# SailPoint IdentityIQ Rules, Scripts and API

## Table of Contents

## Rules, Scripts, and Beanshell

### Rule Arguments

All rule and script hooks are automatically provided at least two input arguments – **context** and **log**.

**Context**

The context argument is a SailPointContext object, the entry point into the SailPoint API – giving you methods for accessing other objects and interacting with the SailPoint database.

**Log**

The log argument is a Logger object, which lets you use Log4j, the same logging utility used by the IdentityIQ core product, to generate logging messages from your rules.

## Print the BeanShell Namespace

If you're unsure about what variables are available to your rule or script, you can use the snippet of BeanShell shown below to figure it out. This logic prints the name and value of each variable in the BeanShell namespace – in other words, all the variables known to BeanShell at that point in time.

The code example below is URL encoded for use within an XML file.

```
System.out.println("Beanshell namespace:");

for (int i = 0 ; i &lt; this.variables.length ; i++) {

}
```

```
// output

Beanshell namespace:
bsf: BSFManager = org.apache.bsf.BSFManager@d4de53d
log: Log4jLog = org.apache.logging.log4j.jcl.Log4jLog@489edd0d
bsh: This = 'this' reference to Bsh object: NameSpace: Bsh Object
(bsh.NameSpace@4b8d8ae6)
context: InternalContext = sailpoint.server.InternalContext@37edef77
```

## Rule Libraries

Many rule objects are just a single block of code, executed together as a unit. But IdentityIQ also allows you to create rule objects that contain multiple separate functions or methods. These are called rule libraries.

Rule libraries are a way you can manage collections of methods that can be accessed from within other rules. These can help promote code re-use.

If you find yourself continually writing the same segment of code over and over again, consider including that logic in a rule library, so you can access it from other rules. If you ever need to change it, you can change it in one place and have it changed everywhere automatically. As a best practice for artifact management, you should group related methods together in a rule library, creating as many rule libraries as you need for different logical groupings.

In this example of rule referencing a rule library method, you see a rule library called My Library that contains a public method called doSomething.

```xml
<Rule name='My Library'>
   <Source>
      public void doSomething() { // do stuff }
      public String returnSomething() { // do stuff and return a String }
   </Source>
</Rule>
```

To use this method in a rule, you need to include a reference to the Rule Library in the XML of that rule with a element, then in the body of the rule itself, you can call the doSomething method and the system will find it in the rule library and execute its code.

```xml
<Rule…>
   <ReferencedRules>
       <Reference class='Rule' name='My Library'/>
   </ReferencedRules>
   <Source>
       doSomething();
   </Source>
</Rule>
```

## API Introduction

### IdentityIQ Architecture

Because IdentityIQ is a Java application, it follows an object-oriented paradigm. When you interact with IdentityIQ's API, you retrieve and act on objects:

- Identity objects
- Application objects
- Certification objects
- Link and bundle objects
- And more

## SailPointContext

> The SailPointContext object is a critical component for working with IdentityIQ's API in BeanShell rules and scripts. It is effectively the entry point to the API – the class that lets you retrieve and work with other objects. This lesson covers the key operations supported by the SailPointContext and the two objects used to direct and constrain its searching functions – the QueryOptions and the Filter objects

The SailPointContext supports object retrieval through multiple methods, and some of these support more than one signature, so they behave differently based on the arguments you provide.

### Multiple Object Retrieval

### getObjects

The `getObjects` method lets you retrieve one or more objects of a specified type. View examples below.

```
List <SailPointObject> getObjects(class)
List <SailPointObject> getObjects(class, QueryOptions) // better performance
```

### search

The `search` method lets you retrieve whole objects, like getObjects does, or individual attributes of objects that match your search criteria.

```
Iterator<SailPointObject> search(class, QueryOptions)

// Example
Iterator apps = context.search(Application.class, qo);

Iterator<SailPointObject> search(class, QueryOptions, properties) // better
performance

// Example
Iterator users = context.search(Identity.class, qo, "name, status");
Iterator users = context.search(Identity.class, qo, props);
```

### Single Object Retrieval

The SailPointContext offers two options for retrieving a single object by a unique identifier. Both require specification of the class name, either the name or GUID value as an argument – effectively, their search criteria.

**getObjectByName**

The `getObjectByName` method requires the class name as a string.

```
SailPointObject getObjectByName(class, String name)

// Example
Identity id = context.getObjectByName(Identity.class, "Bob.Smith");
```

**getObjectById**

The `getObjectbyId` method requires the the GUID value of the class.

```
SailPointObject getObjectById(class, String guid)

// Example
```

```
Application app = context.getObjectById(Application.class,
"ff80808161b51a2a0161b51a434a0002");
```

### QueryOptions - Filtering

```
// Department equals Accounting
Filter.eq(”department”,”Accounting)

// Age greater than 21
Filter.gt(“age”,21)

// First name starts with Ar
Filter.like(“firstName”,”Ar”,MatchMode.START)

// For multi-valued attributes, search for one or more matches with selectors like
“Cost Center contains all of a set of cost center codes"
Filter.containsAll(“costCenter”,costCenterCodes);

// Boolean filters require just an attribute and operator, like isnull or notnull
Filter.isnull(“region”)
Filter.notnull(“region”);
```

There are several ways to combine filters. You can use the operators and and or on the Filter class to combine filters. This can be straightforward or nested to multiple levels.

```
// Returns identities who are in the Finance department who also have Catherine
Simmons as a manager.
Filter f = Filter.and(Filter.eq("department", "Finance"),
Filter.eq("manager.name", "Catherine.Simmons")

// Returns identities who are either in the Finance department or who have
Catherine Simmons as a manager; only one of the criteria must be true.
Filter f = Filter.or(Filter.eq("department", "Finance"), Filter.eq("manager.name",
"Catherine.Simmons"))

// The last example selects identities who are either (in the Finance department
and report to Catherine Simmons) or (they have the last name Smith).
Filter f = Filter.or(Filter.and(Filter.eq("department", "Finance"),
Filter.eq("manager.name", "Catherine.Simmons")), Filter.eq("lastname", "Smith"));
```

## Object Model

### Custom Objects

The Custom Object is the most open-ended of object types in IdentityIQ, with the most varied uses.

A Custom Object is essentially a wrapper around an Attributes map, which is, itself, an extension of a standard Java HashMap (`Attributs<String, Object>`).

Perhaps you need to validate data entry against a defined set of valid values, or display human-readable information in the UI in place of a code value you might store on each identity. You can have these reference tables stored as Custom Objects. Storing that data locally in IdentityIQ, rather than having to retrieve it from an external resource every time, results in better system performance.

Custom objects can be used to set up advanced configurations for request approvals – like variable levels of approvals based on individual entitlement attributes – and other extended system functionality. By their very nature, Custom Objects are completely open ended.

## Common API Uses

### Aggregation Rules

#### Customization Rule

A Customization Rule runs for every account or group in the aggregation every time we aggregate.

It can be used to do data manipulation on specific account attributes or based on specific account attributes.

For example, if you need to exclude certain user accounts from the aggregation based on the user's work location you could use a customization rule.

#### Correlation Rule

When you read in accounts from non-authoritative applications, you need to help IdentityIQ map those accounts to the right identities through correlation logic.

#### Creation Rule

Creation rules only run when the aggregation is creating a new identity cube. So they're most commonly used just on authoritative applications, since authoritative aggregations are the ones we expect to be creating identities from.

## Best Practices

### Logging

#### Standard out vs. Log4j

`System.out.println("I'm logging this message all the time.");`

**Keep in mind that these messages should not remain in your code when you migrate to your production environments. You don't want to accidentally print potentially sensitive information into log files from your production environment.**

For troubleshooting purposes, it can be helpful to have logging options built into your code that can be turned on and off when needed, even in your production environment. IdentityIQ uses Log4j to support this need. The "log" object that's passed to every script and rule is Log4j logger object.

`log.debug(“I’m logging this message when debug logging is turned on.”);`

**Log4j Configuration**

Log4j is the Java-based logging library that IdentityIQ uses. The current version of IdenityIQ uses the version Log4j2, so the the properties file that controls its logging behavior is log4j2.properties.

*/WEB-INF/classes/log4j2.properties*

### Null and Void Checking

In BeanShell, like in Java and many other languages, you need to guard against null values and null pointer exceptions.

Before you call a method on an object variable, you should make sure the variable is defined and set.

- An unset variable is a `null` variable.
- An undefined variable is a `void` variable.

Because BeanShell supports *loose typing*, variables don’t have to be declared, and BeanShell will let you use variables that aren’t yet defined, but this can lead to problems when a rule is executed.

> In programming, loose typing means that a language does not enforce strict data types when variables are declared.

*Example*

```
// Not checking targetIdentity -> can lead to error
String mgrTitle = targetIdentity.getAttribute(“jobtitle”);

// Checking targetIdentity -> correct way
if ((targetIdentity != void) && targetIdentity != null))
    Identity manager = targetIdentity.getManager();
if (manager != null)
    String mgrTitle = manager.getAttribute(“jobtitle”);
```

### Saving Objects

Many rules expect you to change an object and return it, so IdentityIQ will save the object automatically. But you have great flexibility when writing rules and you might perform something beyond what was originally intended for the rule. If you’re accessing and modifying an object that the rule doesn’t expect you to change, the code that calls the rule may not perform the save automatically, so you’ll need to save that object yourself.

In the example below, the rule is setting a password in a place where IdentityIQ isn’t going to save the identity object automatically, so context.saveObject is used to save the user, and context.commitTransaction is used to commit it.

```
// user is an Identity object
user.setPassword(newPassword);
```

```
context.saveObject(user);
context.commitTransaction();
```

This is only necessary in instances where IdentityIQ is not expecting the action you are performing and therefore isn't set up to save the object you're changing.

### Processing Cursors

When working with iterators, like the one you get with a projection query, you must not leave open cursors on the database connection. This situation occurs when code loops through a set of objects looking for a specific condition then breaks out of the loop partway through iterating the list.

There are two options to prevent this situation:

- Ensure that you always process through the entire list, don't break out of an iteration loop
- Use the flushIterator method from the sailpoint.tools.Util class to clear the iterator when you are done with it

```
QueryOptions qo = new QueryOptions();
qo.addFilter(Filter.eq("application.name","Active_Directory");
Iterator iter = context.search(Link.class, qo, "id");

while (iter.hasNext()){
   Object[] row = (Object[]) iter.next();
   String id = row[0];
   // use id of the object to fetch it and process it
}

Util.flushIterator(iter);
```

There is no harm in flushing an iterator even if it was already exhausted, so generally speaking, this is the safer approach.

### Decache

You should ensure your code properly handles cached objects. If you are working with a specific object and no longer need it, you can evict it from the in-RAM cache with a decache statement. The SailPoint Context's decache method can be called with an object argument to clear just that object from memory.

`context.decache(object)`

For example, if you need to re-retrieve an object and want to ensure that you have a fresh copy of it from the database, you can clear the old copy from the cache then re-fetch the object into your script variable.

When you want to clear all loaded objects from the cache, there's a more open-ended decache method which takes no arguments.

`context.decache()`

This decache method evicts and clears all the objects from the cache but does not close your database connection, so open database cursors are left intact.

As an example, you can use this method when iterating over large volumes of objects, such as every 100 objects, to clear the cache.

```
Iterator iterator = context.search(objectClass, queryOptions, "id");
if (iterator != null) {
    while (iterator.hasNext()) {
        count++;
        // Retrieve object and process
        …
        if (count%100==0)
            context.decache();
    }
}
```

This is a recommended memory management practice. The above example runs a decache every 100 objects, freeing up all the memory that those objects have been occupying so it can be reused by this or another process. With potentially large objects like identity objects, this could amount to megabytes of memory being freed.

## Performance

### Searching

First, when performing queries in your custom code, use the projection query - `search()` with properties. This is significantly more efficient than getObjects.

In many cases, you only need an attribute or two anyway, so target your query to return just those attributes. Also, a projection query returns a database cursor instead of loading the entire set of objects into memory, making a difference in your system performance and heap space utilization.

Next, always try to filter your searches with a QueryOptions object that allows you to retrieve the minimum data you need. The less data your code has to process, the better. Let the query do as much of the work up front as possible.

### Iterative Rules

The performance of your rules will, of course, impact the performance of your system. This is especially important in iterative rules – those that run multiple times within a single process.

Most rules that run in aggregation processes are iterative, and rules that are run by an Identity Refresh task will be run for every identity that task is examining.

Examples include:

- BuildMap, MergeMaps, Transformation, ResourceObjectCustomization, Correlation
- Refresh, ManagerCorrelation

Rules that run during certification generation are also iterative. When IdentityIQ is determining who to exclude in an exclusion rule, it's going to examine every identity in the certification, which could be every identity in the system. A pre-delegation rule is also run for every certification item to determine whether to pre-delegate it and to whom.

Consider a buildmap rule that runs as part of an aggregation of 30,000 authoritative accounts.

- Rule runs for every row in a 30,000 line file
- Each row is processed in .02 seconds
- .02 seconds * 30,000 rows = 600 seconds or 10 minutes

Even with sub-second performance, it can still have a performance impact. It adds up quickly when you consider the volume of data you may be working with. If that was 3,000,000 rows, or if the rule took 1 second each time it ran, it would be even more significant.

Look for opportunities to pull operations out of iterative rules. For example, if you need to retrieve data from an external database, don't open a connection, perform a lookup, and close the connection in every rule run. Instead, perform the lookup once – outside of the iterative process – and store the data in a Custom Object in IdentityIQ's database. In this way, your iterative rules can do a more efficient local data lookup.

Some rules even have state objects that can share values across related rules, so make use of those too.

### Meter Object

You can measure the performance of your rules to understand what is and isn't working efficiently and to evaluate the effectiveness of any code changes made to improve performance.

The `sailpoint.api.Meter` class lets you collect statistics about execution timings for sections of code.

```
import sailpoint.api.Meter;

//Begin your code

Meter.enterByName("MyMeter");

//Do work

Meter.exitByName("MyMeter");

//End of code
```

You can import the class then call its enterByName method to mark the start of a process and its exitByName method to mark the end. The system will track statistics for the named meter:

- A count of how many times it executes
- The minimum, maximum and average run times

Include your whole rule under one meter name or have different parts of the rule tracked with different names.

You can also view the stats on the Call Timings page. To access the page, go to **Debug Page - wrench icon - Call Timings**. You'll see the timings for some built-in IdentityIQ processes plus your custom meters.