execute a rule, which uses various attribute getters and setters, to retrieve the needed value from the owner's identity and persist it to the application attribute.

1. Examine the Application extended attribute definition.
   a. Navigate to the **Gear Menu** - **Global Settings** - **Application Attributes**.
   b. What extended attribute has been defined for Applications?
   (Click the attribute to see the attribute name in addition to its display name.)

   _____

2. Use the text editor to open the file **/home/spadmin/APITraining/config/Rule-TRNG-SetOwnerContact.xml**
3. Read the rule's **Description**. What is its purpose?

   _____

4. Locate the rule's <Source> block. The logic to retrieve and loop through all applications is provided for you. What is the variable name used to store each application, as the code loops through the list of applications?

   _____

5. Open JavaDoc and find **sailpoint.object.Application**. What method is available to return the application owner and what type of return value does it provide?
   HINT: This method is inherited from the SailPointObject class, so it will appear in that list within the Application class JavaDoc.

   _____

6. At [1], add the BeanShell code to retrieve that application owner (and remove [1]). It will be patterned like this:

   ```
   VariableType variableName = appVariable.methodName();
   Example: [VariableType] owner = targetApp.[methodName]();
   ```

7. At [2], retrieve the email address of the application owner. Use JavaDoc for the object type retrieved above to find the right method for that.

8. Now, at [3], add the logic to set the email address into the ownerContact application extended attribute you identified in step 1. Consult the JavaDoc for the syntax for the setAttribute() method signature; this is the method used for setting extended attributes.

   TIP: You'll see a number of specific setter methods like setConnector and setDescription in the JavaDoc, but there is no setOwnerContact method because ownerContact is not part of the IdentityIQ default attribute set. This is why you need the generic attribute setter method.

   TIP: There are also methods called setExtended, setExtended1, setExtended2, and so on, which you can use if you know the extended column number where the attribute is recorded in the database. However, if you ever change your database schema, these could get out of sync.

9. In most cases, the core IdentityIQ processes, like aggregation and certification include save and commit operations for objects that will likely be modified in their rules; rules run at those points do not require code to perform those actions. Because this rule will be run outside of any defined IdentityIQ process, it must explicitly specify the logic to save the object and commit the transaction. At [4], complete the saveObject() logic in your rule to save the application variable you have modified. Then save the file.

```
<Source>
 import sailpoint.object.Identity;
 import sailpoint.object.Application;

 // Retrieve applications
 List applications = context.getObjects(Application.class);

 if (null != applications) {
   // Loop through list of applications
   for (Application targetApp : applications) {
      // Retrieve application owner
      //----------------------------
      Identity owner = targetApp.getOwner();
      //----------------------------
      if (null!= owner) {
         // Retrieve owner's email address
         //----------------------------
         String email = owner.getEmail();
         //----------------------------
         if (null != email) {
             // Set ownerContact attribute for application
             //----------------------------
             targetApp.setAttribute("ownerContact",email);
             //----------------------------
         }
      }
      context.saveObject(targetApp);
      context.commitTransaction();
   } // end for loop
 }
</Source>
```

10. Import the rule.

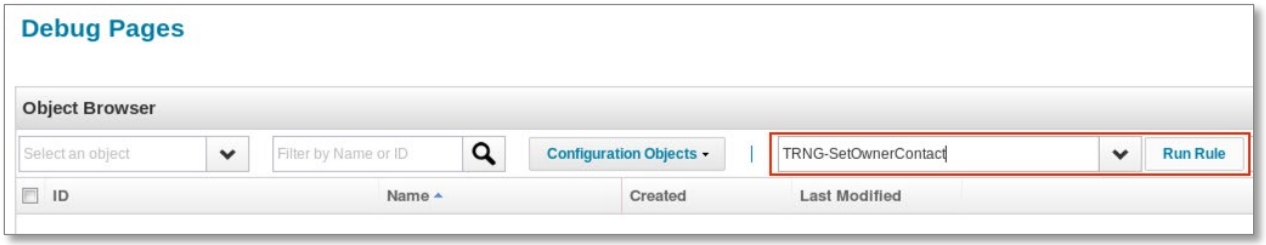
    a. Navigate to gear menu - **Global Settings** - **Import from File**.

    b. Click **Browse** and find the file **/home/spadmin/APITraining/config/Rule-TRNG-SetOwnerContact.xml** and click **Open**.

    c. Click **Import**.

11. Execute the rule in the Debug page.

    NOTE: Because this rule doesn't require any arguments, you can run it from the Debug page by choosing it from the rule list and clicking **Run Rule**.



When you run this rule from the Debug page, why does nothing appear in the dialog box?

_____

12. Examine the application definitions to look for the results.

a. Navigate to **Applications** - **Application Definition**.

b. Choose any of the applications to view its details.

c. Application extended attributes are displayed at the bottom of the **Details** tab. The example below is from the PRISM application.

Explore and Use Rule Arguments

For many of the standard rule types, rule arguments are objects, so those rules can access attributes of those arguments in their logic.

1. Return to the **HR System - Employees** application's **Creation Rule** that you looked at in Exercise 1.

2. Add logic to the Creation rule to fulfill this implementation-specific requirement:

    - Members of the Information Technology department should be assigned the System Administrator capability in IdentityIQ

    a. Since this is the HR System - Employees application, the **department** value is available in each record. Find the "account" argument in the **Arguments** list and click it. What is its variable type?

    _____

    b. Examine the JavaDoc for **sailpoint.object.ResourceObject**. What is one method available for retrieving a value from its map by its key name?

    _____

    c. Open the file **/home/spadmin/APITraining/config/TRNG-CreationSetPassword.xml**. At [1], complete the code with the method call that lets you evaluate the account's "department".

    d. Import the **/home/spadmin/APITraining/config/TRNG-CreationSetPassword.xml** file into IdentityIQ which will override the existing TRNG-CreationSetPassword rule.

e. Navigate to **Applications - Application Definition** and open the **HR System – Employees** application. Go to the **Rules tab** and click the ellipses (…) button next to the **Creation Rule**. Verify that your changes appear in the rule.

3. Add new accounts to the HR System - Employees source file to process.

   NOTE: You need to add new records because Creation rules only run when new identities are created, for example it only runs the first time an authoritative account is aggregated.

   a. Navigate in the file browser to **/home/spadmin/APITraining/Data**

   b. Open the **AuthEmployees.csv** file and scroll to the bottom.

   c. The final two lines are commented out with #. Remove the # to uncomment those two lines so the aggregation will read in accounts for two new users: Alice.Becker who is in the Information Technology department and Jose.Fernandez who is not.

   d. **Save** the file.

4. Run the HR aggregation.

   a. Navigate to **Setup** - **Tasks**.

   b. Right-click the **Aggregate Employees and Contractors** task and click **Execute in Background**. Click **OK** to acknowledge execution.

   c. Click the **Task Results** tab and refresh until the aggregation task shows completed.

5. Validate the rule functionality.

   a. Navigate to **Identities** - **Identity Warehouse**.

   b. Search for and view identities: **Alice.Becker** and **Jose.Fernandez**.

   c. Alice's **User Rights** tab should show the **System Administrator** capability selected while Jose's should not.

      NOTE: Only IT users who are aggregated *after* the implementation of this rule will be assigned this capability by the Creation rule since it only runs at initial identity creation. To fully meet the business requirement of assigning all Information Technology employees the System Administrator capability, you would need a different rule or a manual process to grant that capability to existing IT identities.

Exercise #3: SailPoint Context, Searches, and Filters

Objective

Explore the SailPointContext and its various search options; practice applying Filters into queries using QueryOptions

Overview

All members of the Accounting department based in London are being relocated to Brussels. You will write a rule to update the identities accordingly. This change might typically come through an HR aggregation, but for the purposes of this exercise, you will make the change in IdentityIQ directly.

Change Identity Location Attribute

1. Examine identity Adam Kennedy to confirm that he is currently in the Accounting department in the London location. He will be your test user for verifying the rule functionality.

2. Use the text editor to open the file **/home/spadmin/APITraining/config/Rule-TRNG-MoveAccountingDept.xml**.

3. Add the logic to retrieve all the Identity ID values for all Accounting department personnel in London, using the SailPointContext search method.

    a. Consult JavaDoc on the **SailPointContext** object. What package is the SailPointContext in?

    _____

    TIP: If you are trying to find a class in JavaDoc and you don't already know its package, choose **All Classes** in the Packages pane to make the Classes pane show the entire set of classes. Then search through the list for your desired class, alphabetically.

    b. Note that SailPointContext is an interface which inherits from the PersistenceManager interface. You will find multiple search() methods in the list of **Methods inherited from interface sailpoint.api.PersistenceManager**. Click any of them to open their details and see their different signatures.

        i. Look in the provided rule logic and find the properties variable. What is its type?

        _____

ii. Scroll through the list of search() methods in the **PersistenceManager** JavaDoc to find the method signature that will accept this variable as an argument. Note the method signature here.

_____

4. The search method always requires a QueryOptions argument. Find the line in the existing rule code that creates a QueryOptions.

   a. What is the QueryOptions variable name? _____

   b. Add the following line just after the QueryOptions variable is created to set setCloneResults to true.

   ```
   qo.setCloneResults(true);
   ```

   NOTE: The solution file **/home/spadmin/APITraining/solutions/Rule-TRNG-MoveAccountingDept.xml** for this exercise is missing qo.setCloneResults(true); on line 13. If you view or use this solution, add this line.

   c. Use JavaDoc for sailpoint.object.Filter to discover how to write the logic to build a Filter object specifying the department attribute equals Accounting and the location attribute equals London.
   HINT: There are multiple options here, but the provided rule comments/code guide you through building each filter requirement individually and then combining them together.

   _____

   _____

   _____

   d. Use JavaDoc for sailpoint.object.QueryOptions to discover how to add a filter to a QueryOptions.

   _____

   e. At [1] and [2], add the code from the previous two steps to the rule.

5. At [3], complete the context.search() method call to retrieve the identity ID values.

6. The logic to iterate through the returned IDs is provided for you. You need to add the logic to retrieve each Identity object one at a time.

a. What method will you use for that targeted lookup? Find its signature in the SailPointContext (or PersistenceManager) JavaDoc and note it here.

_____

b. At [4], complete the logic in the rule to perform that lookup and store the Identity object in the variable targetUser.

7. Save your file changes and import this rule into IdentityIQ.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE Rule PUBLIC "sailpoint.dtd" "sailpoint.dtd">
<Rule created="1567785587681" id="7f0000026d071aa8816d074d93e1006a" language="beanshell" name="TRNG-MoveAccountingDept">
  <Description>For each Identity in the Accounting department in the London location, change their location to Brussels</Description>
  <Source>
  import sailpoint.object.Identity;
  import sailpoint.object.QueryOptions;
  import sailpoint.object.Filter;

  // Initialize variables
  String properties = "id";
  QueryOptions qo = new QueryOptions();
  qo.setCloneResults(true);
  // [1] Build filter for department = "Accounting"
  //     Build filter for location = "London"
  //     Combine the two filters in an And relationship
  //------------------------------
  Filter f1 = Filter.eq("department","Accounting");
  Filter f2 = Filter.eq("location","London");
  Filter identityFilter = Filter.and(f1,f2);
  //------------------------------
  // [2] add filter to the QueryOptions
  //------------------------------
  qo.add(identityFilter);
  //------------------------------

  // [3] Retrieve Identity IDs with search()
  //------------------------------
  Iterator identIds = context.search(Identity.class, qo, properties);
  //------------------------------

  // Loop through identity IDs, retrieve each identity, and change its region attribute
  if (null != identIds) {
    while (identIds.hasNext()) {
      String identId = (String) identIds.next()[0];
      // [4] Retrieve single Identity using the id
      //------------------------------
      Identity targetUser = context.getObjectById(Identity.class, identId );
      //------------------------------

      // Change the user's region attribute, save, and commit
      targetUser.setAttribute("location","Brussels");
      context.saveObject(targetUser);
      context.commitTransaction();
    }
  }
  </Source>
</Rule>
```

8. Execute the **TRNG-MoveAccountingDept** rule in the IdentityIQ Console or the Debug page.

9. Examine Identity Adam Kennedy to see that his location attribute has been updated.

Exercise #4: Exploring IdentityIQ Objects

Objective

In this exercise, you will:

- Write API calls to create and operate on key objects in the IdentityIQ Object model which are frequently used in BeanShell development.
- Explore object relationships through object method usage.

Overview

There are two activities in this exercise which give you practice operating on commonly used object types in IdentityIQ.

1. The first builds a Custom object, which you will use in an activity in Exercise 5.
2. The second creates an After Provisioning rule that interrogates the ProvisioningResult returned by the provisioning operation.

Populate a Custom Object

You may have data in an external data source you want to use in an IdentityIQ process. In cases where this data needs to be queried repeatedly or frequently, it can be more efficient to retrieve that data into a custom object so that it is stored inside IdentityIQ, making it more efficient to access.

In this activity, you will write a rule to retrieve a code lookup table from an external database and transcribe it into an IdentityIQ Custom object. The parts of the rule that deal with making a connection to an external source are written for you, since that is beyond the scope of this course.

NOTE: This rule makes use of try-catch-finally blocks. These are standard for error handling in Java programming and are a best practice that applies to BeanShell development as well.

1. Open the file **/home/spadmin/APITraining/config/Rule-TRNG-CreateCustomObject.xml** in the text editor.
2. Your rule needs to import the Custom object's class to be able to access its methods. At [1], add that import statement, specifying the fully qualified class name.
   Follow the statements shown by adding "import sailpoint.object.Custom;" to the import lines which are already in that rule.

3. At [2], create an instance of a Custom object and give it a name. This will be done with two separate lines of code. Use JavaDoc to determine the constructor signature and the method to use to name the object.
   HINT: The method to name the Custom object is inherited from its base class.
   OPTIONAL: To ensure that this rule can be run multiple times without creating multiple regionCodes custom objects, include the following code in section [2]:

```
Custom codesCustObj = context.getObjectByName(Custom.class,
"regionCodes");
if (null == codesCustObj) {
     codesCustObj = new _____________;
     codesCustObj.___________("regionCodes");
}
```

4. At [3], add the logic to append the name-value pair from the resultSet row into the Custom object. This logic is performed inside the while loop so every row in the resultSet gets added to the Custom object.

5. Use JavaDoc to find the SailPointContext methods to save the object to the database and to commit the SQL transaction and add them to the rule at [4].

6. At [5], add the code to print an XML serialized version of the Custom object to standard out. This is step is for debugging purposes, so you can see what you have added to the object and what it looks like at the end of this process,

7. Save your changes. Compare with the completed code below. This image includes the optional code from Step 3.

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
// [1] import Custom class
//-------------------------------------
import sailpoint.object.Custom;
//-------------------------------------

// [2] Create Custom object with default constructor and name it "regionCodes"
//-------------------------------------
Custom codesCustObj = context.getObjectByName(Custom.class, "regionCodes");
if (null == codesCustObj) {
    codesCustObj = new Custom();
    codesCustObj.setName("regionCodes");
}
//-------------------------------------
// Call method in rule library
Connection dbCxn = connectToDatabase();
if (dbCxn == null) {
   // log message and exit
   String errMsg = "Cannot build Custom object - no database connection";
   log.error(errMsg);
   return errMsg;
}
// Run the query and populate the Custom object
try {
   String sqlQuery = "select * from regionCodes";

   PreparedStatement prStmt = dbCxn.prepareStatement(sqlQuery);
   ResultSet rs = prStmt.executeQuery();

   // loop through resultSet adding each row to the Custom object
   while ( (null != rs) &amp;amp;&amp;amp; (rs.next()) ) {
      String name = rs.getString("name");
      String code = rs.getString("code");

      // [3] Add name-value pair to Custom object
      //-------------------------------------
      codesCustObj.put(name,code);
      //-------------------------------------
   }
   rs.close();
   prStmt.close();

   // [4] Save and commit with SailPointContext
   //-------------------------------------
   context.saveObject(codesCustObj);
   context.commitTransaction();
   //-------------------------------------
   // [5] Print the Custom object
   //-------------------------------------
   System.out.println("Custom object: " + codesCustObj.toXml());
   //-------------------------------------

} catch (Exception ex) {
   log.error(ex);
} finally {
   try {
      dbCxn.close();
   } catch (SQLException sqlEx) {
      log.error(sqlEx);
   }
   dbCxn = null;
}
return "regionCodes Custom object complete";
```

8. Import the file **/home/spadmin/APITraining/config/Rule-TRNG-CreateCustomObject.xml** into IdentityIQ.

    a. Note the name of the rule. _____

9. Execute the rule from the Debug page.

    a. If your rule ran successfully, you should see "regionCodes Custom object complete" in the dialog box. If it did not, check the IdentityIQ log file (with the Tail IdentityIQ Log desktop shortcut) for error messages and correct your rule logic accordingly.

    b. Use the Tail Tomcat Standard Out desktop shortcut to view the XML representation of the Custom object. It should look like this:

```
Custom object: <?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE Custom PUBLIC "sailpoint.dtd" "sailpoint.dtd">
<Custom created="1560555706775" id="ff8080816a7f0804016b585e5997102b" name="regi
onCodes">
  <Attributes>
    <Map>
      <entry key="Americas" value="01"/>
      <entry key="Asia-Pacific" value="03"/>
      <entry key="Europe" value="02"/>
    </Map>
  </Attributes>
</Custom>
```

Write an After Provisioning Rule

In provisioning operations, the connector is provided a ProvisioningPlan, representing the requested access, and it records the operation's success or failure in a ProvisioningResult. The After Provisioning Rule can access that result information. In this activity, you will write an After Provisioning Rule for the LDAP application to email the request result information to the target user when it succeeds or to the application owner when it fails.

1. Import the file **/home/spadmin/APITraining/config/Rule-TRNG-ProvisioningResultNotification.xml** into IdentityIQ.

    a. What is the name of the rule? _____

2. Import the file **/home/spadmin/APITraining/config/EmailTemplate-TRNG-ProvisioningResultNotification.xml** into IdentityIQ. This is the email message that will be sent to the appropriate user by the rule.

3. Navigate to **Applications** > **Application Definition** and choose the **LDAP** application.

4. On the **Rules** tab, find the **After Provisioning Rule.** From the drop-down list, select the rule you just imported to attach it to the LDAP application. This makes it automatically run after provisioning requests get processed for the LDAP application.

5. Click the ellipses button (…) to open the rule in the rule editor.

6. Examine the rule argument called “result”. Click it in the Arguments list to see its description.

   According to this description, the provisioning result information may be provided to this rule in one of two places: in the _____ variable or annotated as a ProvisioningResult in the _____ variable. This depends on the connector’s implementation of its provision method. Your rule logic should account for those options.

7. At [1], complete the logic to get the “status” value out of the ProvisioningResult.

   a. If the result argument is not null, retrieve the status from that variable. Check the JavaDoc on sailpoint.object.ProvisioningResult for the method that returns the status attribute’s value.

   b. If the result is null, see if the plan argument contains a non-null ProvisioningResult and get the status from there. Check the JavaDoc on sailpoint.object.ProvisioningPlan for how to retrieve its ProvisioningResult and then use the same method you used before to get the status value out of that ProvisioningResult.

      NOTE: There is also a ProvisioningPlan class in the sailpoint.integration package; be sure you are looking at the JavaDoc for sailpoint.object.ProvisioningPlan.

   NOTE: With some connectors, it could also be returned in the AccountRequest inside the ProvisioningPlan, but for exercise simplicity, this rule will skip that additional layer of evaluation logic.

8. The test for a successful result value is provided for you in the rule. Look in the ProvisioningResult JavaDoc for the constant value to compare to for a failure response and complete the else if statement at [2] to check for request failure.

   NOTE: There are four possible values for a ProvisioningResult status, but in this rule, you only need to deal with the success and failure values.

9. Save your rule changes and save the application definition.

```
import sailpoint.object.ProvisioningPlan;
import sailpoint.object.ProvisioningResult;
import sailpoint.object.Application;

 ProvisioningResult planResult;
String status;
Map args = new HashMap();

// Populate status from result argument if it is set
// or populate it from the plan's ProvisioningResult
//------------------------------------------------------
if ( null != result) {
    planResult = result;
    status = result.getStatus();
} else {
    if (null != plan ) {
        planResult = plan.getResult();
        if (null != planResult) {
           status = planResult.getStatus();
        }
    }
}
//------------------------------------------------------

//add arguments to pass to email template
args.put("status", status);
args.put("group", getProvisionedGroup(plan));
args.put("target", plan.getNativeIdentity());

// Send email to user if successful or to app owner if failure
if (status.equals(ProvisioningResult.STATUS_COMMITTED)) {
    // Call rule library method in TRNG-UtilLibrary rule to send message to target user
    sendEmail(plan.getNativeIdentity(), "TRNG-ProvisioningResultNotification", args);

//Fill in if statement to check for failed requests
//-----------------------------------------------
} else if (status.equals(ProvisioningResult.STATUS_FAILED)) {
//-----------------------------------------------
    args.put("errorMsgs",planResult.getErrors());
    // Call rule library method in TRNG-UtilLibrary rule to send message to application owner
    sendEmail(application.getOwner().getName(), "TRNG-ProvisioningResultNotification", args);
}
// Do nothing if provisioning status is not success or failure
```

10. **Optional**: The logic to send the email message has been written for you, encapsulated inside the TRNG-UtilRuleLibrary's "sendEmail" method. Look at that rule through the Debug page to explore the method's logic.

    a. What are the arguments required by that sendEmail method?

    _____

    b. How does it determine the target user's email address?

    _____

    c. What method does it use to actually send the email message?

    _____

11. Test the success path for this rule.

    a. Submit a user access request through LCM (Manage User Access) to add the LDAP Users group for Adam Kennedy.

    b. Use the **Tail Email Log** desktop shortcut to view the Email log and verify that an email message was sent showing successful provisioning.

NOTE: If your rule fails, for example it contains syntax errors or otherwise does not have the correct logic, the provisioning operation will still succeed, since this rule only runs after provisioning. If you need to fix the rule and run it again, you can either choose a different user to test with or you can reset Adam Kennedy to use him again. To perform that reset, launch the LDAP browser, delete Adam Kennedy from the Users group, and re-aggregate LDAP accounts into IdentityIQ.

12. Test the failure path for this rule.

    a. Open a terminal window. At the command prompt, enter **StopLDAP** to invoke the command alias that will take the LDAP application offline. This will force a failure of provisioning.

    b. Submit a user access request through LCM (Manage User Access) to add the LDAP Users group for Amy Cox.

    c. Use the desktop shortcut to view the Email log and see that an email message was sent to the application owner, in this case, spadmin showing failure of provisioning.

```
To: spadmin@demoexample.com
Message-ID: <4355ab469485489fa17e22151433a285@example.com>
Subject: Provisioning Result Notification
MIME-Version: 1.0
Content-Type: multipart/mixed;
        boundary="----=_Part_48_2078065762.1561143829879"
X-Mailer: smptsend

------=_Part_48_2078065762.1561143829879
Content-Type: text/plain; charset=UTF-8
Content-Transfer-Encoding: 7bit


    Provisioning of Amy.Cox to the Users group in LDAP failed.
    Error messages:
        [ InvalidConfigurationException ]
 [ Possible suggestions ] a) Furnish the correct host and port. b) Ensure the Op
enLDAP host is up and running.
 [ Error details ] Failed to connect to server: training.sailpoint.com:389

------=_Part_48_2078065762.1561143829879--
```

    d. Return to the terminal window and use the **StartLDAP** alias to restart the LDAP application.

Exercise #5: Common API Uses

Objective

Create and attach rules to the Aggregation and Certification processes

Overview

You will create two rules in this exercise.

- In the first activity, you'll create a Customization rule which will run during aggregation of the HR authoritative application to set a new account attribute and identity attribute, called "regionCode", based on the Region value aggregated from those applications. This will use the Custom object you created in the previous exercise.
- In the second activity, you'll create a Certification Exclusion rule to specify which employees will not be included in a manager certification.

Implement a Customization rule in Aggregation

1. Some setup is required to support storing a new attribute value on the identity and on the account, as well as populating the identity attribute from the account attribute. You must define the account schema attribute, the identity attribute, and the mapping that populates the identity attribute *from* the account attribute.
    a. Open the **HR System - Employees** application definition and navigate to the **Configuration** - **Schema** tab.
    b. In the **Account** schema, add a new attribute and call it **regionCode**.
    c. Save the application definition.
    d. Navigate to the **Gear Menu** - **Global Settings** - **Identity Mappings**.
    e. Create a new identity attribute as follows:
        i. **Attribute Name**: regionCode
        ii. **Display Name**: Region Code
        iii. **Attribute Type**: String
        iv. **Edit Mode**: Read Only
        v. **Searchable**: checked
        vi. **Source Mappings**: Application Attribute
        Application = HR System – Employees
        Attribute = regionCode

f. Click **Add**.

g. Save the identity attribute definition.

NOTE: For simplicity, you will only apply this to the HR System - Employees application, though in a real environment, logic like this might apply to all authoritative sources.

2. Import the file **/home/spadmin/APITraining/config/Rule-TRNG-PopulateRegionCode.xml** into IdentityIQ.

   a. Note the name of the rule _____

3. Open the **HR System - Employees** application definition and navigate to the **Rules** tab.

4. Select the rule you noted above from the **Customization Rule** list to attach it to this application. Then click the ellipsis button (…) to open it in the Rule Editor.

5. At [1], add the logic to use the appropriate SailPointContext method to retrieve the "regionCodes" Custom object. **Hint**: This uses one of the getObject… methods.

6. The code to retrieve the region name from the resourceObject is already written. At [2], write the logic to look up the corresponding (String) region code from the Custom object.

7. At [3], write the code to add a regionCode attribute into the ResourceObject with the value you retrieved in the previous step.

8. **Save** your rule changes and **save** the application definition.

```
import sailpoint.object.Custom;
import sailpoint.object.ResourceObject;

// Retrieve the custom object called "regionCodes"
//------------------------------------------------
  Custom regionCodesObj = context.getObjectByName(Custom.class, "regionCodes");
//------------------------------------------------

// Get region from the resourceObject variable "object"
String region = object.getString("region");

// Look up the corresponding code value (a string)
// for the region, in the custom object
//------------------------------------------------
  String regionCode = regionCodesObj.getString(region);
//------------------------------------------------

// Set code value into resourceObject as attribute "regionCode"
//------------------------------------------------
  object.put("regionCode",regionCode);
//------------------------------------------------

// Return the resource object variable from the rule
return object;
```

9. Run the **Aggregate Employees and Contractors** task to update the accounts with the regionCode attribute.

10. Navigate to the **Debug** page and examine the **Identity** object for **Aaron Nichols**. Look for an identity attribute called **regionCode** and verify that it is populated as you expect.

    Note: The regionCode will not appear for Aaron.Nichols in the Identity Warehouse at this point. You would need to modify the UIConfig object to add regionCode to the displayed identity attributes.

Write and Use a Certification Exclusion Rule

Exclusion rules in a certification allow you to filter out whole entities or individual entitlements that you want to skip in a given certification campaign. They are helpful when the built-in configuration options are not already specific enough for your requirements. In this activity, you will create an exclusion rule to apply to a manager certification so that only *Employee* users who do *not* have an account on the PRISM application are included in the campaign.

1. Import the file **/home/spadmin/APITraining/config/Rule-TRNG-ExcludeContractorsAndPRISM.xml** into IdentityIQ.

    a. Note the name of the rule: _____

2. Navigate to **Setup** - **Certifications**.

3. Create a new **Manager** certification with these parameters (any parameters not mentioned should be left as the default values):

    a. **Basic** page:

        i. **Recipient**: check All Managers

        ii. **Run Now**: check

    b. **Advanced** page:

        i. **Exclusion Rule**: select the newly imported TRNG-ExcludeContractorsAndPRISM rule

        ii. **Save Exclusions**: check

4. Click the ellipses button next to the **Exclusion Rule** to open it in the Rule Editor.

5. Look at the rule's arguments list. Click each argument to view its type and description in a dialog box. Note the type and purpose of each of these arguments:

    a. entity _____

    b. items _____

    c. itemsToExclude _____

6. In the case of a Manager certification, the certifiable entity is an Identity object. What line of code in the rule logic validates that fact?

   _____

7. This rule first checks to see if the user meets either of the exclusion conditions, setting the "reason" accordingly. Then, if either condition is met, it will perform the exclusion. Complete the code for each exclusion requirement:

   a. At [1], get the user's type attribute and check if it equals "contractor"

   b. At [2], get a count of the user's accounts (Links) on the PRISM application so the subsequent code can determine if they have any PRISM accounts.

      NOTE: Examination of an identity's links (accounts) should be done through a helper class called the IdentityService (sailpoint.api.IdentityService). In JavaDoc, find the IdentityService method that counts the user's accounts on an application, given the identity and application as inputs, and complete the rule code accordingly.

8. If the identity has met either condition, perform the exclusion. Exclusion is performed by adding items to the exclusion (itemsToExclude) list and removing them from the items-to-certify (items) list. Both the items list and the itemsToExclude list are **java.util.ArrayList** objects. Do an internet search for the JavaDoc for that class. Find the methods needed to:

   a. Add all items currently in the items list into the itemsToExclude list

   b. Remove all items from the items list

   Write that logic into the rule at [3] and [4], respectively.

   NOTE: In this rule's case, you are performing an all-or-nothing operation on the user's items. In other circumstances, your rule's logic might need to exclude or include individual items more selectively. For example, if you only wanted to exclude items which pertained to the PRISM account, that would have required different, more complex logic than excluding *all* access for a user with a PRISM account.

9. Click the Returns variable in the Rule Editor. What is the purpose of the return value from this rule type?

   _____

10. **Save** the rule and click **Schedule Certification** to execute the certification.

```
import sailpoint.object.Identity;
import sailpoint.api.IdentityService;
import sailpoint.object.Application;

String reason = null;

// If this is not an Identity-focused certification, this rule won't exclude anything
if (entity instanceof Identity) {

   // [1] Get type from entity and then
   //     check for the entity's type attribute = "contractor"
   //--------------------------------------------------------
   String userType = entity.getType();
   if ((userType != null) &amp;amp;&amp;amp; (userType.equals("contractor"))) {
   //--------------------------------------------------------
      reason = "User is a Contractor";
   } else {
      Application chatApp = context.getObjectByName(Application.class,"PRISM");
     // [2] Count accounts this user has on the PRISM application
     //--------------------------------------------------------
     IdentityService is = new IdentityService(context);
     int prismAccounts = is.countLinks(entity,chatApp);
     //--------------------------------------------------------

     if (prismAccounts > 0) {
          reason = "User has a PRISM application account";
     }
   }
   // If we found a reason to exclude, exclude all items for the identity
   if (reason != null) {
      // [3] Add the entire items list to the itemsToExclude list
      //-----------------------------------------
      itemsToExclude.addAll(items);
      //-----------------------------------------

      // [4] Blank out the items list
      //-----------------------------------------
      items.clear();
      //-----------------------------------------
   }
}
// return the reason these items were excluded; this is displayed in the certification
// exclusions list if exclusions are saved
 return reason;
```

11. The certification campaign will generate quickly in the training environment. When it has been generated and has an active status, click the campaign in the **Certifications** list. Click **View Exclusions** link in the header. Confirm that the list of excluded items, each with its reason for exclusion, is shown in a dialog box.

Exercise #6: Best Practices and Performance

Objective

Practice defining and using custom loggers; use metering of code to measure performance

Overview

In this exercise, you will define a custom logger and enable it for debug-level logging in the log4j2.properties file. You will also add some meter statements to calculate execution timings on two different parts of the rule logic: one that uses getObjects() to retrieve and print information about identities and another that uses a search() projection query for the same task.

NOTE: The point of this metering exercise is to demonstrate how to use the meter and logging features, not to prove anything about these two operations' performance. The volume of data in your training environment and the specific operations the rule is executing do not reflect real-world processes. The statistics you will see may reflect disparate or equal performance results and should not be interpreted as proof that one of these operations is always more efficient than the other.

Explore Custom Loggers and Metering

1. Open the file **/home/spadmin/APITraining/config/Rule-TRNG-MeterPerformance.xml** in the text editor.

2. At [1], create a custom Log4j logger in the rule that will be unique to this rule. Following the naming convention [project].[objectType].[descriptive identifier], the logger name will be TRNG.rule.MeterPerformance. (You can choose a different naming convention if you prefer.)

3. At [2] and [3], complete the two logging lines in the rule by calling your logger's debug method to print the specified output to the log file.

4. At [4] and [5], call the Meter class's "enterByName" and "exitByName" methods around the first block of code. Give it a meaningful name, such as "TRNG-MeterPerformance getObject". Use the same name in both the enter and exit method calls.

5. At [6] and [7], repeat this around the second block of code, using a different name, for example, TRNG-MeterPerformance search).

6. **Optional**: Insert logger.trace() statements every 5 lines throughout the rule. Examples:

```
logger.trace("Starting MeterPerformance rule");
logger.trace("Running getObjects query");
logger.trace("Printing object values to log");
logger.trace("Running search query");
logger.trace("Printing search values to log");
logger.trace("Ending MeterPerformance rule");
```

NOTE: You will not see these values printed to the log file while the logger is set to print at the debug level.

7. Save your rule.

8. Import the Rule-TRNG-MeterPerformance.xml file into your IdentityIQ instance.

9. Configure logging in the log4j2.properties file.

   a. Open the **/home/spadmin/tomcat/webapps/identityiq/WEB-INF/classes/log4j2.properties** file in the text editor.

   b. At the bottom of the properties file, verify or modify the required two lines to define your logger and set its log level to debug". The name value must match the name you used in step 2 above.

```
logger.TRNGMeterPerformance.name=TRNG.rule.MeterPerformance
logger.TRNGMeterPerformance.level=debug
```

   NOTE: The tag on your logger ("TRNGMeterPerformance" in this example) does not matter, as long as it is unique across your properties file *and* the same tag is shared between the two lines. But it is a good practice to keep it in sync with your logger name.

   c. Save your changes.

   NOTE: In IdentityIQ, the log4j configuration file is checked for changes and auto-reloaded every 20 seconds, so you do not have to explicitly reload the configuration. If you disable this option in your implementation for any reason, it can still be reloaded manually through the **Debug** - **Logging** menu option.

```
import sailpoint.object.Identity;
import sailpoint.object.QueryOptions;
import sailpoint.object.Filter;
import sailpoint.tools.Util;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import sailpoint.api.Meter;

// [1] Set up custom logger
//------------------------------
Log logger = LogFactory.getLog("TRNG.rule.MeterPerformance");
//------------------------------

// Initialize variables
String properties = "name,location";
QueryOptions qo = new QueryOptions();
qo.addFilter(Filter.eq("region","Americas"));

// Retrieve identities with getObjects

// [4] Start metering getObjects
//----------------------------------
Meter.enterByName("TRNG-MeterPerformance getObjects");
//----------------------------------
List identityObjects = context.getObjects(Identity.class,qo);

if (null != identityObjects) {
  // Loop through list of identities
  for (Identity identityTarget : identityObjects) {
    if (null != identityTarget) {
        // [2] Log identity name and location with custom logger
        //------------------------------
        logger.debug("Target identity name: " + identityTarget.getName() + ", location: " +
                identityTarget.getAttribute("location"));
        //------------------------------
    }
  }
}
// [5] Stop metering getObjects
//----------------------------------
Meter.exitByName("TRNG-MeterPerformance getObjects");
//----------------------------------

context.decache();

// Retrieve identity attributes with projection query

// [6] Start metering search
//----------------------------------
Meter.enterByName("TRNG-MeterPerformance search");
//----------------------------------
Iterator identities = context.search(Identity.class,qo,properties);
if (null != identities) {
  while (identities.hasNext()) {
    Object target = identities.next();
    String name = (String) target[0];
    String location = (String) target[1];
    // [3] Log identity name and location with custom logger
    //------------------------------
    logger.debug("Target identity name: " + name + ", location: " + location);
    //------------------------------
  }
}
// [7] Stop metering search
//----------------------------------
Meter.exitByName("TRNG-MeterPerformance search");
//----------------------------------

Util.flushIterator(identities);
```

10. Run the **TRNG-MeterPerformance** rule from the Debug page.

11. Use the desktop shortcut to examine the **IIQ Log** file. See the debug log message from your rule's logger.

12. From the IdentityIQ Debug page, click the wrench icon and choose **Call Timings**. Scroll down the list to find your two meters and see the call timing statistics it gathered.

13. **Optional**: If you inserted the trace statements into your rule, change your log level in the log4j2.properties file to "trace" for your logger and re-run the rule. Observe the additional information in your IIQ Log file and the new statistics on the Call Timings page.