Before using this information and the product it supports, read the information in “Notices”.

Edition notice
This edition applies to version 7.1.0.6 of IBM Tivoli Netcool/Impact and to all subsequent releases and modifications until otherwise indicated in new editions.

References in content to IBM products, software, programs, services or associated technologies do not imply that they will be available in all countries in which IBM operates. Content, including any plans contained in content, may change at any time at IBM's sole discretion, based on market opportunities or other factors, and is not intended to be a commitment to future content, including product or feature availability, in any way. Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice and represent goals and objectives only. Please refer to the developerWorks terms of use for more information.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
Creating task data types ........................................ 84
Creating task data items ...................................... 84
Adding the tasks to the schedule ............................ 85
Writing a top scheduler policy ................................ 85
Running policies using the policy activator ............... 86
Creating a policy activator .................................... 86

Chapter 10. Handling hibernations ............................ 87
Hibernations overview ........................................... 87
Hibernating a policy ............................................. 87
Examples of hibernating a policy ........................... 87
Retrieving hibernations ........................................ 88
Retrieving hibernations by action key search .......... 88
Retrieving hibernations by filter ............................ 89
Waking a hibernation .......................................... 89
Retrieving the hibernation .................................. 89
Calling ActivateHibernation ................................. 90
Example ................................................................ 90
Removing hibernations ........................................ 90
Handling strings and arrays .................................. 90
Handling strings ................................................. 90
Handling arrays ................................................. 90

Chapter 11. Using Netcool/Impact functions to integrate with Webtop ... 97
Netcool/Impact functions ....................................... 97
Supported operating systems ................................. 97
Setting up the integration ..................................... 98

Chapter 12. Working with IPL to XML functions ................. 99
IPL to XML functions overview ............................... 99
Creating the XML document object ....................... 100
Adding a sub element ......................................... 100
Creating an unassociated element ......................... 101
Adding XML attributes to elements, simple approach .101
Adding XML attributes to elements that use Attributes .................................................. 102
Adding XML attributes to element objects adding attributes from an XmlNode ....................... 103
Adding the content to an XML element object ....... 103
Appending content to XML element objects .......... 104
Adding XML comments to element objects .......... 104
Adding XML element objects to each other
(ning) ................................................................ 105
Generating XML strings from document objects ..... 105
Replacement of default XML entities .................. 106
Element ordering in XML ..................................... 106
Examples of IPLtoXML functions usage ............. 106

Chapter 13. Working with functions .......................... 111
Activate ............................................................... 111
ActivateHibernation .......................................... 112
AddDataItem ...................................................... 113
BatchDelete ....................................................... 114
BatchUpdate ...................................................... 116
BeginTransaction ............................................. 117
CallDBFunction ............................................... 117
CallStoredProcedure ........................................ 118
ClassOf ............................................................ 119
CommandResponse ........................................... 120
CommitChanges ............................................... 127
CommitTransaction ........................................... 128
ConvertObjectsToJSON .................................... 128
ConvertXMLtoImpactObjects ............................... 129
CurrentContext ................................................. 131
Decrypt ............................................................ 132
DeleteDataItem ................................................. 132
Deploy ............................................................. 133
DirectSQL ......................................................... 135
Distinct ............................................................ 137
Encrypt .............................................................. 138
Escape .............................................................. 138
Eval ................................................................. 139
EvalArray ........................................................ 140
Exit ................................................................. 141
Extract .............................................................. 142
Float ................................................................. 143
FormatDuration ................................................ 144
GetByFilter ...................................................... 145
GetByKey .......................................................... 146
GetByLinks ........................................................ 148
GetByXPath ...................................................... 150
GetClusterName .............................................. 154
GetDate ............................................................. 154
GetFieldValue .................................................. 155
GetGlobalVar ................................................... 156
GetHTTP .......................................................... 157
GetHibernatingPolicies .................................... 162
GetHostAddress ............................................... 163
GetScheduleMember ......................................... 163
GetServerName ................................................ 165
GetServerVar ................................................... 165
Hibernate ........................................................ 166
Illegal ............................................................... 167
Int ................................................................. 167
JavaCall ........................................................... 168
JRExecAction .................................................... 170
Keys ................................................................. 171
Length ............................................................... 172
Load ................................................................. 172
LocalTime ......................................................... 173
Log ................................................................. 174
Merge ............................................................... 175
NewEvent ........................................................ 176
NewJavaObject ............................................... 177
NewObject ....................................................... 179
ParseDate ........................................................ 179
ParseJSON ........................................................ 180
Random ............................................................ 182
RDF functions .................................................. 183
RDFModel ........................................................ 183
RDFModelToString .......................................... 183
RDFModelUpdateNS ......................................... 184
RDFNodesIsResource ....................................... 185
RDFNodeIsAnon .............................................. 186
RDFParse ........................................................ 187
RDFRegister .................................................... 188
RDFUnRegister ............................................... 191
RDFSelect ....................................................... 192
| RDFStatement | ReceiveJMSMessage | RemoveHibernation | Replace | RESTfulAPIGET | RESTfulAPIPOST | RESTfulAPIPUT | RESTfulAPIDELETE | ReturnEvent | REextract | REextractAll | RollbackTransaction | SendEmail | SendInstantMessage | SendJMSMessage | SetFieldValue | SetGlobalVar | SetServerVar | SNMPGetAction | SNMPGetNextAction | SNMPSetAction | SnmpTrapAction | Split | String | Strip | Substring | Synchronized | ToLower | ToUpper | Trim | TBSM functions | PassToTBSM | RemoteTBSMShell | Known issue | UpdateDB | UpdateEventQueue | URLDecode | URLEncode | WSIvokeDL | WSNewArray | WSNewEnum | WSNewObject | WSNewSubObject | WSetDefaultPKGName | T rademarks | Index | Appendix. Notices | Notices | Index |
|--------------|------------------|-------------------|---------|--------------|--------------|--------------|-----------------|-------------|-----------|-----------|-------------------|------------|-------------------|------------|-------------|-------------|-------------|--------------|-----------------|--------------|--------------|------|--------|-------|-----------|-------------|----------|----------|------|-----------|----------|----------------|----------|-------------|-----------|--------|-----------------|----------|----------|--------|----------|----------|-----------|-------------|----------------|----------|--------|
About this publication

The Netcool/Impact Policy Reference Guide contains descriptions and complete syntax references for the Impact Policy Language (IPL) and JavaScript.

Intended audience

This publication is for users who are responsible for writing Netcool/Impact policies.

Publications

This section lists publications in the Netcool/Impact library and related documents. The section also describes how to access Tivoli® publications online and how to order Tivoli publications.

Netcool/Impact library

- Quick Start Guide, CN1LAML
  Provides concise information about installing and running Netcool/Impact for the first time.
- Administration Guide, SC27491805
  Provides information about installing, running and monitoring the product.
- Policy Reference Guide, SC27492105
  Contains complete description and reference information for the Impact Policy Language (IPL).
- DSA Reference Guide, SC27491905
  Provides information about data source adaptors (DSAs).
- Operator View Guide, SC27492005
  Provides information about creating operator views.
- Solutions Guide, SC27492305
  Provides end-to-end information about using features of Netcool/Impact.

Accessing terminology online

The IBM® Terminology Web site consolidates the terminology from IBM product libraries in one convenient location. You can access the Terminology Web site at the following Web address:


Accessing publications online

Publications are available from the following locations:

- The Quick Start DVD contains the Quick Start Guide. Refer to the readme file on the DVD for instructions on how to access the documentation.
Note: If you print PDF documents on paper other than letter-sized paper, set the option in the File ‧ Print window that allows Adobe Reader to print letter-sized pages on your local paper.


## Ordering publications


You can also order by telephone by calling one of these numbers:
- In the United States: 800-879-2755
- In Canada: 800-426-4968

In other countries, contact your software account representative to order Tivoli publications. To locate the telephone number of your local representative, perform the following steps:
2. Select your country from the list and click Go.
3. Click About this site in the main panel to see an information page that includes the telephone number of your local representative.

## Accessibility

Accessibility features help users with a physical disability, such as restricted mobility or limited vision, to use software products successfully. In this release, the Netcool/Impact console does not meet all the accessibility requirements.

## Tivoli technical training


## Support for problem solving

If you have a problem with your IBM software, you want to resolve it quickly. This section describes the following options for obtaining support for IBM software products:
- “Obtaining fixes”
- “Receiving weekly support updates” on page ix
- “Contacting IBM Software Support” on page ix

### Obtaining fixes

A product fix might be available to resolve your problem. To determine which fixes are available for your Tivoli software product, follow these steps:
2. Navigate to the Downloads page.
3. Follow the instructions to locate the fix you want to download.
4. If there is no Download heading for your product, supply a search term, error code, or APAR number in the search field.


**Receiving weekly support updates**

To receive weekly e-mail notifications about fixes and other software support news, follow these steps:

2. Click the My IBM in the toolbar. Click My technical support.
3. If you have already registered for My technical support, sign in and skip to the next step. If you have not registered, click register now. Complete the registration form using your e-mail address as your IBM ID and click Submit.
4. The Edit profile tab is displayed.
5. In the first list under Products, select Software. In the second list, select a product category (for example, Systems and Asset Management). In the third list, select a product sub-category (for example, Application Performance & Availability or Systems Performance). A list of applicable products is displayed.
6. Select the products for which you want to receive updates.
7. Click Add products.
8. After selecting all products that are of interest to you, click Subscribe to email on the Edit profile tab.
9. In the Documents list, select Software.
10. Select Please send these documents by weekly email.
11. Update your e-mail address as needed.
12. Select the types of documents you want to receive.
13. Click Update.

If you experience problems with the My technical support feature, you can obtain help in one of the following ways:

**Online**
Send an e-mail message to erchelp@u.ibm.com, describing your problem.

**By phone**
Call 1-800-IBM-4You (1-800-426-4409).

**World Wide Registration Help desk**
For worldwide support information check the details in the following link: [https://www.ibm.com/account/profile/us?page=reghelpdesk](https://www.ibm.com/account/profile/us?page=reghelpdesk)

**Contacting IBM Software Support**
Before contacting IBM Software Support, your company must have an active IBM software maintenance contract, and you must be authorized to submit problems to IBM. The type of software maintenance contract that you need depends on the type of product you have:
For IBM distributed software products (including, but not limited to, Tivoli, Lotus®, and Rational® products, and DB2® and WebSphere® products that run on Windows or UNIX operating systems), enroll in Passport Advantage® in one of the following ways:

**Online**

**By phone**

**For customers with Subscription and Support (S & S) contracts, go to the Software Service Request Web site at [https://techsupport.services.ibm.com/ssr/login](https://techsupport.services.ibm.com/ssr/login).**


For IBM eServer™ software products (including, but not limited to, DB2 and WebSphere products that run in zSeries, pSeries, and iSeries environments), you can purchase a software maintenance agreement by working directly with an IBM sales representative or an IBM Business Partner. For more information about support for eServer software products, go to the IBM Technical Support Advantage Web site at [http://www.ibm.com/servers/eserver/techsupport.html](http://www.ibm.com/servers/eserver/techsupport.html).

If you are not sure what type of software maintenance contract you need, call 1-800-IBMSERV (1-800-426-7378) in the United States. From other countries, go to the contacts page of the IBM Software Support Handbook on the Web at [http://www14.software.ibm.com/webapp/set2/sas/f/handbook/home.html](http://www14.software.ibm.com/webapp/set2/sas/f/handbook/home.html) and click the name of your geographic region for phone numbers of people who provide support for your location.

To contact IBM Software support, follow these steps:
1. “Determining the business impact”
2. “Describing problems and gathering information” on page xi
3. “Submitting problems” on page xi

**Determining the business impact**
When you report a problem to IBM, you are asked to supply a severity level. Use the following criteria to understand and assess the business impact of the problem that you are reporting:

**Severity 1**
The problem has a critical business impact. You are unable to use the program, resulting in a critical impact on operations. This condition requires an immediate solution.

**Severity 2**
The problem has a significant business impact. The program is usable, but it is severely limited.

**Severity 3**
The problem has some business impact. The program is usable, but less significant features (not critical to operations) are unavailable.
Severity 4

The problem has minimal business impact. The problem causes little impact on operations, or a reasonable circumvention to the problem was implemented.

Describing problems and gathering information

When describing a problem to IBM, be as specific as possible. Include all relevant background information so that IBM Software Support specialists can help you solve the problem efficiently. To save time, know the answers to these questions:

- Which software versions were you running when the problem occurred?
- Do you have logs, traces, and messages that are related to the problem symptoms? IBM Software Support is likely to ask for this information.
- Can you re-create the problem? If so, what steps were performed to re-create the problem?
- Did you make any changes to the system? For example, did you make changes to the hardware, operating system, networking software, and so on.
- Are you currently using a workaround for the problem? If so, be prepared to explain the workaround when you report the problem.

Submitting problems

You can submit your problem to IBM Software Support in one of two ways:

Online

Click Submit and track problems on the IBM Software Support site at http://www.ibm.com/software/support/probsub.html. Type your information into the appropriate problem submission form.

By phone

For the phone number to call in your country, go to the contacts page of the IBM Software Support Handbook at http://www14.software.ibm.com/webapp/set2/sas/f/handbook/home.html and click the name of your geographic region.

If the problem you submit is for a software defect or for missing or inaccurate documentation, IBM Software Support creates an Authorized Program Analysis Report (APAR). The APAR describes the problem in detail. Whenever possible, IBM Software Support provides a workaround that you can implement until the APAR is resolved and a fix is delivered. IBM publishes resolved APARs on the Software Support Web site daily, so that other users who experience the same problem can benefit from the same resolution.

Conventions used in this publication

This publication uses several conventions for special terms and actions, operating system-dependent commands and paths, and margin graphics.

Typeface conventions

This publication uses the following typeface conventions:

Bold

- Lowercase commands and mixed case commands that are otherwise difficult to distinguish from surrounding text
- Interface controls (check boxes, push buttons, radio buttons, spin buttons, fields, folders, icons, list boxes, items inside list boxes,
• Keywords and parameters in text

**Italic**

• Citations examples: titles of publications, diskettes, and CDs
• Words defined in text (example: a nonswitched line is called a *point-to-point line*)
• Emphasis of words and letters (words as words example: "Use the word *that* to introduce a restrictive clause."); letters as letters example: "The LUN address must start with the letter *L*."
• New terms in text (except in a definition list): a *view* is a frame in a workspace that contains data.
• Variables and values you must provide: ... where *myname* represents....

**Monospace**

• Examples and code examples
• File names, programming keywords, and other elements that are difficult to distinguish from surrounding text
• Message text and prompts addressed to the user
• Text that the user must type
• Values for arguments or command options

**PDF code examples with single quotation marks**

How to resolve issues with PDF code examples with single quotation marks.

Throughout the documentation, there are code examples that you can copy and paste into the product. In instances where code or policy examples that contain single quotation marks are copied from the PDF documentation the code examples do not preserve the single quotation marks. You need to correct them manually. To avoid this issue, copy and paste the code example content from the html version of the documentation.

**Operating system-dependent variables and paths**

This publication uses the UNIX convention for specifying environment variables and for directory notation.

When you use the Windows command line, replace the $variable with the %variable% for environment variables and replace each forward slash (/) with a backslash (\) in directory paths. The names of environment variables are not always the same in the Windows and UNIX environments. For example, %TEMP% in Windows environments is equivalent to $TMPDIR in UNIX environments.

**Note:** If you are using the bash shell on a Windows system, you can use the UNIX conventions.

• On UNIX systems, the default installation directory is /opt/IBM/tivoli/impact.
• On Windows systems, the default installation directory is C:\Program Files\IBM\Tivoli\impact.

Windows information, steps, and process are documented when they differ from UNIX systems.
Chapter 1. Getting started

This chapter contains information you need to get started with Netcool/Impact policies.

Managing policies

A policy contains a set of statements written in either the Impact Policy Language (IPL) or JavaScript.

Each statement is an instruction that describes a task to perform when certain events or status conditions occur in your environment. Instructions can be for high-level or low-level tasks. A single implementation of Netcool/Impact can have any number of associated policies.

An example of a high-level task is “Send an event to the Netcool/OMNibus ObjectServer.” An example of a low-level task is “Store a value in the internal MyDate variable.”

For an example of a policy, see “Policy example” on page 38.

Policy language components

You use the Impact Policy Language (IPL), or JavaScript to write the policies that you want Netcool/Impact to run.

The IPL is a scripting language similar in syntax to programming languages like C/C++ and Java. It provides a set of data types, built-in variables, control structures, and functions that you can use to complete a wide variety of event management tasks. You can create your own variables and functions, as in other programming languages.

JavaScript is a scripting programming language that is commonly used to add interactivity to web pages. It can also be used in browser environments. JavaScript uses the same programming concepts that are used in IPL to write policies. For more information about JavaScript syntax, see http://www.w3schools.com/js/default.asp

Policy log

The policy log is a text stream that records messages that are created during the run time of a policy. The messages in the policy log provide information about the system status and about any exceptions that might occur. You can write custom messages to the log from within a policy by using the Log function.

Policy context

The policy context is the set of all the variables whose values are assigned in the current policy. The policy context includes built-in variables such as EventContainer and the variables that you define. You can access the value of this context from within a policy with the CurrentContext function. This function returns a string that contains the names and current value of all the variables in the policy.

Policy scope

The scope of all variables in a policy is global. Everywhere you use a
Policy capabilities

You use policies to instruct Netcool/Impact to complete a wide variety of actions. Common actions include sending and retrieving Netcool/OMNIbus events, and adding, modifying, or deleting data that is stored in external data sources. In addition, you can also use a policy to send and receive email, to send and receive instant messages, and to communicate with external systems, devices, and applications.

You can create policies with the policy editor in the GUI. You can also copy and paste policies from a web page or document file into a plain text editor to remove the rich text format. Then, paste the policy into the policy editor. The policy editor tool provides a syntax checker, a policy tree view, and other utilities that help you create and manage policies more easily. For more information, see “Working with the policy editor” on page 4.

Policies are run by services that monitor the environment for events or changes in status.

These include the event reader and event listener services. You can also run a policy manually in the GUI or the nc_i_trigger command line utility. For more information, see “Managing policy triggers” on page 3.

Event handling

Most Netcool/Impact policies provide instructions for parsing incoming event data.

Handling event data from event reader and event listener services is one of the primary tasks that you perform using Netcool/Impact. Policies also often update event data that is stored in the ObjectServer or another event source. This is particularly the case for event enrichment policies, which correlate events with information stored in other data sources and then update the events using that information.

You can use a policy to perform event-related tasks.

- Parse event data that originates with the Netcool/OMNIbus ObjectServer or another event source (for example, event data stored in an SQL database)
- Send new events to ObjectServer or another event source
- Update existing events in ObjectServer or other event sources
- Delete existing events in ObjectServer or other event sources

Data handling

You can use a policy to perform data-related tasks.

- Retrieve data from a data source (for example, the internal data repository, or an SQL database)
- Add new data to a data source
- Update data stored in a data source
- Delete data stored in a data source
Retrieving data from external data sources is a very common policy task, particularly for event enrichment policies, which operate as described in “Event handling” on page 2.

**E-mail**
You can use a policy to perform e-mail-related tasks.
- Send new e-mail to an SMTP server using the e-mail sender service
- Parse e-mail retrieved from a IMAP or POP server using the e-mail reader service

**Instant messages**
You can use a policy to perform instant message-related tasks.
- Send instant messages to users of Yahoo! Messenger, AOL Instant Messenger, Microsoft Messenger, ICQ, and instant messaging clients that use a Jabber messaging service
- Parse instant messages sent to Netcool/Impact through a Jabber messaging service

**Integration with external systems, applications, and devices**
You can use a policy to integrate with external systems, applications, and devices that communicate.

Use the following methods:
- Web services API
- XML over HTTP
- JMS
- Custom socket protocols
- Proprietary third-party interfaces provided by data source adaptors (DSAs).

Exchanging data with using a Web services API, XML over HTTP, or JMS is a common way for a policy to communicate with external systems, applications, and devices. You can use these communication mechanisms to integrate Netcool/Impact with a very large variety of vendor software components.

**Accessing Service-related information from a policy**
You can also create a policy to access information related to Netcool/Impact services.

The following policy example can check if a service is running, and can start or stop a service.

```java
GetByFilter("Service", "Name = 'OMNIbusEventReader'", false);
Reader = OrgNode;
log("Is Reader Running " + Reader.Running);

// Starting the Reader
Reader.Running = true;

// Stop the Reader
Reader.Running = false;
```

**Managing policy triggers**
A policy trigger is a component or feature of Netcool/Impact that is capable of starting a policy.
Some policy triggers, like event readers, are controlled by Netcool/Impact. You control other policy triggers, like the nci_trigger script and the graphical user interface (GUI).

The nci_trigger script sends event data and a policy name to the event processor for handling. The event processor starts the policy and passes the event data to the policy in the form of an event container variable.

You can use the GUI to pass output parameters to the policy, but you cannot pass event data. Use of the GUI for running policies is for testing purposes only.

**Event readers as policy triggers**

An event reader service queries a Netcool/OMNIbus ObjectServer or other event source at intervals and retrieves new and updated events. It then determines whether any of the retrieved events are related to a policy that you have defined and then sends those events to the event processor service for handling. The event processor starts the appropriate policy in response to each incoming event and passes the event data to the policy in the form of an event container variable.

**Database listeners as policy triggers**

Currently listener services for use with Oracle databases are supported.

**Email readers as policy triggers**

The email reader service queries an IMAP or POP email server at intervals for new email messages. When the service retrieves a new email, it converts the email contents to the Impact event format and then determines whether the event matches one of the policies that you have defined. If the event matches a policy, the email reader sends it to the event processor service for handling. The event processor starts the appropriate policy and passes the event data to the policy in the form of an event container variable.

**Jabber readers as policy triggers**

The Jabber reader service listens for incoming instant messages. When the service receives a message, it converts the message contents to the event format and then sends the event to the event processor service for handling. The event processor starts the policy that is specified in the Jabber reader configuration and passes the event data to the policy in the form of an event container variable.

**Web services listeners as policy triggers**

The web services listener listens at an HTTP port for incoming SOAP/XML messages. When the service receives a message, it converts its contents to policy input parameters and passes them to the event processor service for handling. The event processor starts the policy that is specified by the web services listener configuration and passes the event data in the form of policy input variables.

**JMS listeners as policy triggers**

The JMS listener service listens for incoming JMS messages. When the service receives a message, it converts its contents to the event format and sends them to the event processor service for handling. The event processor starts the policy that is specified in the JMS listener configuration and passes the event data to the policy in the form of an event container variable.

**Working with the policy editor**

The GUI provides a policy editor that you can use to create and edit policies.
The policy editor offers a text editor with syntax highlighting, a function browser, a syntax checker, and other utilities to make it easy to manage policies. You can also write policies in an editor of your choice and then upload them into Netcool/Impact. After they are uploaded, you can edit them and check the syntax by using the policy editor.

**Tip:** Throughout the documentation, there are code examples that you can copy and paste into the product. In instances where code or policy examples that contain single quotation marks are copied from the PDF documentation the code examples do not preserve the single quotation marks. You need to correct them manually. To avoid this issue, copy and paste the code example content from the html version of the documentation.

Netcool/Impact 7.1.0.6 ships with a new Policy Editor. By default, the Netcool/Impact displays the existing Policy Editor. You can access the new policy editor by enabling the **Use new policy editor (beta)** checkbox in the user preferences. The new policy editor offers several enhancements to the existing editor including bracket matching, code folding, tab key support, status reporting, improved find/replace and an auto-complete tool.

**Note:** If you create and edit a policy by using an external editor of your choice, you must check its syntax with the `nci_policy` script before you run it. For more information about the `nci_policy` script, see the *Administration Guide.*
Chapter 2. Policy fundamentals

This chapter contains information about basic concepts needed to create policies using either the Impact Policy Language (IPL) or JavaScript.

Differences between IPL and JavaScript

When writing policies using IPL or JavaScript there are a number of differences. Use the following table as a reference.

Table 1. IPL and JavaScript differences

<table>
<thead>
<tr>
<th>IPL</th>
<th>JavaScript</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPL is not case-sensitive.</td>
<td>JavaScript is case-sensitive, keep this in mind when you are creating variables, statements, objects, and, functions.</td>
<td>&quot;RExtractAll&quot; on page 206</td>
</tr>
</tbody>
</table>
| When creating Arrays, in IPL you must use {} curly braces to assign array values. | In JavaScript, you must use [] square braces to assign array values. | • "Array" on page 11  
• "JavaCall" on page 168 |
| In IPL, the escape character can be either \s or \$. | In JavaScript, the escape character must be \\s. | "RExtractAll" on page 206 |
| In IPL, integers return as whole numbers, for example 1. | In JavaScript, integers are float as a result, numbers always display with decimals for example 1 becomes 1.0. | • "Distinct" on page 137  
• "EvalArray" on page 140  
• "String operators" on page 21  
• "ClassOf" on page 119 |
| ClassOf  
• If you pass an integer variable to ClassOf it returns as long in IPL.  
• If you pass a context variable to ClassOf, it returns as BindingsVarGetSettable in IPL.  
• If you pass an OrgNode variable to ClassOf, it returns as OrgNode in IPL. | • If you pass an integer variable to ClassOf it returns as double in JavaScript.  
• If you pass a context variable to ClassOf, it returns as JavaScriptScriptableWrapper in JavaScript.  
• If you pass an OrgNode variable to ClassOf, it returns as VarGetSettable in JavaScript. | |
| Event Container  
If you are using IPL, you can optionally reference event field variables using the dot notation or the @ notation. The @ notation is a special shorthand that you can use to reference members of EventContainer instead of spelling out the full name @Identifier. | If you are using JavaScript, you must use the dot notation EventContainer.Identifier. | "EventContainer" on page 15 |
### Table 1. IPL and JavaScript differences (continued)

<table>
<thead>
<tr>
<th>IPL</th>
<th>JavaScript</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exit</td>
<td>In JavaScript, when you use Exit in a user-defined function it exits the entire policy. If you want to stop a function in a JavaScript policy you must use the return command in the policy.</td>
<td>“Exit” on page 141.</td>
</tr>
<tr>
<td>Float</td>
<td>In JavaScript, the float function returns extra precision, for example 10.69567199999998 instead of 10.695672 for the function Eval.</td>
<td>“Eval” on page 139.</td>
</tr>
<tr>
<td>Eval</td>
<td>In JavaScript, integer division of 10/5 is 2 for the function Eval.</td>
<td>“Eval” on page 139.</td>
</tr>
<tr>
<td>JavaCall</td>
<td>JavaScript uses doubles for the numbers, when using JavaScript, for a JavaCall that needs an integer argument, you must use the Integer.parseInt JavaCall to create an actual integer.</td>
<td>“JavaCall” on page 168.</td>
</tr>
<tr>
<td>Like</td>
<td>JavaScript does not use the Like operator. The equivalent example is /.<em>abc.</em>/.test(teststring);</td>
<td>“Comparison operators” on page 20</td>
</tr>
<tr>
<td>vIf</td>
<td>In IPL, when you use Exit in a user-defined function it exits that function, and the policy continues.</td>
<td>“Exit” on page 141.</td>
</tr>
<tr>
<td>vElseIf</td>
<td>In IPL, the float function converts an integer, string, or Boolean expression to a floating point number.</td>
<td>“Float” on page 143.</td>
</tr>
<tr>
<td>vWhile</td>
<td>In IPL, Integer division of 10/5 is 2.0.</td>
<td>“Eval” on page 139.</td>
</tr>
<tr>
<td>vFunction</td>
<td>In IPL numbers are whole.</td>
<td>“JavaCall” on page 168.</td>
</tr>
<tr>
<td>vError</td>
<td>In IPL supports the Like operator. For example, teststring LIKE &quot;.<em>abc.</em>&quot;;</td>
<td>“Comparison operators” on page 20</td>
</tr>
<tr>
<td>vRaise</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vHandle</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Words and characters reserved for use by Netcool/Impact

A number of words and characters are reserved for use by Netcool/Impact.

You cannot use these words or characters in your policies.

### Words reserved for IPL

The following words are reserved for use by IPL:
- If
- ElseIf
- While
- Function
- Error
- Raise
- Handle

Policy function names, such as GetHTTP, are also reserved.
Words reserved for JavaScript variables

Any JavaScript reserved word or predefined JavaScript object and class names must not be used as variable names in a JavaScript policy.

Words reserved for the UI data provider

The UI data provider uses the UIobject1d field to index key fields. You cannot use the UIobject1d field in any of your policies. Netcool/Impact uses the AS UIDPROWNUM field in the query that is used for DB2 and Oracle databases.

You cannot use UIDPROWNUM as a field name for any of the connected DB2 and Oracle databases.

The topology and tree widgets use the UITreeNodeId field. The UITreeNodeId field is reserved for use by the UI data provider and it contains the following fields that are also reserved:
- UITreeNodeId
- UITreeNodeParent
- UITreeNodeStatus
- UITreeNodeLabel
- UITreeNodeType

Special characters reserved for the UI data provider

The UI data provider uses the comma (,) and ampersand (&) characters on the UI and in the URL for policies that use the DirectSQL policy function. You can use AS instead of ampersand (&) in policies.

Customize data output to follow the JavaScript standard

A property that is called impact.featuretostringassource.enabled can be added to the <servername> server.props file. The default value of the property is true, it converts the data into a string to enable the data output by the Log function to be the same in JavaScript and IPL.

To make the output the same in JavaScript and IPL the same, you must use the Object.toSource() function in JavaScript.

To enable the data output by the Log function to be the same in JavaScript and IPL, make the value true.

impact.featuretostringassource.enabled=true

To follow the JavaScript standard, make the value false.

impact.featuretostringassource.enabled=false

Policy-level data types

The policy-level data type is a different entity than the data types that are part of the data model.
When you are writing policies there are two categories of data types: simple data types and complex data types. The simple data types are integer, float, string, boolean, and date. The complex data types are array, context, data item, and event container.

**Simple data types**

Simple data types represent a single value.

Simple data types used to create policies:

**Integer**

The integer data type represents a positive whole number or its negative value. Examples of integers are 0, 1, 2, 3 and 4.

**Float**

The float data type represents a floating-point or decimal number. Examples of floats are 0.1243 and 12.245.

**String**

A string represents a sequence of characters, up to a length of 4 KB. In a policy, you enclose a string literal in either double or single quotation marks. An example of a string is abcdefg.

**Boolean**

Boolean represents a value of true or false. In a policy, you specify a boolean value using the true and false keywords.

**Date**

A date is a formatted string that represents a time or a calendar date. IPL and JavaScript use the following format for dates: YYYY-MM-DD HH:MM:SS.n, where n is a millisecond value. The date string can be enclosed in double or single quotation marks.

You specify the n millisecond value in date fields when you add a data item to a data type or when you update a data item. If the date does not have a millisecond value, specify 0. You also specify the millisecond value when you work with date fields in the GUI.

The following policy example shows how to add a new data item to a data type that contains a date field:

```java
MyContext = NewObject();
MyContext.FirstName = "Joe";
MyContext.LastName = "Example";
MyContext.Id = "1234";
MyContext.Created = "2006-07-31 11:12:13.4";

AddDataItem("Customer", MyContext);
```

**Complex data types**

Complex data types represent sets of values.

**Context**

Context is a data type that you can use to store sets of data.

Contexts are like the struct data type in C/C++. Contexts can be used to store elements of any combinations of data types, including other contexts and arrays.
This data is stored in a set of variables called member variables that are “contained” inside the context. Member variables can be of any type, including other contexts.

You reference member variables using the dot notation. This is also the way that you reference member variables in a struct in languages like C and C++. In this notation, you specify the name of the context and the name of the member variable separated by a period (.). You use this notation when you assign values to member variables and when you reference the variables elsewhere in a policy.

**Important:** A built-in context is provided, called the policy context, that is created automatically whenever the policy is run. The policy context contains all of the variables used in the policy, including built-in variables.

Unlike arrays and scalar variables, you must explicitly create a context using the NewObject function before you can use it in a policy. You do not need to create the member variables in the context. Member variables are created automatically the first time you assign their value.

The following example shows how to create a new context, and how to assign and reference its member variables:

```plaintext
MyContext = NewObject();
MyContext.A = "Hello, World!";
MyContext.B = 12345;
Log(MyContext.A + ", " + MyContext.B);
```

This example prints the following message to the policy log:

```
Hello, World!, 12345
```

The following policy shows how to create a context called MyContext and assign a set of values to its member variables.

```plaintext
MyContext = NewObject();
MyContext.One = "One";
MyContext.Two = 2;
MyContext.Three = 3.0;
Log(String1);
```

When you run this policy, it prints the following message to the policy log:

```
One, 2, 3.0
```

**Array**

The array is a native data type that you can use to store sets of related values.

An array in Netcool/Impact represents a heterogeneous set of data, which means that it can store elements of any combination of data types, including other arrays and contexts. The data in arrays is stored as unnamed elements rather than as member variables.

In IPL you assign values to arrays using the curly braces notation. This notation requires you to enclose a comma-separated list inside curly braces. The values can be specified as literals or as variables whose values you have previously defined in the policy:
**arrayname** = {**element1**, **element2**, **elementn**}

**Attention:** Arrays in IPL and JavaScript are zero-based, which means that the first element in the array has an index value of 0.

In JavaScript, use the square braces notation to assign array values as a comma-separated series of numeric, string, or boolean literals:

```
arrayname = [**element1**, **element2**, **elementn**]
```

**Important:** You can create an array of any size by manually defining its elements. You cannot import it from a file. You cannot have an array in an array unless it is a multi-dimensional array.

You access the value of arrays using the square bracket notation. This notation requires you to specify the name of the array followed by the index number of the element enclosed in square brackets. Use the following syntax to access the elements of a one-dimensional array and a multi-dimensional array:

```
arrayname[**element index**]
arrayname[first dimension **element index**][second dimension **element index**]
```

**Examples**

Here is an example of a one-dimensional array in IPL:

```
MyArray = {"Hello, World!", 12345};
Log(MyArray[0] + ", " + MyArray[1]);
```

Here is an example of a one-dimensional array in JavaScript:

```
MyArray = ["Hello, World!", 12345];
Log(MyArray[0] + ", " + MyArray[1]);
```

It prints the following text to the policy log:

```
Hello, World!, 12345
```

Here is an example of a two-dimensional array in IPL:

```
MyArray = {{"Hello, World!", 12345}, {"xyz", 78, 7, "etc"}};
Log(MyArray[0][0] + ". " + MyArray[1][0]);
```

Here is an example of a two-dimensional array in JavaScript:

```
MyArray = [["Hello, World!", 12345], ["xyz", 78, 7, "etc"]];
Log(MyArray[0][0] + ". " + MyArray[1][0]);
```

It prints the following text to the policy log:

```
Hello, World!.xyz
```

This example policy in IPL, uses the same two-dimensional array and prints the label and the value of an element to the parser log:

```
MyArray = {{"Hello, World!", 12345}, {"xyz", 78, 7, "etc"}};
log("MyArray is " + MyArray);
log("MyArray Length is " + length(MyArray));
ArrayA = MyArray[0];
log("ArrayA is " + ArrayA + " Length is " + length(ArrayA));
i = 0;
While(i < length(ArrayA)) {
    log("ArrayA[" + i + "] = " + ArrayA[i]);
i = i+1;
}
ArrayB = MyArray[1];
```
log("ArrayB is " + ArrayB + " Length is " + length(ArrayB));
i = 0;
While(i < length(ArrayB)) {
    log("ArrayB[" + i + "] = " + ArrayB[i]);
    i = i + 1;
}

This example policy in JavaScript, uses the same two-dimensional array and prints the label and the value of an element to the parser log:

MyArray is [["Hello, World!", 12345], ["xyz", 78, 7, "etc"]]
MyArray Length is 2
ArrayA is ["Hello, World!", 12345] Length is 2
ArrayA[0] = Hello, World!
ArrayA[1] = 12345
ArrayB is ["xyz", 78, 7, "etc"] Length is 4
ArrayB[0] = xyz
ArrayB[1] = 78
ArrayB[3] = etc

Here is the output in the parser log:
ArrayA[0] = Hello World!
ArrayA[1] = 12345

In the following policy, you assign a set of values to arrays and then print the values of their elements to the policy log.

Array1 = {"One", "Two", "Three", "Four", "Five"};
Array2 = {1, 2, 3, 4, 5};
Array3 = {"One", 2, "Three", 4, "Five"};

String1 = "One";
String2 = "Two";
Array4 = {String1, String2};

Log(Array1[0]);
Log(Array2[2]);
Log(Array3[3]);
Log(Array4[1]);
Log(CurrentContext());

Here, you assign sets of values to four different arrays. In the first three arrays, you assign various string and integer literals. In the fourth array, you assign variables as the array elements.

When you run the policy, it prints the following message to the policy log:

One
3
4
Two
"Prepared with user supplied parameters: "="("String2=Two (String),
ActionType=1 (Byte),
String1=One (String),
ReportsPolicyName=TestArrays1 (String),
EventContainer=(EventReaderName=DefaultEventReader) (EventContainer),
PolicyName=TestArrays1 (String),
ActionNodeName=TestArrays1 (String),
Array1=One (String), Two (String),
Array2=One (String), 2 (Long), Three (String), 4 (Long), Five (String)),
Array2=1 (Long), 2 (Long), 3 (Long), 4 (Long), 5 (Long),
Array1=(One (String), Two (String), Three (String), Four (String), Five (String)))
Data item
The data item is a policy-level data type that is used to represent data items in the Netcool/Impact data model.

Like a context, a data item consists of a set of named member variables. In a data item, however, the member variables have a one-to-one correspondence with fields in the underlying data source. As with contexts, you reference the member variables using the dot notation.

Many built-in functions return an array of data items. These functions include GetByFilter, GetByKey, GetByLinks, GetHibernatingPolicies, and GetScheduleMembers.

The following example shows how to assign and reference values in a data item:
MyCustomers = GetByKey("Customer", 12345, 1);
Log(MyCustomers[0].Name + ", " + MyCustomer[0].Location);
MyCustomers[0].Location = "New Location";

Data items store data by reference, rather than by value. When you change the value of a member variable in a data item, Netcool/Impact immediately changes the value in the underlying data source. Either by directly accessing it from the internal data repository or by sending an SQL UPDATE statement to the data source (for SQL database data sources).

Event container
The event container is a policy-level data type that represents an event.

Like contexts and arrays, an event container consists of a set of named member variables. In an event container, the member variables have a one-to-one correspondence with fields in the associated event source. As with contexts and arrays, you reference the member variables using the dot notation.

You can create your own event container in a policy by calling the NewEvent function. The following example shows how to create an event container, and how to assign and reference its member variables.
MyEvent = NewEvent("OMNIbusEventReader");
MyEvent.Node = "ORACLE_01";
MyEvent.Summary = "System not responding to ping request";

Variables
IPL and JavaScript have built-in variables, and user-defined variables.

You use built-in variables to handle data retrieved from external data sources and to handle event data. The built-in variables are DataItem, DataItems Num, and EventContainer.

You can use user-defined variables to store values during the lifetime of a policy in the same way that you use variables in other programming languages.

You also use variables to pass values to functions. The variable is updated after the function is complete. As a result, you can only use variables to pass values to functions.
Any JavaScript reserved word or predefined JavaScript object and class names must not be used as variable names in a JavaScript policy.

**Built-in variables**

Built-in variables during policy runtime are automatically populated and managed.

IPL and JavaScript have the following built-in variables:

- EventContainer
- DataItems
- DataItem
- Num

**EventContainer**

EventContainer is a built-in variable of the event container data type that represents an incoming event.

EventContainer has a set of member variables that correspond to fields in the incoming event. It also contains two predefined member variables. You can use these variables to specify the state of the event when you return it to the event source using the ReturnEvent function.

When a policy is triggered by an event reader, an email reader, or another mechanism, the event processor service creates a new EventContainer and populates its member variables with the field values in the event. The event processor creates one new member variable for each field in the event and assigns it the field value.

If you are using IPL, you can optionally reference event field variables using the dot notation or the @ notation. The @ notation is a special shorthand that you can use to reference members of EventContainer instead of spelling out the full name.

If you are using JavaScript you must use the dot notation EventContainer.Identifier.

The following example shows the use of the optional @ notation to reference event field variables for IPL.

```javascript
Log(@ActionKey);
Log(@ActionKey + ":" + @Summary);
@Summary = @Summary + ": Updated by Netcool/Impact";
```

The following example shows the use of the dot notation EventContainer.Identifier for JavaScript or IPL.

```javascript
Log(EventContainer.ActionKey);
Log(EventContainer.ActionKey + ":" + EventContainer.Summary);
EventContainer.Summary = EventContainer.Summary + ": Updated by Netcool/Impact";
```

Event state variables are the two predefined member variables that you can use to specify the state of an event when you return it to the event source using the ReturnEvent function. The policy engine does not automatically populate these variables when the policy is triggered.

[Table 2 on page 16](#) shows the event state variables.
Table 2. Event state variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EventContainer.JournalEntry</td>
<td>Set this field to add a new journal entry when you return an updated event to the event source. You specify the contents of the journal entry in single quotation marks. If you want to include newline or tab characters, you concatenate them separately with the string and enclose them in double quotation marks. You can add only journal entries to events when you update them from within a policy. You cannot add journal entries to new events.</td>
</tr>
<tr>
<td>EventContainer.DeleteEvent</td>
<td>Set this field to true to delete the event when you return it to the event source.</td>
</tr>
</tbody>
</table>

The following example shows how to add a new journal entry when you return an event.

```c
// Set the EventContainer.JournalEntry variable
EventContainer.JournalEntry = 'Modified on ' + LocalTime(GetDate()) \
    + "\r\n" + 'Modified by Netcool/Impact.';

// Return the event to the event source
ReturnEvent(EventContainer);
```

The following example shows how to delete an event when you return it to the event source.

```c
// Set the EventContainer.DeleteEvent variable
EventContainer.DeleteEvent = true;

// Return the event to the event source
ReturnEvent(EventContainer);
```

**DataItems**

DataItems is a built-in variable that stores an array of data items.

The DataItems variable is populated automatically by the functions that return data item arrays, such as GetByFilter, GetByKey, GetByLinks, GetHibernatingPolicies, and GetScheduleMember. You can use these functions to assign the returned data item array to other variables.

The following example shows the use of DataItems in a policy.

```c
// Call GetByFilter and pass the name of a data type, // and a filter as input parameters. GetByFilter // assigns the matching data items to the DataItems variable.
DataType = "Node";
Filter = "Location = 'New York'";
CountOnly = false;
GetByFilter(DataType, Filter, CountOnly);

// For each data item referenced by DataItems, // print the value of the Name field to the policy // log.
I = Num;
While (I > 0) {
    Log(DataItems[I-1]);
    I = I - 1;
}
```
**DataItem**

DataItem is a built-in variable that references the first element in the DataItems array.

You can use DataItem as shorthand in instances where you know the DataItems will contain only one element, or when you want to handle only the first element in the array. DataItem is equivalent to DataItems[0].

**Num**

Num is a built-in variable that stores the number of elements currently stored in the DataItems array.

You can use Num to count the number of data items returned by functions like GetByFilter, GetByKey, and GetByLinks, or you can use it to iterate through the DataItems array in a policy.

**Note:** The Num variable is supported for compatibility with earlier versions only. To retrieve the number of elements in an array returned by GetByFilter, GetByKey, or GetByLinks, use the Length function. For more information, see "Length" on page 172.

The following example shows how to use the Num variable in a policy.

```javascript
// Call GetByFilter and pass the name of a data type, 
// and a filter as input parameters. GetByFilter 
// assigns the matching data items to the DataItems variable.

dataType = "Node";
Filter = "Location = 'New York'";
CountOnly = false;

GetByFilter(dataType, Filter, CountOnly);

// For each data item referenced by DataItems, 
// print the value of the Name field to the policy 
// log.

I = Num;
While (I > 0) {
    Log(DataItems[I-1]);
    I = I - 1;
}
```

**User-defined variables**

User-defined variables are variables that you define when you write a policy.

Any JavaScript reserved word or predefined JavaScript object and class names must not be used as variable names in a JavaScript policy.

You can use any combination of letters and numbers as variable names as long as the first character starts with a letter:

You do not need to initialize variables used to store single values, such as strings or integers. For context variables, you call the NewObject function, which returns a new context. For event container variables, you call NewEvent. You do not need to initialize the member variables in contexts and event containers.

The following example shows how to create and reference user-defined variables:
MyInteger = 1;
MyFloat = 123.4;
MyBoolean = True;
MyString = "Hello, World!"

MyContext = NewObject();
MyContext.Member = "1";

MyEvent = NewEvent("OMNIBusEventReader");
MyEvent.Summary = "Event Summary";

Log(MyInteger + ", ", + MyEvent.Summary);

In the example in this section, you create a set of variables and assign values to them. Then, you use the Log function in two different ways to print the value of the variables to the policy log.

The first way you use Log is to print out each of the values as a separate call to the function. The second way is to print out all the variables in the policy context at once, using the CurrentContext function. The CurrentContext function returns a string that contains the names and values of all the variables currently defined in the policy.

VarOne = "One";
VarTwo = 2;
VarThree = 3.0;
VarFour = VarOne + ", ", + VarTwo + ", ", + VarThree;

Log(VarOne);
Log(VarTwo);
Log(VarThree);
Log(VarFour);

Log(CurrentContext());

When you run this policy, it prints the following message to the policy log:

One
2
3.0
One, 2, 3.0
"Prepared with user supplied parameters: " =(
ActionType=1 (Byte),
VarTwo=2 (Long),
ReportsPolicyName=TestVar (String),
EventContainer=(EventReaderName=DefaultEventReader) (EventContainer),
VarFour=One, 2, 3.0 (String),
PolicyName=TestVar (String),
ActionNodeName=TestVar (String),
VarOne=One (String),
VarThree=3.0 (Double))

In this example TestVar is the name of the policy. You do not have to declare variables before assigning their values in the way that you do in languages like C/C++ and Java. Arrays and scalar variables like integers or strings are created automatically the first time you assign a value to them. Contexts and event containers, however, must be explicitly created using the NewObject and NewEvent functions, as described later in this guide.

**Empty variables**

It is good practice to check that a variable has a value rather than assuming that it is set to a valid value. There are various times during the execution of policies
when a variable may not be defined or it may be null. To see if a variable is empty in IPL policies, add a NULL check for the variables. In JavaScript, check for null and undefined.

Below are examples of good defensive programming:

//javascript
if (typeof myVar === 'undefined' || myVar === null) {
    myVar = "";
}

//IPL
if (myVar == NULL) {
    myVar = "";
}

Operators

Operators are a special type of built-in function that modifies or compares a value or values.

IPL and JavaScript support a standard set of assignment, mathematical, comparison, boolean, and string operators. You use them to assign and retrieve values from variables, perform mathematical operations, compare values, and concatenate strings. IPL and JavaScript also define a set of operators that you can use to perform bitwise operations.

Assignment operator

The assignment operator is the equal sign =.

Use the single equal sign for assigning values to variables.

The following examples show the use of the assignment operator:

```javascript
a = 3;
b = "This is a test";
c = {"One", 2, true};
MyContext.a = "Test";
MyArray[0] = "Another test";
EventContainer.Summary = "Node " + EventContainer.Node + " not responding";
```

Bitwise operators

IPL and JavaScript support the use of the following Bitwise operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>AND: Returns a one in each bit position for which the corresponding bits of both operands are ones.</td>
<td>a &amp; b</td>
</tr>
<tr>
<td></td>
<td>OR: Returns a one in each bit position for which the corresponding bits of either or both operands are ones.</td>
<td>a</td>
</tr>
</tbody>
</table>
Table 3. Bitwise operators (continued)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>XOR: Returns a one in each bit position for which the corresponding bits of either but not both operands are ones.</td>
<td>a ^ b</td>
</tr>
</tbody>
</table>

Boolean operators

IPL and JavaScript support a range of boolean operators.

Table 4. Boolean operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>and</td>
</tr>
<tr>
<td>!</td>
<td>not</td>
</tr>
</tbody>
</table>

The following examples show the use of the boolean operators.

If ((a == 4) || (b <= 3)) ...
If ((a == "Test") && (b != "Test")) ...
If (!MyBool) ...

Comparison operators

IPL and JavaScript support a range of comparison operators.

Table 5. Comparison operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>==</td>
<td>Equal to</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal to</td>
</tr>
<tr>
<td>LIKE</td>
<td>Performs a comparison using Perl 5 regular expressions. JavaScript does not have the LIKE operator.</td>
</tr>
</tbody>
</table>

The following examples show the use of the comparison operators in IPL.

If ((a >= 4) || (b <= 3)) ...
If ((a == "Test") && (b != "Test")) ...
If (a LIKE "New York.*") ...

Mathematic operators

IPL and JavaScript support a range of mathematic operators.

Table 6. Mathematic operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
</tbody>
</table>
Table 6. Mathematic operators (continued)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Subtraction</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>%</td>
<td>Modulus</td>
</tr>
</tbody>
</table>

**Note:** The division operator always returns a floating-point value. Even if the dividend and the divisor are integers and if the quotient returned by the operation has no fractional value.

The following examples show the use of the mathematic operators.

\[
a = 1 + 2; \\
b = 4 - 3; \\
c = 6 * 4; \\
d = 6 / 4; \\
e = 8 \% 5;
\]

**String operators**

IPL and JavaScript support the use of the addition operator (+) for concatenating strings.

One common operation is to concatenate string values within an SQL filter. When you concatenate strings within a filter, make sure that all string literals are enclosed in single quotation marks within the resulting filter. For more information about SQL filters, see [“SQL filters” on page 79](#).

**Examples of using string operators**

**Note:**

The following example shows how to concatenate strings using the addition operator.

```javascript
MyString1 = "This is a";
MyString2 = "test.";
MyString3 = MyString1 + " " + MyString2;
Log(MyString3);
```

This example prints the following message to the policy log:

*Parser log: This is a test.*

The following example shows how to concatenate strings within an SQL filter.

```javascript
// Call GetByFilter and pass a data type name
// and a filter as input parameters

DataType = "Node";
Filter = "NodeName = '" + EventContainer.Node + "'";
CountOnly = false;

MyNode = GetByFilter(DataType, Filter, CountOnly);
```

Yet another example demonstrates how to add a string, to an array:

```javascript
MyArray = [];
MyString = 'test';
MyArray = MyArray + MyString;
```
//This will add MyString as a new element to the Array.

//Also in an Array you can remove an element using the Subtraction operator.
//If you do the following:

MyArray = ['test', 'test2'];
MyString = 'test';
MyArray = MyArray - MyString;
Log(MyArray);

The resulting array contains only the test2 string.

## Working with control structures

Control structures specify which code statements are executed under which conditions, and at what times.

IPL and JavaScript use two control structures: If and While. You use the If structure to perform branching operations. You use the While structure to loop over a set of instructions until a certain condition is met.

### If statements

You use the if statement to perform branching operations.

Use the if statement to control which statements in a policy are executed by testing the value of an expression to see if it is true. The if statement in the Impact Policy Language is the same as the one used in programming languages like C/C++ and Java.

The syntax for an if statement is the if keyword followed by a Boolean expression enclosed in parentheses. This expression is followed by a block of statements enclosed in curly braces. Optionally, the if statement can be followed by the else or elseif keywords, which are also followed by a block of statements.

```
if (condition){
  statements
} elseif (condition){
  statements
} else {
  statements
}
```

Where condition is a boolean expression and statements is a group of one or more statements. For example:

```
if (x == 0) {
  Log("x equals zero");
} elseif (x == 1){
  Log("x equals one");
} else {
  Log("x equals any other value.");
}
```

When the if keyword is encountered in a policy, the Boolean expression is evaluated to see if it is true. If the expression is true, the statement block that follows is executed. If it is not true, the statements is skipped in the block. If an else statement follows in the policy, the corresponding else statement block is executed.
In this example policy, you use the if statement to test the value of the Integer1 variable. If the value of Integer1 is 0, the policy runs the statements in the statement block.

```java
Integer1 = 0;
if (Integer1 == 0) {
    Log("The value of Integer1 is zero.");
}
```

When you run this policy, it prints the following message to the policy log:
The value of Integer1 is zero.

Another example shows how to use the else statement. Here, you set the value of the Integer1 variable to 2. Since the first test in the if statement fails, the statement block that follows the else statement is executed.

```java
Integer1 = 2;
if (Integer1 == 1) {
    Log("The value of Integer1 is one.");
} else {
    Log("The value of Integer1 is not one.");
}
```

When you run this example, it prints the following message to the policy log:
The value of Integer1 is not one.

### While statements

You use the while statement to loop over a set of instructions until a certain condition is met.

You can use the while statement to repeat a set of operations until a specified condition is true. The while statement in the Impact Policy Language is the same as the one used in programming languages like C, C++, and Java.

The syntax for the while statement is the while keyword followed by a Boolean expression enclosed in parentheses. This expression is followed by a block of statements enclosed in curly braces.

```java
while (condition) {
    statements
}
```

where condition is a boolean expression and statements is a group of one or more statements. For example:

```java
I = 10;
while(I > 0) {
    Log("The value of I is: " + I);
    I = I - 1;
}
```

When the while keyword is encountered in a policy, the Boolean expression is evaluated to see if it is true. If the expression is true, the statements in the following block are executed. After the statements are executed, Netcool/Impact again tests the expression and continues executing the statement block repeatedly until the condition is false.

The most common way to use the while statement is to construct a loop that is executed a certain number of times depending on other factors in a policy. To use the while statement in this way, you use an integer variable as a counter. You set the value of the counter before the while loop begins and decrement it inside the
The following example shows a simple use of the while statement:

```
Counter = 10;
while (Counter > 0) {
    Log("The value of Counter is ", Counter);
    Counter = Counter - 1;
}
```

Here, you assign the value of 10 to a variable named Counter. In the while statement, the policy tests the value of Counter to see if it is greater than zero. If Counter is greater than zero, the statements in the block that follows is executed. The final statement in the block decrements the value of Counter by one. The While loop in this example executes 10 times before exiting.

When you run this example, it prints the following message to the policy log:

```
The value of Counter is 10
The value of Counter is 9
The value of Counter is 8
The value of Counter is 7
The value of Counter is 6
The value of Counter is 5
The value of Counter is 4
The value of Counter is 3
The value of Counter is 2
The value of Counter is 1
```

The following example shows how to use the While statement to iterate through an array. You often use this technique when you handle data items retrieved from a data source.

```
MyArray = {"One", "Two", "Three", "Four"};
Counter = Length(MyArray);
while (Counter > 0) {
    Index = Counter - 1;
    Log(MyArray[Index]);
    Counter = Counter - 1;
}
```

Here, you set the value of Counter to the number of elements in the array. The While statement loops through the statement block once for each array element. You set the Index variable to the value of the Counter minus one. This is because arrays in IPL are zero-based. This means that the index value of the first element is 0, rather than 1.

When you run this example, it prints the following message to the policy log:

```
Four
Three
Two
One
```

In these examples, when you use this technique to iterate through the elements in an array, you access the elements in reverse order. To avoid doing this, you can increment the counter variable instead of decrementing it in the loop. This requires you to test whether the counter is less than the number of elements in the array inside the While statement.
The following example shows how to loop through an array while incrementing the value of the counter variable.

```plaintext
MyArray = {"One", "Two", "Three", "Four"};

ArrayLength = Length(MyArray);
Counter = 0;

while (Counter < ArrayLength) {
    Log(MyArray[Counter]);
    Counter = Counter + 1;
}
```

When you run this policy, it prints the following message to the policy log:

```
One
Two
Three
Four
```
Chapter 3. Function types

The Impact Policy Language (IPL) and JavaScript support built-in functions and user-defined functions.

The built-in functions are the functions, that are immediately available after the product is installed. Built-in functions complete common high-level and low-level tasks such as, sending an event to the ObjectServer or use regular expressions to extract a substring that matches this pattern. Examples of built-in functions are web services functions and SNMP functions. You can use the IPL or JavaScript to create and use function libraries.

User-defined functions are functions that you can use to organize the custom code in your policies.

Web services functions

Web services functions are functions that you use with the web services DSA.

You can use the following functions a to send messages to a web service provided by another application and to handle the message replies.

- WSInvokeDL
- WSNewArray
- WSNewEnum
- WSNewObject
- WSNewSubObject
- WSSetDefaultPKGName

For reference information about web services functions, see relevant sections in [Chapter 13, “Working with functions,” on page 111]. For more information about web services DSA, see the DSA Reference Guide.

SNMP functions

SNMP functions are functions that you use with the SNMP DSA.

You use these functions to send data to and retrieve data from SNMP agents. You can also use SNMP functions to send traps and notifications to SNMP managers.

The following SNMP functions are provided:

- SNMPGetAction
- SNMPGetNextAction
- SNMPSetAction
- SNMPTrapAction

For reference information about provided SNMP functions, see relevant sections in [Chapter 13, “Working with functions,” on page 111]. For more information about the SNMP DSA, see the DSA Reference Guide.
Java policy functions

With Java Policy functions, a policy can run a Java program or call any Java method if their Java classes are available from the Netcool/Impact runtime class path.

Before you can use Java Policy functions, you must make the Java classes available to Netcool/Impact during run time. To make the Java classes available complete the following steps:

1. Copy the Java classes to the $NCHOME/dsalib directory.
2. Restart the Impact Server to load the Java archive (JAR) files.

You must repeat this procedure for each Impact Server. This is necessary because the Java class files in the $NCHOME/dsalib directory are not replicated between servers.

Here are some of the most obvious scenarios where you would use these functions:

• To provide functionality not supported by the Impact Policy Language (IPL). Through the use of the Java Policy functions you can access java.io.* library of Java API and all other functions the Java API provides.

• To use third-party libraries or APIs within policies. Some system has a special communication protocol to access its data and it provides a Java library, which implements its communication protocol. You can add this library JAR file to the runtime class path and start to communicate with the system by starting its Java classes from a policy through the Java Policy functions.

• To complement usage of other DSAs. The Java Policy functions can be used to resolve interoperability issues among different web services platforms, for example, WebSphere, Weblogic, Axis, JAX-RPC, etc. The Netcool/Impact Web Services DSA is often used to invoke the web services running on the other web services platforms. But if the other platform is using certain complex types in their web services implementation, such as Hashtable or List, the call from Netcool/Impact fails because the Web Services DSA would not successfully compile the WSDL file due to these special types. To solve this problem, you can add the web services client jars generated by the other platform to the runtime class path. This will enable policies to make the soap calls by using these client classes, thus resolving the interoperability issue.

The following Java Policy functions are provided:

• GetFieldValue
• JavaCall
• NewJavaObject
• SetFieldValue

For reference information about these Java Policy functions, see relevant sections in Chapter 13, “Working with functions,” on page 111.

User-defined functions

User-defined functions are functions that you use to organize your code in the body of a policy.

Once you have defined a function, you can call it in the same way as the built-in action and parser functions. Variables passed to a function are passed by reference,
rather than by value. This means that changing the value of a variable within a
function also changes the value of the variable in the general scope of the policy.

User-defined functions cannot return a value as a return parameter. You can return
a value by defining an output parameter in the function declaration and then
assigning a value to the variable in the body of the function. Output parameters
are specified in the same way as any other parameter.

You can also declare your own functions and call them within a policy.
User-defined functions help you encapsulate and reuse functionality in your policy.

The syntax for a function declaration is the Function keyword followed by the
name of the function and a comma-separated list of input parameters. The list of
input parameters is followed by a statement block enclosed in curly braces.

Unlike action and parser functions, you cannot specify a return value for a
user-defined function. However, because the scope of variables in IPL policy is
global, you can approximate this functionality by setting the value of a return
variable inside the function.

Function declarations must appear in a policy before any instance where the
function is called. The best practice is to declare all functions at the beginning of a
policy.

The following example shows how to declare a user-defined function called
GetNodeByHostname. This function looks up a node in an external data source using
the supplied host name. Where "Node" represents the existing data type and
"Hostname" exists as a column in that data type.

Function GetNodeByHostName(Hostname) {

    DataType = "Node";
    Filter = "Hostname =" + Hostname + ";
    CountOnly = False;

    MyNodes = GetByFilter(DataType, Filter, CountOnly);
    MyNode[0] = MyNodes;
}

You call user-defined functions in the same way that you call other types of
functions. The following example shows how to call the function.

GetNodeByHostName("ORA_HOST_01");

Here, the name of the node that you want to look up is ORA_HOST_01. The function
looks up the node in the external data source and returns a corresponding data
item named MyNode. For more information about looking up data and on data
items, see the next chapter in this book.

**Function declarations**

Function declarations are similar to those in scripting languages like JavaScript.
Valid function names can include numbers, characters and underscores, but cannot
start with a number.

An example of a user-defined function:
Function MyFunc(DataType, Filter, MyArray) {
    MyArray = GetByFilter(DataType, Filter, False);
}

**Calling user-defined functions**

You can call a user-defined function as follows:
```
Funcname([[param1, param2 ...]])
```

The following example shows a user-defined function call:
```
MyFunc("User", "Location = 'New York'", Users);
```

**Examples of user-defined functions**

The following example show how variables are passed to a function by reference:
```
// Example of vars by reference
Function IncrementByA(NumberA, NumberB) {
    NumberB = NumberB + NumberA;
}
SomeInteger = 10;
SomeFloat = 100.001;
IncrementByA(SomeInteger, SomeFloat);
Log("SomeInteger is now: " + SomeInteger);
// will return: IntegerA is now 10
Log("SomeFloat is now: " + SomeFloat);
// will return: FloatB is now 110.001
```

The following example shows how policies handle return values in user-defined functions:
```
// Example of no return output
Function LogTime(TimeToLog) {
    If (TimeToLog == NULL) {
        TimeToLog = getdate();
    }
    Log("At the tone the time will be: " + localtime(TimeToLog));
}
LoggedTime = LogTime(getdate());
Log("LoggedTime = " +LoggedTime);
// will return: "LoggedTime = NULL" as nothing can be
// returned from user functions
```

**Local transactions**

You use local transactions in a policy if you want to use more than one SQL operation to be treated as a single unit of work.

This can be useful for cases where there is a group of related SQL commands that have to be committed to the database only when all of them are executed successfully. The following functions in the policy language enable the use of local transactions in an Impact policy:
- BeginTransaction
• CommitTransaction
• RollbackTransaction

For reference information about these functions, see “Local transactions template” on page 49 and relevant sections in Chapter 13, “Working with functions,” on page 111.

Working with function libraries

Function libraries is a feature you can use to create a set of stored functions that can be called from any policy.

Use external function libraries to encapsulate and reuse the custom code in your policies and are available for IPL and JavaScript policy languages.

Creating function libraries

A function library is a special type of policy that contains only user-defined functions.

You create this policy and define the functions that it contains in the same way you create standard policies. You can use parameters in the functions as both input and output variables.

• For an example of how to create function libraries that use JavaScript, refer to the example in the Load function, see “Load” on page 172.

• The following IPL example shows a function library. This library is a policy named UTILS_LIBRARY.

```
// NormalizeString trims whitespace, replaces the space character with an underscore and converts all characters to upper case
Function NormalizeString(StringToNormalize) {
    StringToNormalize = Trim(StringToNormalize);
    StringToNormalize = Replace(StringToNormalize, " ", "_" );
    StringToNormalize = ToUpper(StringToNormalize);
}
```

```
// GetCustomersByNode returns an array of data items from a data source, where each data item represents a customer
Function GetCustomersByLocation(Location, Customers) {
    Type = "Customer";
    Filter = "Location = "+ Location + ";";
    CountOnly = false;
    Customers = GetByFilter(Type, Filter, CountOnly);
}
```

Calling functions in a library

To call a function in function library, you specify the library and function name.

• For an example of how to call functions in a library that uses JavaScript, refer to the example in the Load function, see “Load” on page 172.

• For IPL, use the following example which uses the following format:

```
function_library.function_name(param, [param ...])
```

Where function_library is the name of the library policy, function_name is the name of the function, and param is the value of one or more parameters required by the function. You do not need to explicitly reference or include the library name before you call its functions, as is required in programming languages like C and
C++. The following example shows how to call functions in the library named UTILS_LIBRARY. The functions are the same as those defined in the previous example.

```cpp
// Normalize location string
UTILS_LIBRARY.NormalizeString(Location);

// Get customers at the specified location
UTILS_LIBRARY.GetCustomersByLocation(Location, Customers);

// Print customer info to the policy log
Log(Customers);
```

### Synchronized statement blocks

You can use synchronized statement blocks to write thread-safe policies for use with a multi-threaded event processor.

You can use synchronized statement blocks in situations where more than one instance of a policy or different policies that access the same resource run simultaneously on different event processor threads.

Synchronized statement blocks consist of any set of IPL statements that are enclosed in curly braces and set apart with the `synchronized` keyword and a synchronization identifier.

The syntax for a synchronized statement block is as follows:

```cpp
synchronized(identifier) { statements }
```

Where `identifier` is a unique name for the statement block and `statements` are any IPL programming statements.

The following example shows how to create a synchronized statement block for IPL:

```cpp
synchronized(update_table) {
    
   DataType = "Customer";
    Filter = "Location = 'New York'";
    UpdateExpression = "Location = 'Raleigh', Facility = 'SE_0014'"

    BatchUpdate(DataType, Filter, UpdateExpression);
}
```

To create a synchronized statement block for JavaScript you create a function, and then call the function with a Synchronizer.

```javascript
function update_table() {
    DataType = "Customer";
    Filter = "Location = 'New York'";
    UpdateExpression = "Location = 'Raleigh', Facility = 'SE_0014'"
    BatchUpdate(DataType, Filter, UpdateExpression);
}
```

```javascript
syncmyFunc = new Packages.org.mozilla.javascript.Synchronizer(update_table);
syncmyFunc();
```

When Netcool/Impact processes a synchronized statement block, it registers the synchronization identifier and does not allow other synchronized statement blocks
with that identifier to run until the registered block is finished running. Other statement blocks with the same identifier to run sequentially, rather than simultaneously, with the first block. As a result, resources accessed in the synchronized portion of the policy are protected from simultaneous access by multiple threads.
Chapter 4. Configuring policy exceptions

You can raise and handle policy-level exceptions.

The Impact Policy Language and JavaScript can raise and catch exceptions within a policy and handle Java exceptions that are raised internally when a policy is run.

Raising exceptions

To raise an exception, you use the Raise keyword.

The following example shows the syntax for Raise:

```
Raise ExceptionName(ExceptionText);
```

where `ExceptionName` is a unique name for the exception and `ExceptionText` is the text output of the exception. This output is printed to the server log when the error is encountered. You can also access it inside an error handler using the `ErrorMessage` variable.

The following example shows how to raise an exception using the `Raise` keyword. In this example, the function raises an exception named `IntOutOfRangeException` if the value of the `Param1` parameter is less than 0.

```
Function MyFunction(Param1, Param2) {
    If (Param1 < 0) {
        Raise IntOutOfRangeException("Value of Param1 must be greater than 0");
    }
}
```

Handling exceptions

To handle an exception, you declare an exception handler.

The handler is a function that is called each time that a specific exception is raised. The exception is raised at the policy level, or a specific Java™ exception is raised by Netcool/Impact during the execution of a policy.

Declare exception handlers in advance of any position where they are triggered in a policy. Insert error handlers at the beginning of a policy before you specify any other operations.

The following example shows the syntax for exception handlers:

```
Handle ExceptionName {
    statements ...
}
```

Where `ExceptionName` is the name of the exception raised by using the `Raise` keyword, or the name of the Java exception class that is raised by Netcool/Impact during the execution of the policy.

The following example shows how to handle policy-level exceptions by using an exception handler.
The following example shows how to handle Java exceptions by using exception handlers.

```java
Handle java.lang.NullPointerException {
    Log("Null pointer exception in policy.");
}
Handle java.lang.Exception {
    Log("ErrorMessage: "+ErrorMessage);
    MyException = javaCall("java.lang.Exception", ExceptionMessage, "getCause", null);
    Log("MyException is "+MyException);
    MyException = javaCall("java.lang.Exception", MyException, "getCause", null);
    Log("MyException again is: "+MyException);
}
```

The following examples show how to handle JavaScript exceptions by using exception handlers.

```javascript
try {
    //Run some code here
} catch(err) {
    //Handle errors here
}

// Example 1:
try {
    MyFunction(Param1, Param2);
} catch(e) {
    if (e == "IntOutOfRangeException") {
        Log("Error: Value of parameter submitted to MyFunction is less than 0");
    }
}
```

```javascript
function MyFunction(Param1, Param2) {
    If (Param1 < 0) {
        throw "IntOutOfRangeException";
    }
}
// Example 2:
try {
    //...code that is running...
} catch(e) {
    if (e instanceof java.lang.NullPointerException) {
        Log("Null pointer exception in policy.");
    }
    if (e instanceof java.lang.Exception) {
        Log("ErrorMessage: "+ErrorMessage);
        MyException = javaCall("java.lang.Exception", ExceptionMessage, "getCause", null);
        Log("MyException is "+MyException);
        MyException = javaCall("java.lang.Exception", MyException, "getCause", null);
        Log("MyException again is: "+MyException);
    }
}
```

---

**Clear cache syntax**

The policy language provides a syntax that you can use to clear the cache associated with a particular data type.

Netcool/Impact stores information about data types in another, system level data type named Types. Each data item in Types represents a data type that is defined in the system.
To clear a data type cache, you first call the GetByFilter or the GetByKey function and retrieve the data item from Types that corresponds to the data type. The key value for data items in Types is the data type name. Then you set the clearcache member variable associated with the data item to true.

The following example shows how to clear the cache of a data type named User:

```csharp
DataType = "Types";
Key = "User";
MyTypes = GetByKey(DataType, Key, 1);
MyTypes[0].clearcache = true;
```

### Date/Time patterns

The policy language provides a date/time pattern syntax that you can use with the LocalTime function to format date/time strings and with the ParseDate function to convert formatted strings to the number of seconds in UNIX time.

The pattern syntax consists of a set of symbols that specify how the date/time is formatted. You use the symbols alone, or with other formatting characters, such as : and /.

#### Symbols

The following table shows the symbols used in date/time patterns.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>Era</td>
</tr>
<tr>
<td>y</td>
<td>Year</td>
</tr>
<tr>
<td>M</td>
<td>Month</td>
</tr>
<tr>
<td>d</td>
<td>Day</td>
</tr>
<tr>
<td>h</td>
<td>Hour</td>
</tr>
<tr>
<td>H</td>
<td>Hour</td>
</tr>
<tr>
<td>m</td>
<td>Minute</td>
</tr>
<tr>
<td>s</td>
<td>Second</td>
</tr>
<tr>
<td>S</td>
<td>Millisecond</td>
</tr>
<tr>
<td>E</td>
<td>Day in week</td>
</tr>
<tr>
<td>D</td>
<td>Day in year</td>
</tr>
<tr>
<td>F</td>
<td>Day of week in month (for example, if day is third Friday in a month, value is 3).</td>
</tr>
<tr>
<td>w</td>
<td>Week in year</td>
</tr>
<tr>
<td>W</td>
<td>Week in month</td>
</tr>
<tr>
<td>a</td>
<td>AM/PM marker</td>
</tr>
<tr>
<td>k</td>
<td>Hour in a day</td>
</tr>
<tr>
<td>K</td>
<td>Hour in AM/PM</td>
</tr>
<tr>
<td>z</td>
<td>Time zone</td>
</tr>
</tbody>
</table>
Examples

The following example shows how to return the given number of seconds in various formats using LocalTime.
Seconds = GetDate();

Time = LocalTime(Seconds, "MM/dd/yy");
Log(Time);

Time = LocalTime(Seconds, "HH:mm:ss");
Log(Time);

This example prints the following message to the policy log:
06/19/03
13:11:24

Policy example

The following policy is a complete example of a Netcool/Impact policy.

This policy accepts incoming event information from an event reader service and uses this information to look up affected internal customers in an external database. The policy then sends an e-mail to each customer and reports the event data to a third-party help desk system using the web services API provided by that application.

// Look up customers affected by the event in the external database.
// Customer is name of a Netcool/Impact data type associated with // the database table that stores customer data.

DataType = "Customer";
Filter = "Server =" + EventContainer.Node + "";
CountOnly = false;

Customers = GetByFilter(DataType, Filter, CountOnly);

// Send an e-mail to each customer with notification of the event
Count = Length(Customers);
while (Count > 0) {
    Address = Customer.Email;
    Subject = "Change in server status";
    Message = "Netcool/Impact has detected a change in status for server "
    + " EventContainer.Node " + ". Summary is " + EventContainer.Summary;
    Sender = "Netcool/Impact";
    ExecuteOnQueue = false;
    Count = Count - 1;
    SendEmail(Customer, Address, Subject, Message, Sender, null, ExecuteOnQueue);
}

// Report the event details to the third-party help desk system using // the provided web services API.
WSSetDefaultPKGName("helpdesk");

WebServiceName = "HelpDeskService";
WebServicePort = "HelpDeskPort";
EndPoint = "http://helpdesk_01/webservice/";
Method = "submitTicket";
Params = { GetDate(), \n        "Netcool/Impact", \n        EventContainer.Node, \n        EventContainer.Summary }; 

Results = WSGlobalDL(WebServiceName, WebServicePort, EndPoint, Method, Params); 

// Print results of the web services call to the policy log. 
Log("Result of web services call: " + Results);
Chapter 5. Working with policy parameters

You can define the parameters that you pass to a policy when you run it either using the GUI or the nci_trigger script.

You can use these parameters when testing a policy in the GUI or when you want to automate a policy by external means from the command line (for example, using the UNIX cron tool).

Configuring policy settings in the policy editor

Use this procedure to configure the policy settings for your policy in the policy editor.

Procedure

1. In the policy editor toolbar, click the Configure Policy Settings icon to open the policy settings editor. You can create policy input and output parameters and also configure actions on the policy that relates to UI Data Provider and Event Isolation and Correlation options.

2. Click New to open the Create a New Policy Input Parameter window or the Create a New Policy Output Parameter window or the Create New policy action window as required. For more information, see “Configuring policy parameters and enabling actions”.

   Enter the information in the configuration window. Required fields are marked with an asterisk (*). If you select DirectSQL as the format, see “Creating custom schema values for output parameters” on page 42.

3. To edit an existing input or output parameter, select the check box next to the parameter and select edit in the corresponding cell of the Edit column.

4. To enable a policy to run with an UI data provider select the Enable policy for UI Data Provider Actions check box.

5. To enable a policy to run in with the Event Isolation and Correlation capabilities, select the Enable Policy for Event Isolation and Correlation Actions check box.

6. Click OK to save the changes to the parameters and close the window.

Configuring policy parameters and enabling actions

You can use the Policy settings editor to configure input and output parameters on a policy. You can also enable options for use with UI data providers and Event Isolation and Correlation features.

Procedure

1. In the Policy Input Parameters section, click New to create a policy input parameter.
   a. In the Name field, type a name to describe the parameter.
   b. In the Label field, add a label. The label is displayed in the Policy Trigger window.
   c. From the Format menu, select the format of the parameter.
   d. In the Default Value field, add a default value. This value is displayed in the Policy Trigger window.
e. In the **Description** field, add a description for the parameter.

2. In the **Policy Output Parameters** section, click **New** to create a policy output parameter.

   **Tip:** When you create multiple output parameters, remember each policy output parameter that you create generates its own data set. When you assign a data set to a widget, only those tasks that are associated with the specific output parameter are run.

   a. In the **Name** field, type a name to describe the parameter.
   b. In the **Policy Variable Name** field, add the variable name. The variable name is displayed in the Policy Trigger window.
   c. From the **Format** menu, select the format of the parameter.
   d. Click the **Schema Definition Editor** icon. If you define output parameters that use the **DirectSQL/UI Provider Datatype**, **Impact Object** or **Array of Impact Object** formats, click this icon to create the custom schema definition values. For more information, see "Creating custom schema values for output parameters."
   e. In the **Default Value** field, add a default value. This value is displayed in the Policy Trigger window.
   f. In the **Data Source Name** field, type the name of the data source that is associated with the output parameter.
   g. In the **Data Type Name** field, type the name of the data type associated with the output parameter.

3. In the **UI Data Provider Policy Related Actions** section, click **New** to create a UI Data Provider policy related action. You can use this option to enable a policy action on a widget in the console (Dashboard the Dashboard Application Services Hub) in Jazz for Service Management.

   a. In the **Name** field, add a name for the action. The name that you add displays in the widget in the console when you right-click an item in the specified widget.
   b. In the **Policy Name** menu, select the policy that you want the action to relate to.
   c. In the **Output Parameter** menu, select the output parameter that is associated with this action. If you select the **All Output Parameters** option, the action will be available for all output parameters for the current policy.

4. To enable a policy to run with an UI data provider, select the **Enable policy for UI Data Provider Actions** check box.

5. To enable a policy to run in with the Event Isolation and Correlation capabilities, select the **Enable Policy for Event Isolation and Correlation Actions** check box.

6. Click **OK** to save the changes to the parameters and close the window.

**Creating custom schema values for output parameters**

When you define output parameters that use the **DirectSQL**, **Array of Impact Object**, or **Impact Object** format in the user output parameters editor, you also must specify a name and a format for each field that is contained in the **DirectSQL**, **Array of Impact Object**, or **Impact Object** objects.
About this task

Custom schema definitions are used by Netcool/Impact to visualize data in the console and to pass values to the UI data provider and OSLC. You create the custom schemas and select the format that is based on the values for each field that is contained in the object. For example, you create a policy that contains two fields in an object:

```
O1.city="NY"
O1.ZIP=07002
```

You define the following custom schemas values for this policy:

**Table 8. Custom schema values for City**

<table>
<thead>
<tr>
<th>Field</th>
<th>Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>City</td>
</tr>
<tr>
<td>Format</td>
<td>String</td>
</tr>
</tbody>
</table>

**Table 9. Custom schema values for ZIP**

<table>
<thead>
<tr>
<th>Field</th>
<th>Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>ZIP</td>
</tr>
<tr>
<td>Format</td>
<td>Integer</td>
</tr>
</tbody>
</table>

If you use the DirectSQL policy function with the UI data provider or OSLC, you must define a custom schema value for each DirectSQL value that you use.

If you want to use the chart widget to visualize data from an Impact object or an array of Impact objects with the UI data provider and the console, you define custom schema values for the fields that are contained in the objects. The custom schemas help to create descriptors for columns in the chart during initialization. However, the custom schemas are not technically required. If you do not define values for either of these formats, the system later rediscovers each Impact object when it creates additional fields such as the key field. `UIObjectId`, or the field for the tree widget, `UITreeNodeId`. You do not need to define these values for OSLC.

Procedure

1. In the Policy Settings Editor, select DirectSQL, Impact Object, or Array of Impact Object in the Format field.

2. The system shows the Open the Schema Definition Editor icon beside the Schema Definition field. To open the editor, click the icon.

3. You can edit an existing entry or you can create a new one. To define a new entry, click New. Enter a name and select an appropriate format.

4. To mark an entry as a key field, select the check box in the Key Field column. You do not have to define the key field for Impact objects or an array of Impact objects. The system uses the `UIObjectId` as the key field instead.

5. To delete an entry, select the entry and click Delete.
Running policies with parameters in the editor

If you specified any parameters for the policy, you can run the policy with these parameters in the policy editor.

**Procedure**

1. Click the **Run with Parameters** icon to open the Policy Trigger window.

   **Note:** The fields you see depend on the policy parameters and values you specified for the policy. If you have not set a default value for a parameter you must provide it now, otherwise a NULL value will be passed.

   Output parameters are required if you want to show policy output through a UI data provider. For more information about setting parameters, see [“Configuring policy settings in the policy editor” on page 41](#).

2. Click **Execute** to run the policy with parameters.

Running a policy using the nci_trigger script

Use this procedure to run a policy using the nci_trigger script.

**Procedure**

To run the policy, you start nci_trigger from the command line, as in the following example.

In this example, the name of the policy is POLICY_01. The value of Value1 is Testing1, and the value of Value2 is Testing2.

```bash
nci_trigger NCI impactadmin/netcool POLICY_01 Value1 Testing1 Value2 Testing2
```

In the following example, Value1 and Value2 are the input parameters that the policy handles.

```c
// Value1 and Value2 are passed to the policy
Value3 = EventContainer.Value1 + " " + EventContainer.Value2;
Log(Value3);
```

Printing to the policy log

Printing messages to the policy log is one of the most useful capabilities of Netcool/Impact when it comes to testing and debugging policies.

You print messages to the policy log using the `Log` function. The `Log` function takes the message you want to print as its input parameter.

This example is a version of the classic "Hello, World!" program used to teach developers how to program in the C programming language. In the C version, you print Hello, World! to the standard output. You are not permitted to access the standard output stream using the policy language but you can print the message to the policy log.

The policy, which consists of a single line, is as follows.

```c
Log("Hello, World!");
```

Here, you simply call the `Log` function and pass the string `Hello, World!` as an input parameter. As in programming languages like C/C+ and Java, you enclose string literals in double quotation marks.
When you run the policy, it prints the following message to the policy log:

Hello, World!

**Working with chained policies**

Policy chaining is a feature where you chain multiple policies to run together sequentially when an event reader service triggers them.

Policies are run in series, rather than simultaneously, and each policy in the chain inherits the policy context from the previously run policy. This means that variables whose values were assigned in a previous policy in the chain maintain their values in subsequent policies.

When the event reader retrieves an event from the Netcool/OMNIbus ObjectServer, it compares the event data to each defined event mapping in the service configuration. If the event matches multiple chained mappings, it runs each of the mapped policies in sequence as they appear in the event mapping window.

**Chaining policies**

Follow this procedure to chain your policies.

**Procedure**

1. Click the **Services** link to open the **Services** tab.
2. Double click the name of the event reader service that you want to use to run the chained policies.
3. Select the **Event Mapping** tab in the window that opens.
4. For each policy in the chain, create an event mapping that associates a restriction filter with a policy name.
   
   When you create the event mapping, select the **Chain** option in the event mapping window.
5. Click **OK**.

**Encrypted policies**

An encrypted policy is a policy whose text content has been encrypted to a non-human readable format.

Encrypted policies can be run in the same way as non-encrypted policies.

You encrypt policies using the `nci_encryptpolicy` script. This script is located in the `$NCHOME/bin` directory and has the following syntax:

```
nci_encryptpolicy server_name
    password
    input_policy
    output_policy
```

where `server_name` is the name of the Impact Server, `password` is the encryption password, `input_policy` is the name of the policy you want to encrypt, and `output_policy` is the name of the resulting encrypted policy.

Before you can run the encrypted policy on the originating Impact Server or on another server, you first import it into the system using the GUI.

To import the policy, complete the following steps:

1. Open the **Policies** task pane in the Navigation panel.
2. Click the **Upload Local IPL File** button.
3. Click the **Browse** button in the window that opens and select the encrypted policy file.
4. Click **OK**.

Once you have imported the policy, you can run it in the same way that you run any other policy.

**Line continuation character**

The line continuation character in IPL and JavaScript is the backslash (\).

You use this character to indicate that the code on a subsequent line is a continuation of the current statement. The line continuation character helps you format your policies so that they are easier to read and maintain.

**Note:** You cannot use the line continuation character inside a string literal as specified with enclosing quotation marks. This usage is not allowed and an error is reported during processing.

The following example shows the use of the line continuation character:

```
Log("You can use the line continuation character \" + \"
  "to format very long statements in a policy.");
```

**Code commenting**

IPL and JavaScript support both C-style comment blocks and C++-style single-line code commenting.

Comment blocks are single or multi-line blocks of comments enclosed by the forward slash (/) and asterisk ( *) characters. The following example shows comment blocks.

```
/*
 This is a single-line comment block */

/*
 This is a multi-line comment block */
```

Single-line comments are prefixed by two forward slash characters.

The following example shows single-line comments

```
// These
// are
// single-line
// comments
```

**Executing external commands**

External command execution is the process of running external commands, scripts, and applications from within a policy. You can use the JRExec server or the command and response feature to run external commands.

The JRExec server is a runnable component of Netcool/Impact that you can use to run external commands on the system where the Netcool/Impact server is located. Command and response is a more advanced feature that you can use to run interactive and non-interactive programs on both local and remote systems.

You can run any type of external command that can be started from a command line. Including operating system commands, shell scripts, and many other types of applications.
Running commands using the JRExec server

To run a command using the JRExec server, you call the JRExecAction function and pass the name of the command and any command-line arguments as input parameters in a policy.

You can also pass a value that specifies whether you want the JRExec server to wait for the command to be completed before executing any other commands, or to continue processing without waiting.

The following example shows how to run an external command using the JRExec server. In this example, you send a page to an administrator using a paging application named pageit that is installed in the $NCHOME/bin directory on the system. The pageit application takes the phone number of the person paged and the return contact number as command-line arguments. In this application, the JRExec server waits for the application to finish before continuing to process any other commands. Run this command in the terminal: ./nci_jrexecc see Starting the JRExec server in the Configuring servers section of the documentation.

// Call JRExecAction and pass the command string and // other parameters

Command = "/opt/pager/bin/pageit";
Args = {"2125551212", "2126353131"};
ExecuteOnQueue = False;
Timeout = 60;

JRExecAction(Command, Args, ExecuteOnQueue, Timeout);

Using CommandResponse

Command and response is an advanced feature that lets you run interactive and non-interactive programs on both local and remote systems.

You can invoke this feature within a policy using the CommandResponse function. For more information about the syntax of the function, see CommandResponse in the Policy Reference Guide.
Chapter 6. Using local transactions in a policy

You use local transactions in a policy if you want to use more than one SQL operation to be treated as a single unit of work.

The following code is a typical template of a policy that uses local transactions. In this policy, SQL_Operation_1() will be executed first. As soon as the operation is completed successfully, the changes will be committed to the database. When the BeginTransaction() method is executed, all SQL operations following it will be executed using the same transaction. As a result any changes that they have made will not be committed to the database until the operations are executed successfully and CommitTransaction() is executed.

```java
Handle com.micromuse.response.action.TransactionException {
    Log("Transaction Failed " + ErrorMessage);
    RollbackTransaction();
}

SQL_Operation_1();
....
BeginTransaction();
...
SQL_Operation_2();
SQL_Operation_3();
....
CommitTransaction();
...
SQL_Operation_4();
```

Local transactions template

Here is a typical template of a policy that uses local transactions.

In this policy, SQL_Operation_1() will be executed first. As soon as the operation is completed successfully, the changes will be committed to the database. When the BeginTransaction() method is executed, all SQL operations following it will be executed using the same transaction. As a result any changes that they have made will not be committed to the database until the operations are executed successfully and CommitTransaction() is executed.

```java
Handle com.micromuse.response.action.TransactionException {
    Log("Transaction Failed " + ErrorMessage);
    RollbackTransaction();
}

SQL_Operation_1();
....
BeginTransaction();
...
SQL_Operation_2();
SQL_Operation_3();
....
CommitTransaction();
...
SQL_Operation_4();
```

The following examples examine various scenarios that may result after running the policy in the template.
**SQL_Operation_2() and SQL_Operation_3() is executed successfully**

In this scenario when the thread reaches the `CommitTransaction()` function, it will commit both SQL operations to the database and then enable the Auto Commit functionality so that as soon as `SQL_Operation_4()` is gets executed, the changes get committed to the database.

**SQL_Operation_2() fails and SQL_Operation_3() is executed successfully**

When `SQL_Operation_2()` fails, an exception will be thrown that gets caught by the Handle block. In the Handle block, the `RollbackTransaction()` function is called which rolls back any changes done by `SQL_Operation_2()`. When the Handle block is finished, the execution goes back to the policy right after `SQL_Operation_2()`, which is `SQL_Operation_3()`. This SQL Operation will get executed but since it is being executed after a rollback has occurred, the changes will not get committed to the database. When the thread reaches the `CommitTransaction()` function, it would not commit anything to the database since a rollback had occurred. The only operation done by `CommitTransaction()` would be to enable auto-commit for any SQL operation following it. When the thread executes `SQL_Operation_4()`, any changes done will be committed to the database as soon the operation is completed as Netcool/Impact will be non-transactional after `CommitTransaction()`.

**SQL_Operation_2() is executed successfully and SQL_Operation_3() fails**

Let us assume that after `BeginTransaction()`, the `SQL_Operation_2()` gets executed successfully. The changes will not get committed to the database until all the operations between `BeginTransaction()` and `CommitTransaction()` get executed successfully. In this scenario, we have assumed that `SQL_Operation_3()` fails. As a result an exception will be thrown that will send the policy execution inside the Handle block where the `RollbackTransaction()` method gets called. This function will roll back any changes done since `BeginTransaction()` and, when the Handle block is completed, the execution is returned to the policy statement following `SQL_Operation_3()`. When the policy executes the `CommitTransaction()` method, it will not commit anything to the database because rollback has occurred. The only operation done by `CommitTransaction()` would be to enable auto-commit for any SQL operations following it. When the thread executes `SQL_Operation_4()`, any changes done will be committed to the database as soon the operation is completed as Netcool/Impact will be non-transactional after `CommitTransaction()`.

**Both SQL_Operation_2() and SQL_Operation_3() fails**

When `SQL_Operation_2()` fails, the policy execution will enter the Handle block where `RollbackTransaction()` is executed, it will roll back any changes made since `BeginTransaction()` and transfer the policy execution to the statement following `SQL_Operation_2()`. When `SQL_Operation_3()` gets executed and fails, the `RollbackTransaction()` will be executed again in the Handle block and the same process repeats. When the policy execution reaches the `CommitTransaction()` function, it will not commit any changes and will enable auto-commit for any SQL operations following it.
Local transactions best practices

Here are some practical tips on the usage of local transactions.

- Local transactions should ideally combine SQL operations to a single database. Even though nothing is stopping you from using different data sources between BeginTransaction() and CommitTransaction(), you are recommended not to do that.

- Avoid doing operations inside the transaction block that are not related to the actual SQL operations.

- Do not rely solely on com.micromuse.response.action.TransactionException to be thrown for every possible failure in the transaction block. One way to get around this would be to also handle the general java.lang.Exception and call RollbackTransaction() in it. For example:

```java
Handle com.micromuse.response.action.TransactionException {
    Log("Transaction Failed " + ErrorMessage);
    RollbackTransaction();
}

Handle java.lang.Exception {
    Log("Policy Execution Failed" + ErrorMessage);
    // If there is no transaction, this won't do anything
    RollbackTransaction();
}
```
Chapter 7. Calling stored procedures from within a policy

You can call database stored procedures from within a policy using the CallStoredProcedure function.

You can use this function with Sybase, Microsoft SQL Server, DB2SQL, and Oracle databases.

Oracle stored procedures

The CallStoredProcedure function works with Oracle data sources.

The CallStoredProcedure function works in two ways:

- With automatic schema discovery
- Without automatic schema discovery

Automatic schema discovery is a feature of CallStoredProcedure with which Netcool/Impact automatically discovers the schema of the procedure before sending the procedure request to the database. This is the default behavior of the application.

Automatic schema discovery makes it easier to write stored procedure policies. This is because you do not have to explicitly declare the procedure schema in the policy body before you call the CallStoredProcedure function. However, because two database requests are made every time it calls the function is called, running the policy with automatic schema discovery creates an additional performance load on the database. This can also slow the performance, because the policy engine waits for the Oracle database to respond to both requests in sequence before continuing on to process the rest of the policy.

To avoid the extra processing load on the database and to minimize effects on performance, you can disable automatic schema discovery and explicitly specify the stored procedure schema in the body of the policy.

If you use multiple procedures that have the same name but are stored in different packages, you must disable automatic schema discovery. If automatic schema discovery is not disabled, the Netcool/Impact cannot resolve the procedure correctly and it displays an error. After you disable the automatic schema discovery, define the procedure name argument as <packagename>.<procedurename> in the policy where you invoke the CallStoredProcedure function.

To disable schema discovery globally for all policies, set the following property in the server properties file: impact.storedprocedure.discoverprocedureschema=false

The server properties file is named servername_server.props, where servername is the name of the Impact Server. The default value for this property is true. You can also disable schema discovery on a per-policy basis.

Writing policies with automatic schema discovery

You can call certain types of stored procedures from within a policy.

- Procedures that return scalar values.
• Procedures that return arrays. Oracle allows stored procedures to return an array of values as an output parameter. Typically, this array represents a row and each element in the array represents a row field.

• Procedures that return cursors. Oracle allows stored procedures to return a cursor as an output parameter. This cursor is an array of arrays that typically represents a set of rows in a database.

Calling procedures that return scalar values
Use this procedure to call an Oracle stored procedure that returns scalar values as output parameters.

Procedure

• Create a new context called Sp_Parameter that is used to store input and output variables for the procedure.

• Populate the Sp_Parameter member variables with the input parameter values for the stored procedure.

  This example shows how to populate the Sp_Parameter member variables with values for the Hostname and Location input parameters in the stored procedure.

  ```
  Sp_Parameter.Hostname = "192.168.1.25";
  Sp_Parameter.Location = "New York";
  ```

• Call the CallStoredProcedure function and pass the name of the data source, the name of the stored procedure, and Sp_Parameter.

  Make sure that you use the data source name, not the name of a data type. In addition, the name of the stored procedure is case sensitive and has to appear exactly as it is defined in the database.

  The following example shows how to call the CallStoredProcedure function:

  ```
  CallStoredProcedure("ORA_01", "GetHostnameByIP", Sp_Parameter);
  ```

Creating the Sp_Parameter context:

Before you call any stored procedure you need to create a new context called Sp_Parameter.

Procedure

To create a new Sp_Parameter context call the NewObject function as follows.

```
Sp_Parameter = NewObject();
``` 

Populating the Sp_Parameter member variables:

Use these guidelines to populate the Sp_Parameter member variables.

Make sure that the name of the member variables is exactly the same as those of the input parameters in the procedure. Specify a value for each parameter in the stored procedure, even if you want to accept the default.

The following example shows how to populate the Sp_Parameter member variables with values for the Hostname and Location input parameters in the stored procedure.

```
Sp_Parameter.Hostname = "192.168.1.25";
Sp_Parameter.Location = "New York";
```
The following example shows how to populate the Sp_Parameter member variables with values for the CustType and Location input parameters in the stored procedure.

```java
Sp_Parameter.CustType = "Premium";
Sp_Parameter.Location = "New York";
```

Use this code snippet to populate the Sp_Parameter member variable with a new context that will contain an output parameter returned from the stored procedure. In this example, the output parameter is called Name and contains an Oracle VARRAY.

```java
Sp_Parameter.Name = NewObject();
Sp_Parameter.Name.type = "CUST";
```

The following example shows how to populate the Sp_Parameter member variables with values for the IPAddress and Location input parameters.

```java
Sp_Parameter.IPAddress = "192.168.1.25";
Sp_Parameter.Location = "Singapore";
```

**Calling the CallStoredProcedure function:**

When you call CallStoredProcedure, the function calls the procedure in the specified data source and returns the output parameters as member variables in Sp_Parameter.

You pass it the name of the data source, the name of the stored procedure, and the Sp_Parameter variable. Names of the member variables correspond to the names of the output parameters. Make sure that you use the data source name, not the name of a data type.

The following examples demonstrate how to call the CallStoredProcedure function.

In this example, the name of the data source is Oracle_01 and the name of the stored procedure is GetCustomerByID.

```java
Call StoredProcedure("Oracle_01", "GetCustomerByID", Sp_Parameter);
```

In this example, the name of the data source is SYB_03 and the name of the stored procedure is GetCustomersByLocation. The results of the stored procedure are assigned to the MyReturn variable.

```java
MyReturn = CallStoredProcedure("SYB_03", "GetCustomersByLocation", Sp_Parameter);
```

In this example, the name of the data source is DB2DS and the name of the stored procedure is GetHostnameByIP.

```java
Call StoredProcedure('DB2DS', 'GetHostnameByIP', Sp_Parameter);
```

**Example of an Oracle stored procedure that returns a scalar value:**

The following example shows how to call an Oracle stored procedure that returns a scalar value.

**Procedure**

In this example, you call a procedure named GetHostnameByIP. This procedure has one input parameter named IPAddress and one output parameter named Hostname. The example calls the stored procedure and then prints the Hostname value to the policy log.
Create the Sp_Parameter context

Sp_Parameter = NewObject();

Populate the Sp_Parameter member variables with input parameter values

Sp_Parameter.IPAddress = "192.168.1.25";

Call CallStoredProcedure and pass the name of the data source, the name of the stored procedure and Sp_Parameter

DataSource = "ORA_01";
StoredProc = "GetHostnameByIP";

CallStoredProcedure(DataSource, StoredProc, Sp_Parameter);

Print the value of the Hostname output parameter to the policy log

Log(Sp_Parameter.Hostname);

The next example shows a shorter version:

Sp_Parameter = NewObject();
Sp_Parameter.IPAddress = "192.168.1.25";
CallStoredProcedure("ORA_01", "GetHostnameByIP", Sp_Parameter);
Log(Sp_Parameter.Hostname);

Calling procedures that return an array

Use this procedure to call an Oracle stored procedure that returns an array as an output parameter.

Procedure

- Create a new context called Sp_Parameter that is used to store input and output variables for the procedure.
- Populate the Sp_Parameter member variables with the input parameter values for the stored procedure.
  The following example shows how to populate the Sp_Parameter member variables with values for the CustType and Location input parameters in the stored procedure.

  Sp_Parameter.CustType = "Premium";
  Sp_Parameter.Location = "New York";

  The following example shows how to populate the Sp_Parameter member variable with a new context that will contain an output parameter returned from the stored procedure. In this example, the output parameter is called Name and contains an Oracle VARRAY.

  Sp_Parameter.Name = NewObject();
  Sp_Parameter.Name.type = "CUST";

- Call the CallStoredProcedure function and pass the name of the data source, the name of the stored procedure, and Sp_Parameter.
  Make sure that you use the data source name, not the name of a data type. In addition, the name of the stored procedure is case sensitive and has to appear exactly as it is defined in the database.

  The following example shows how to call the CallStoredProcedure function:

  CallStoredProcedure("ORA_01", "GetCustomersByLocation", Sp_Parameter);

- You can now handle the returned array by accessing a member of the Sp_Parameter context that uses the same name as the underlying array type in the data source.
Creating the Sp_Parameter context:

Before you call any stored procedure you need to create a new context called Sp_Parameter.

Procedure

To create a new Sp_Parameter context call the NewObject function as follows.

```
Sp_Parameter = NewObject();
```

Populating the Sp_Parameter member variables:

Use these guidelines to populate the Sp_Parameter member variables.

Make sure that the name of the member variables is exactly the same as those of the input parameters in the procedure. Specify a value for each parameter in the stored procedure, even if you want to accept the default.

The following example shows how to populate the Sp_Parameter member variables with values for the Hostname and Location input parameters in the stored procedure.

```
Sp_Parameter.Hostname = "192.168.1.25";
Sp_Parameter.Location = "New York";
```

The following example shows how to populate the Sp_Parameter member variables with values for the CustType and Location input parameters in the stored procedure.

```
Sp_Parameter.CustType = "Premium";
Sp_Parameter.Location = "New York";
```

Use this code snippet to populate the Sp_Parameter member variable with a new context that will contain an output parameter returned from the stored procedure. In this example, the output parameter is called Name and contains an Oracle VARRAY.

```
Sp_Parameter.Name = NewObject();
Sp_Parameter.Name.type = "CUST";
```

The following example shows how to populate the Sp_Parameter member variables with values for the IPAddress and Location input parameters.

```
Sp_Parameter.IPAddress = "192.168.1.25";
Sp_Parameter.Location = "Singapore";
```

Calling the CallStoredProcedure function:

When you call CallStoredProcedure, the function calls the procedure in the specified data source and returns the output parameters as member variables in Sp_Parameter.

You pass it the name of the data source, the name of the stored procedure, and the Sp_Parameter variable. Names of the member variables correspond to the names of the output parameters. Make sure that you use the data source name, not the name of a data type.

The following examples demonstrate how to call the CallStoredProcedure function.

In this example, the name of the data source is Oracle_01 and the name of the stored procedure is GetCustomerByID.
CallStoredProcedure("Oracle_01", "GetCustomerByID", Sp_Parameter);

In this example, the name of the data source is SYB_03 and the name of the stored procedure is GetCustomersByLocation. The results of the stored procedure are assigned to the MyReturn variable.
MyReturn = CallStoredProcedure("SYB_03", "GetCustomersByLocation", Sp_Parameter);

In this example, the name of the data source is DB2DS and the name of the stored procedure is GetHostnameByIP.
CallStoredProcedure('DB2DS', 'GetHostnameByIP', Sp_Parameter);

Handling the returned array:

When you call the stored procedure, the underlying data source returns an array.

Procedure

Netcool/Impact assigns this array as a member variable in the Sp_Parameter context. The name of the member variable is the same name as the user-defined array data type returned from the underlying data source. The following example shows how to handle the array returned by a stored procedure. In this example, the name of the array data type in the Oracle database is CUST.
CallStoredProcedure("ORA_01", "GetCustomersByLocation", Sp_Parameter);

Log("The name of the Customer is " + Sp_Parameter.Name.elements[0]);
Log("The location of the Customer is " + Sp_Parameter.Name.elements[1]);

Example of an Oracle stored procedure that returns an array:

The following complete example shows how to call an Oracle stored procedure that returns an array as an output parameter.

Procedure

In this example, the name of the data source is ORA_01, and the name of the stored procedure is GetCustomerByLocation. The stored procedure returns a CUST array with two fields. The first field stores the customer's name. The second field stores the customer's location.
// Create the Sp_Parameter context.
Sp_Parameter = NewObject();

// Populate the Sp_Parameter member variables with values
// for the input parameters of the stored procedure
Sp_Parameter.CustType = "Premium";
Sp_Parameter.Location = "New York";

// Create an Sp_Parameter member variable that stores
// returned VARRAY
Sp_Parameter.Name = NewObject();
Sp_Parameter.Name.Type = "CUST";

// Call CallStoredProcedure and pass the name of the data source,
// the name of the stored procedure and the Sp_Parameter context.
DataSource = "ORA_01";
ProcName = "GetCustomerByLocation";

CallStoredProcedure(DataSource, ProcName, Sp_Parameter);

// Print the name and location of the customer to the policy log.
Log("The name of the Customer is " + Sp_Parameter.Name.elements[0]);
Log("The location of the Customer is " + Sp_Parameter.Name.elements[1]);

**Calling procedures that return a cursor**

Use this procedure to call an Oracle stored procedure that returns a cursor as an output parameter.

**Procedure**

- Create a new context called Sp_Parameter that is used to store input and output variables for the procedure.
- Populate the Sp_Parameter member variables with the input parameter values for the stored procedure.
  
  The following example shows how to populate the Sp_Parameter member variables with values for the CustType and Location input parameters in the stored procedure.
  
  ```
  Sp_Parameter.CustType = "Basic";
  Sp_Parameter.Location = "Shanghai";
  ```

- Create the output parameter context using NewObject
- Call the CallStoredProcedure function and pass the name of the data source, the name of the stored procedure, and Sp_Parameter.
  
  The following example shows how to call the CallStoredProcedure function:
  
  ```
  CallStoredProcedure("ORA_01", "GetCustomersByLocation", Sp_Parameter);
  ```

- You can now handle the returned cursor by accessing the output parameter context that you created.

**Creating the Sp_Parameter context:**

Before you call any stored procedure you need to create a new context called Sp_Parameter.

**Procedure**

To create a new Sp_Parameter context call the NewObject function as follows.

```
Sp_Parameter = NewObject();
```

**Populating the Sp_Parameter member variables:**

Use these guidelines to populate the Sp_Parameter member variables.

Make sure that the name of the member variables is exactly the same as those of the input parameters in the procedure. Specify a value for each parameter in the stored procedure, even if you want to accept the default.

The following example shows how to populate the Sp_Parameter member variables with values for the Hostname and Location input parameters in the stored procedure.

```
Sp_Parameter.Hostname = "192.168.1.25";
Sp_Parameter.Location = "New York";
```
The following example shows how to populate the Sp_Parameter member variables with values for the CustType and Location input parameters in the stored procedure.

```csharp
Sp_Parameter.CustType = "Premium";
Sp_Parameter.Location = "New York";
```

Use this code snippet to populate the Sp_Parameter member variable with a new context that will contain an output parameter returned from the stored procedure. In this example, the output parameter is called Name and contains an Oracle VARRAY.

```csharp
Sp_Parameter.Name = NewObject();
Sp_Parameter.Name.type = "CUST";
```

The following example shows how to populate the Sp_Parameter member variables with values for the IPAddress and Location input parameters.

```csharp
Sp_Parameter.IPAddress = "192.168.1.25";
Sp_Parameter.Location = "Singapore";
```

**Calling the CallStoredProcedure function:**

When you call CallStoredProcedure, the function calls the procedure in the specified data source and returns the output parameters as member variables in Sp_Parameter.

You pass it the name of the data source, the name of the stored procedure, and the Sp_Parameter variable. Names of the member variables correspond to the names of the output parameters. Make sure that you use the data source name, not the name of a data type.

The following examples demonstrate how to call the CallStoredProcedure function.

In this example, the name of the data source is Oracle_01 and the name of the stored procedure is GetCustomerByID.

```csharp
CallStoredProcedure("Oracle_01", "GetCustomerByID", Sp_Parameter);
```

In this example, the name of the data source is SYB_03 and the name of the stored procedure is GetCustomersByLocation. The results of the stored procedure are assigned to the MyReturn variable.

```csharp
MyReturn = CallStoredProcedure("SYB_03", "GetCustomersByLocation", Sp_Parameter);
```

In this example, the name of the data source is DB2DS and the name of the stored procedure is GetHostnameByIP.

```csharp
CallStoredProcedure("DB2DS", 'GetHostnameByIP', Sp_Parameter);
```

**Handling the returned cursor:**

When you call the stored procedure, the underlying data source returns the cursor.

**Procedure**

Netcool/Impact converts the cursor as an array of arrays, where the first set of arrays represents the rows returned in the cursor, and the second set of arrays represents the fields in the rows. The following example shows how to handle the cursor returned from a stored procedure.
Count = Length(Sp_Parameter.CUSTOMERS.elements);

While (Count > 0) {
    Index = Count - 1;
    Elements = Sp_Parameter.CUSTOMERS.elements[Index];
    Log("Customer name is: " + Elements.Name);
    Log("Customer ID is: " + Elements.ID);
    Count = Count - 1;
}

Example of an Oracle stored procedure that returns a cursor:

The following complete example shows how to call an Oracle stored procedure that returns a cursor as an output parameter.

Procedure

In this example, the name of the data source is ORA_02 and the name of the stored procedure is GetCustomerByLocation. The stored procedure returns a cursor that consists of multiple rows from the database. Each row has multiple fields, among which are Name and ID.

// Create the Sp_Parameter context.
Sp_Parameter = NewObject();

// Populate the Sp_Parameter member variables with // values to pass as input parameters to the stored // procedure.
Sp_Parameter.CustType = "Basic";
Sp_Parameter.Location = "Shanghai";

// Create the output parameter context
Sp_Parameter.CUSTOMERS = NewObject();

// Call CallStoredProcedure and pass the name of the // data source, the stored procedure name and the // Sp_Parameter context.
DataSource = "ORA_02";
ProcName = "GetCustomersByLocation";
CallStoredProcedure(DataSource, ProcName, Sp_Parameter);

// Iterate through the arrays in the output parameter // context and print out the values of the Name and // ID fields.
Count = Length(Sp_Parameter.CUSTOMERS);

While (Count > 0) {
    Index = Count - 1;
    Elements = Sp_Parameter.CUSTOMERS.elements[Index];
    Log("Customer name is: " + Elements.Name);
    Log("Customer ID is: " + Elements.ID);
    Count = Count - 1;
}

Writing policies without automatic schema discovery

Follow these steps if you want to write policies without automatic schema discovery.
**Procedure**

- Disable schema discovery globally for all policies.
  
  You do this by setting the
  
  `impact.storedprocedure.discoverprocedureschema=false` property in the server properties file:
  
  You can also disable schema discovery on a per-policy basis.

- Create a new context called Sp_Parameter that is used to store input and output variables for the procedure.

- Create a new context for each parameter that you intend to pass to the procedure and assign this context to an Sp_Parameter member variable.

- Optional: Create a return parameter context.

  If you are calling a stored function that has a return parameter, you specify it as the first parameter in the Sp_Parameter context.

- Optional: Set the DiscoverProcedureSchema variable.

  If you do not want to globally disable schema discovery, you can disable it on a per policy basis.

- Call the CallStoredProcedure function and pass the name of the data source, the name of the stored procedure, and Sp_Parameter.

  The following example shows how to call the CallStoredProcedure function:
  
  ```java
  CallStoredProcedure("Oracle_01", "GetCustomerByID", Sp_Parameter);
  ```

**Disabling schema discovery globally**

Use this procedure to disable schema discovery globally for all policies.

**About this task**

To avoid extra processing load on the database and to minimize performance issues, you can disable automatic schema discovery.

If you use multiple procedures that have the same name but that are stored in different packages, you must disable automatic discovery. If you do not disable automatic schema discovery, Netcool/Impact displays an error.

In both cases, you can specify the stored procedure schema directly in the policy as an alternative.

**Procedure**

Set the following property in the server properties file:

`impact.storedprocedure.discoverprocedureschema=false`

The server properties file is named `servername_server.props`, where `servername` is the name of the Impact Server. The default value for this property is `true`.

**Creating the Sp_Parameter context**

Before you call any stored procedure you need to create a new context called Sp_Parameter.

**Procedure**

To create a new Sp_Parameter context call the NewObject function as follows.

```java
Sp_Parameter = NewObject();
```
Creating the parameter contexts
After you have created Sp_Parameter context, you create a new context for each parameter that you intend to pass to the procedure and assign this context to an Sp_Parameter member variable.

Procedure
To create a new parameter context, you call the NewObject function and assign it as a member of Sp_Parameter as follows:

\[ \text{Sp_Parameter}["1"] = \text{NewObject}(); \]

where the name of the member variable is an index value starting with 1. Specify the index number as a string. Specify the parameters in the order in which they appear in the stored procedure call.

After you have created a parameter context, you then assign it a set of member variables that specify the parameter name, type, type name, direction, and value. Table 10 shows the valid parameter types and type names.

Table 10. Parameter types and type names

<table>
<thead>
<tr>
<th>Type</th>
<th>Typename</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Decimal, Number, Integer, Numeric, Smallint, Float</td>
</tr>
<tr>
<td>12</td>
<td>Varchar, Varchar2, String</td>
</tr>
<tr>
<td>93</td>
<td>Date</td>
</tr>
<tr>
<td>1111</td>
<td>Clob, Varray, Ref Cursor</td>
</tr>
</tbody>
</table>

You can assign the members variables in the parameter context as follows:

\[
\begin{align*}
\text{Sp_Parameter}["1"].\text{name} &= \text{"FirstName"}; \\
\text{Sp_Parameter}["1"].\text{type} &= 12; \\
\text{Sp_Parameter}["1"].\text{typename} &= \text{"String"}; \\
\text{Sp_Parameter}["1"].\text{direction} &= \text{"INOUT"}; \\
\text{Sp_Parameter}["2"].\text{value} &= \text{"Muse"};
\end{align*}
\]

Creating a return parameter context
If you are calling a stored function that has a return parameter, you specify it as the first parameter in the Sp_Parameter context.

Procedure
You specify the name and direction of this parameter as RETURN and the value as an empty string. Identify the return parameter with an index value of 1.

You can create the return parameter context and assign its member variables as follows:

\[
\begin{align*}
\text{Sp_Parameter}["1"] &= \text{NewObject}(); \\
\text{Sp_Parameter}["1"].\text{name} &= \text{"RETURN"}; \\
\text{Sp_Parameter}["1"].\text{type} &= 3; \\
\text{Sp_Parameter}["1"].\text{typename} &= \text{"Integer"}; \\
\text{Sp_Parameter}["1"].\text{direction} &= \text{"RETURN"}; \\
\text{Sp_Parameter}["1"].\text{value} &= \"\";
\end{align*}
\]

Setting the DiscoverProcedureSchema variable
Use this procedure to set the DiscoverProcedureSchema variable.
Procedure

If you do not want to globally disable schema discovery as described previously in this section, you can disable it on a per policy basis by setting the DiscoverProcedureSchema variable to false as follows:

```
DiscoverProcedureSchema = false;
```

**Calling the CallStoredProcedure function**

When you call CallStoredProcedure, the function calls the procedure in the specified data source and returns the output parameters as member variables in `Sp_Parameter`.

You pass it the name of the data source, the name of the stored procedure, and the `Sp_Parameter` variable. Names of the member variables correspond to the names of the output parameters. Make sure that you use the data source name, not the name of a data type.

The following examples demonstrate how to call the CallStoredProcedure function.

In this example, the name of the data source is `Oracle_01` and the name of the stored procedure is `GetCustomerByID`.

```
CallStoredProcedure("Oracle_01", "GetCustomerByID", Sp_Parameter);
```

In this example, the name of the data source is `SYB_03` and the name of the stored procedure is `GetCustomersByLocation`. The results of the stored procedure are assigned to the `MyReturn` variable.

```
MyReturn = CallStoredProcedure("SYB_03", "GetCustomersByLocation", Sp_Parameter);
```

In this example, the name of the data source is `DB2DS` and the name of the stored procedure is `GetHostnameByIP`.

```
CallStoredProcedure('DB2DS', 'GetHostnameByIP', Sp_Parameter);
```

**Example of specifying schemas in a policy**

The following examples show how to explicitly specify schemas in a policy when calling a variety of different stored procedure types.

**Example**

```
log("-------------Test standalone procedure combine_and_format_names------")
Sp_Name="combine_and_format_names"
Sp_Parameter = NewObject()
Sp_Parameter["1"] = NewObject()
Sp_Parameter["1"].name = "firstName"
Sp_Parameter["1"].type = 12;
Sp_Parameter["1"].typename = "String"
Sp_Parameter["1"].direction = "INOUT"
Sp_Parameter["1"].value = "Micro"
Sp_Parameter["2"] = NewObject()
Sp_Parameter["2"].name = "lastName"
Sp_Parameter["2"].type = 12;
Sp_Parameter["2"].typename = "String"
Sp_Parameter["2"].direction = "INOUT"
Sp_Parameter["2"].value = "Muse"
Sp_Parameter["3"] = NewObject()
```
Sp_Parameter["3"].name = "fullName";
Sp_Parameter["3"].type = 12;
Sp_Parameter["3"].typename = "String";
Sp_Parameter["3"].direction = "OUT";
Sp_Parameter["3"].value = "";

Sp_Parameter["4"] = NewObject();
Sp_Parameter["4"].name = "nameFormat";
Sp_Parameter["4"].type = 12;
Sp_Parameter["4"].typename = "String";
Sp_Parameter["4"].direction = "IN";
Sp_Parameter["4"].value = "FIRST,LAST";

DiscoverProcedureSchema = false;

CallStoredProcedure('oracleOnOracle1', Sp_Name, Sp_Parameter);
log("Full name in given name format: " + Sp_Result.FULLNAME);
log("--------------End combine_and_format_names------------------");

Example

log("--------------Test function returning NUMBER-------------------------");

Sp_Name = "returnNumber";
Sp_Parameter = NewObject();

// Note: Return parameter has to be always first parameter

Sp_Parameter["1"] = NewObject();
Sp_Parameter["1"].name = "RETURN";
Sp_Parameter["1"].type = 3;
Sp_Parameter["1"].typename = "NUMBER";
Sp_Parameter["1"].direction = "RETURN";
Sp_Parameter["1"].value = "";

Sp_Parameter["2"] = NewObject();
Sp_Parameter["2"].name = "MAX_SALARY";
Sp_Parameter["2"].type = 3;
Sp_Parameter["2"].typename = "NUMBER";
Sp_Parameter["2"].direction = "IN";
Sp_Parameter["2"].value = 4;

Sp_Parameter["3"] = NewObject();
Sp_Parameter["3"].name = "factor";
Sp_Parameter["3"].type = 3;
Sp_Parameter["3"].typename = "NUMBER";
Sp_Parameter["3"].direction = "IN";
Sp_Parameter["3"].value = 2;

DiscoverProcedureSchema = false;

CallStoredProcedure('oracleOnOracle1', Sp_Name, Sp_Parameter);
log("power(max_salary, factor): " + Sp_Result.RETURN);
log("--------------End function returnNumber-------------------------");

Example

log("--------------Test procedure into package returning ARRAY------------------");

function printArray_ActionNode()
{
    log("Array type: " + Sp_Result.pbooks.type);
log("Array elements : " + Sp_Result.pbooks.elements[0]);
log("Array elements : " + Sp_Result.pbooks.elements[1]);
}

Sp_Name = "procedureAndFunction.select_into_subject1";

Sp_Parameter = NewObject();

Sp_Parameter["1"] = NewObject();
Sp_Parameter["1"].name = "psubject_id";
Sp_Parameter["1"].type = 12;
Sp_Parameter["1"].typename = "String";
Sp_Parameter["1"].direction = "IN";
Sp_Parameter["1"].value = "CS1";

Sp_Parameter["2"] = NewObject();
Sp_Parameter["2"].name = "psubject_name";
Sp_Parameter["2"].type = 12;
Sp_Parameter["2"].typename = "String";
Sp_Parameter["2"].direction = "IN";
Sp_Parameter["2"].value = "Computer Science";

Sp_Parameter["3"] = NewObject();
Sp_Parameter["3"].name = "pbooks";
Sp_Parameter["3"].type = 1111;
Sp_Parameter["3"].typename = "VARRAY";
Sp_Parameter["3"].direction = "OUT";
Sp_Parameter["3"].value = NewObject();
Sp_Parameter["3"].value.type = "BOOKLIST1";

DiscoverProcedureSchema = false;

CallStoredProcedure('oracleOnOracle1 ' , Sp_Name , Sp_Parameter);

printArray_ActionNode();

log("-------------End procedure procedureAndFunction.select_into_subject1---");

Example
log("--------Test function into package with no parameters returning ResultSet----");

log("Example -- Cursor as a OUT parameter");

function printCursorNum_ActionNode() {
    totalItems = Sp_Result.RETURN.Num;
    array = Sp_Result.RETURN.elements;
    log("Total Number of elements in cursor: " + totalItems);
    getEachCursorElement_ActionNode(totalItems, array);
}

function getEachCursorElement_ActionNode(totalItems, array) {
    log("Printing Cursor Values");
    item = array[index];
    log("emp id: " + item.EMP_ID); // column names have to be UpperCase /
    log("emp Name: " + item.EMP_NAME);
    index=index+1;
    runFunction0 = false;
    if (index < totalItems) {
        runFunction0=true;
    } else {
        printCursorNum_ActionNode();
    }
}

Sp_Name = "procedureAndFunction.returnResultSet";
Sp_Parameter = NewObject();
Sp_Parameter["1"] = NewObject();
Sp_Parameter["1"].name = "RETURN";
Sp_Parameter["1"].type = 1111;
Sp_Parameter["1"].typename = "REF CURSOR";
Sp_Parameter["1"].direction = "RETURN";
Sp_Parameter["1"].value = "";

index = 0;
DiscoverProcedureSchema = false;
CallStoredProcedure('oracleOnOracle1', Sp_Name, Sp_Parameter);

printCursorNum_ActionNode();

log("-----End function procedureAndFunction.returnResultSet-------------------");

Example

log("-----DATE support-----------------------------------------------------------");

function printCursorNum_ActionNode() {
    totalItems = Sp_Result.RETURN.Num;
    array = Sp_Result.RETURN.elements;
    log("PP_SP Total Number of elements in cursor : " + totalItems);
    getEachCursorElement_ActionNode(array);
}

function getEachCursorElement_ActionNode(array) {
    item = array[index];
    log("StartTime: " + item.STARTTIME); // column names have to be UpperCase
    log("applicationName : " + item.APPLICATIONNAME);
    log("interfaceName : " + item.INTERFACENAME);
    index=index+1;
    runFunction0 = false;
    if (index < totalItems) {
        runFunction0=true;
    }
    if ( runFunction0 = true ) {  
        printCursorNum_ActionNode();
    }
}

Sp_Name = "procedureAndFunction.supportdate";
Sp_Parameter = NewObject();

/// Following three date formats are supported /
///Sp_Parameter.STARTTIME_IN = "12-MAY-2003";
///Sp_Parameter.STARTTIME_IN = "2003-05-12";
///Sp_Parameter.STARTTIME_IN = "2003-05-12 02:03:04.5";

Sp_Parameter["1"] = NewObject();
Sp_Parameter["1"].name = "RETURN";
Sp_Parameter["1"].type = 1111;
Sp_Parameter["1"].typename = "REF CURSOR";
Sp_Parameter["1"].direction = "RETURN";
Sp_Parameter["1"].value = "";

Sp_Parameter["2"] = NewObject();
Sp_Parameter["2"].name = "STARTTIME_IN";
Sp_Parameter["2"].type = 93;
Sp_Parameter["2"].typename = "DATE";
Sybase and Microsoft SQL Server stored procedures

You can use CallStoredProcedure to call the following types of procedures.

- Procedures that return a single value
- Procedures that return database rows

Automatic schema discovery is always used when Sybase or Microsoft SQL Server stored procedures are called. Unlike with Oracle stored procedures, you cannot disable automatic schema discovery.

Calling procedures that return a single value

Use this procedure to call a Sybase stored procedure that returns a single value.

Procedure

- Create a new context called Sp_Parameter that is used to store input and output variables for the procedure.
- Populate the Sp_Parameter member variables with the input parameter values for the stored procedure.
  
  The following example shows how to populate the Sp_Parameter member variables with values for the IPAddress and Location input parameters:
  
  ```
  Sp_Parameter.IPAddress = "192.168.1.25";
  Sp_Parameter.Location = "Singapore";
  ```

- Call the CallStoredProcedure function and pass the name of the data source, the name of the stored procedure, and Sp_Parameter.
  
  The following example shows how to call CallStoredProcedure. In this example, the name of the data source is SYB_01 and the name of the stored procedure is GetCustomersByLocation. The results of the stored procedure are assigned to the MyReturn variable.
  
  ```
  MyReturn = CallStoredProcedure("SYB_01", "GetCustomersByLocation", Sp_Parameter);
  ```

- You can now handle the returned value by accessing the first element of the array returned by CallStoredProcedure.

Creating the Sp_Parameter context

Before you call any stored procedure you need to create a new context called Sp_Parameter.
**Procedure**

To create a new Sp_Parameter context call the NewObject function as follows.

```javascript
Sp_Parameter = NewObject();
```

**Populating the Sp_Parameter member variables**

Use these guidelines to populate the Sp_Parameter member variables.

Make sure that the name of the member variables is exactly the same as those of the input parameters in the procedure. Specify a value for each parameter in the stored procedure, even if you want to accept the default.

The following example shows how to populate the Sp_Parameter member variables with values for the Hostname and Location input parameters in the stored procedure.

```javascript
Sp_Parameter.Hostname = "192.168.1.25";
Sp_Parameter.Location = "New York";
```

The following example shows how to populate the Sp_Parameter member variables with values for the CustType and Location input parameters in the stored procedure.

```javascript
Sp_Parameter.CustType = "Premium";
Sp_Parameter.Location = "New York";
```

Use this code snippet to populate the Sp_Parameter member variable with a new context that will contain an output parameter returned from the stored procedure. In this example, the output parameter is called Name and contains an Oracle VARRAY.

```javascript
Sp_Parameter.Name = NewObject();
Sp_Parameter.Name.type = "CUST";
```

The following example shows how to populate the Sp_Parameter member variables with values for the IPAddress and Location input parameters.

```javascript
Sp_Parameter.IPAddress = "192.168.1.25";
Sp_Parameter.Location = "Singapore";
```

**Calling the CallStoredProcedure function**

When you call CallStoredProcedure, the function calls the procedure in the specified data source and returns the output parameters as member variables in Sp_Parameter.

You pass it the name of the data source, the name of the stored procedure, and the Sp_Parameter variable. Names of the member variables correspond to the names of the output parameters. Make sure that you use the data source name, not the name of a data type.

The following examples demonstrate how to call the CallStoredProcedure function.

In this example, the name of the data source is Oracle_01 and the name of the stored procedure is GetCustomerById.

```javascript
CallStoredProcedure("Oracle_01", "GetCustomerById", Sp_Parameter);
```

In this example, the name of the data source is SYB_03 and the name of the stored procedure is GetCustomersByLocation. The results of the stored procedure are assigned to the MyReturn variable.

```javascript
MyReturn = CallStoredProcedure("SYB_03", "GetCustomersByLocation", Sp_Parameter);
```
In this example, the name of the data source is DB2DS and the name of the stored procedure is GetHostnameByIP.

```
CallStoredProcedure('DB2DS', 'GetHostnameByIP', Sp_Parameter);
```

**Handling the returned value**
When you call the stored procedure, the underlying data source returns a value as the result.

**Procedure**

Netcool/Impact assigns this value to the first element of an array and returns it from the CallStoredProcedure function.

The following example shows how to handle the results of a Sybase stored procedure that returns a single value.

```
MyReturn = CallStoredProcedure("SYB_01", "GetNodeByIPAddress", Sp_Parameter);
Log("The value returned by the procedure is: " + MyReturn[0]);
```

**Example of a Sybase stored procedure that returns a single value**
The following complete example shows how to call a Sybase stored procedure that returns a single value.

In this example, the data source name is SYB_01 and the stored procedure name is GetNodeByIPAddress. The results of the stored procedure are assigned to the MyReturn array.

```
// Create the Sp_Parameter context.
Sp_Parameter = NewObject();

// Populate the Sp_Parameter member variables with values for
// the stored procedure input parameters.
Sp_Parameter.IPAddress = "192.168.1.250";
Sp_Parameter.Location = "Melbourne";

// Call CallStoredProcedure and pass the name of the data source,
// the name of the stored procedure and Sp_Parameter as input
// parameters
DataSource = "SYB_01";
ProcName = "GetHostnameByIPAddress";
MyReturn = CallStoredProcedure(DataSource, ProcName, Sp_Parameter);
Log("The hostname of the system is: " + MyReturn[0]);
```

**Calling procedures that return database rows**

Use this procedure to call a Sybase stored procedure that returns a set of database rows.

**Procedure**

- Create a new context called Sp_Parameter that is used to store input and output variables for the procedure.
- Populate the Sp_Parameter member variables with the input parameter values for the stored procedure.

The following example shows how to populate the Sp_Parameter member variables with values for the CustType and Location input parameters.
Sp_Parameter.CustType = "Platinum";
Sp_Parameter.Location = "Singapore";

- Call the CallStoredProcedure function and pass the name of the data source, the name of the stored procedure, and Sp_Parameter.

In this example, the name of the data source is SYB_03 and the name of the stored procedure is GetCustomersByLocation. The results of the stored procedure are assigned to the MyReturn variable.

MyReturn = CallStoredProcedure("SYB_03", "GetCustomersByLocation", Sp_Parameter);

- You can now handle the returned rows by accessing the array of contexts returned by the CallStoredProcedure function.

Creating the Sp_Parameter context

Before you call any stored procedure you need to create a new context called Sp_Parameter.

Procedure

To create a new Sp_Parameter context call the NewObject function as follows.

Sp_Parameter = NewObject();

Populating the Sp_Parameter member variables

Use these guidelines to populate the Sp_Parameter member variables.

Make sure that the name of the member variables is exactly the same as those of the input parameters in the procedure. Specify a value for each parameter in the stored procedure, even if you want to accept the default.

The following example shows how to populate the Sp_Parameter member variables with values for the Hostname and Location input parameters in the stored procedure.

Sp_Parameter.Hostname = "192.168.1.25";
Sp_Parameter.Location = "New York";

The following example shows how to populate the Sp_Parameter member variables with values for the CustType and Location input parameters in the stored procedure.

Sp_Parameter.CustType = "Premium";
Sp_Parameter.Location = "New York";

Use this code snippet to populate the Sp_Parameter member variable with a new context that will contain an output parameter returned from the stored procedure. In this example, the output parameter is called Name and contains an Oracle VARRAY.

Sp_Parameter.Name = NewObject();
Sp_Parameter.Name.type = "CUST";

The following example shows how to populate the Sp_Parameter member variables with values for the IPAddress and Location input parameters.

Sp_Parameter.IPAddress = "192.168.1.25";
Sp_Parameter.Location = "Singapore";

Calling the CallStoredProcedure function

When you call CallStoredProcedure, the function calls the procedure in the specified data source and returns the output parameters as member variables in Sp_Parameter.
You pass it the name of the data source, the name of the stored procedure, and the `Sp_Parameter` variable. Names of the member variables correspond to the names of the output parameters. Make sure that you use the data source name, not the name of a data type.

The following examples demonstrate how to call the `CallStoredProcedure` function.

In this example, the name of the data source is `Oracle_01` and the name of the stored procedure is `GetCustomerByID`.

```c
CallStoredProcedure("Oracle_01", "GetCustomerByID", Sp_Parameter);
```

In this example, the name of the data source is `SYB_03` and the name of the stored procedure is `GetCustomersByLocation`. The results of the stored procedure are assigned to the `MyReturn` variable.

```c
MyReturn = CallStoredProcedure("SYB_03", "GetCustomersByLocation", Sp_Parameter);
```

In this example, the name of the data source is `DB2DS` and the name of the stored procedure is `GetHostnameByIP`.

```c
CallStoredProcedure('DB2DS', 'GetHostnameByIP', Sp_Parameter);
```

**Handling the returned rows**

When you call the stored procedure, the underlying data source returns a set of database rows.

**Procedure**

Netcool/Impact creates an array of contexts and assigns each row to a context. Within each context, the member variables correspond to fields in the row. Netcool/Impact then returns the array from the `CallStoredProcedure` function. The following example shows how to handle the set of database rows returned from a Sybase stored procedure.

```c
MyReturn = CallStoredProcedure("SYB_03", "GetCustomersByLocation", Sp_Parameter);
Count = Length(MyReturn);
While(Count > 0) {
   Index = Count - 1;
   Log("The customer name is: " + MyReturn[Index].Name);
   Log("The customer ID is: " + MyReturn[Index].ID);
   Count = Count - 1;
}
```

**Example of a Sybase stored procedure that returns a set of database rows**

The following complete example shows how to call a Sybase stored procedure that returns a set of database rows.

In this example, the data source is named `SYB_03` and the stored procedure is named `GetCustomersByLocation`. The results of the stored procedure are stored as an array of contexts in the `MyResults` array.

```c
// Create the Sp_Parameter context
Sp_Parameter = NewObject();
// Populate the Sp_Parameter member variables with values for the stored procedure input parameters.
Sp_Parameter.CustType = "Platinum";
```
Sp_Parameter.Location = "Mumbai";

// Call CallStoredProcedure and pass the data source name, the
// stored procedure name and Sp_Parameter as input parameters.

DataSource = "SYB_03";
ProcName = "GetCustomerByLocation";
MyResults = CallStoredProcedure(DataSource, ProcName, Sp_Parameter);

// Print the customer name and IDs to the policy log.

Count = Length(MyResults);

While(Count > 0) {
    Index = Count - 1;
    Log("The customer name is: " + MyReturn[Index].Name);
    Log("The customer ID is: " + MyReturn[Index].ID);
    Count = Count - 1;
}

---

### DB2 SQL stored procedures

You can call DB2 SQL stored procedures from within a policy using the CallStoredProcedure function.

The CallStoredProcedure function sends a request to the database that contains the procedure name and its parameters. The results of the procedure are returned to the policy in a format that can be processed by Netcool/Impact. You can use the CallStoredProcedure function to call the following types of procedures:

- Procedures that accept **IN**, and **INOUT** parameters.
- Procedures that return values in **INOUT** and **OUT** parameters.
- Procedures that return Result Sets.

Parameters are useful in DB2 SQL procedures when implementing logic that is conditional on a particular input or set of input scalar values. You can also use the parameters when you want to return one or more output scalar values and you do not want to return a result set. DB2 SQL supports stored procedures with parameters that only accept an input value **IN**, that only return an output value **OUT**, or that accept an input value and return an output value **INOUT**. **IN** and **OUT** parameters are passed by value, and **INOUT** parameters are passed by reference.

A result set is the set of rows that a DB2 SQL procedure returns for a SELECT statement. You can either discover result set definitions by specifying values for the input values, or you manually define a result set and its columns.

Automatic schema discovery is always used when DB2 SQL stored procedures are called. Unlike Oracle stored procedures, you cannot disable automatic schema discovery for DB2 SQL stored procedures.

### Calling procedures that return scalar values

Use the following steps to create a DB2 SQL stored procedure.

**Procedure**

- Create a DB2 SQL data source
- Create a new context called Sp_Parameter that is used to store input and output variables for the procedure.
When you have an DB2 SQL data source, the next step before you call any DB2 SQL stored procedure is to create a context called Sp_Parameter. The Sp_Parameter context is used to store input and output variables for the DB2 SQL stored procedure.

- Populate the Sp_Parameter member variables with the input parameter values for the stored procedure. Include the IN and INOUT parameters. There is no need to populate the Sp_Parameter with an OUT parameter because the value will be in the Sp_Parameter automatically after the CallStoredProcedure function runs successfully.

The following example shows how to populate the Sp_Parameter member variables with values for the Hostname and Location input parameters in the stored procedure:

```c
Sp_Parameter.Hostname = '192.168.1.25';
Sp_Parameter.Location = 'New York';
```

- Call the CallStoredProcedure function and pass the name of the data source, the name of the stored procedure, and Sp_Parameter.

When you call CallStoredProcedure, the function calls the procedure in the DB2 data source. The function then passes values for the input parameters from Netcool/Impact to the DB2 stored procedure. Then the function returns the output parameters as member variables in the Sp_Parameter. The names of the member variables correspond to the names of the output parameters.

The following example shows how to call CallStoredProcedure. In this example, the name of the data source is DB2DS and the name of the stored procedure is GetHostnameByIP:

```c
CallStoredProcedure('DB2DS', 'GetHostnameByIP', Sp_Parameter);
```

### Creating a DB2 SQL data source

You must have a DB2 SQL data source for DB2 SQL stored procedures to work in Netcool/Impact.

**Procedure**

Information about how to create a DB2 SQL data source is documented in the online help. Go to Data sources, and refer to the section on Creating SQL data sources.

### Creating the Sp_Parameter context

Before you call any stored procedure you need to create a new context called Sp_Parameter.

**Procedure**

To create a new Sp_Parameter context call the NewObject function as follows.

```c
Sp_Parameter = NewObject();
```

**Populating the Sp_Parameter member variables**

Use these guidelines to populate the Sp_Parameter member variables.

Make sure that the name of the member variables is exactly the same as those of the input parameters in the procedure. Specify a value for each parameter in the stored procedure, even if you want to accept the default.
The following example shows how to populate the Sp_Parameter member variables with values for the Hostname and Location input parameters in the stored procedure.

```java
Sp_Parameter.Hostname = "192.168.1.25";
Sp_Parameter.Location = "New York";
```

The following example shows how to populate the Sp_Parameter member variables with values for the CustType and Location input parameters in the stored procedure.

```java
Sp_Parameter.CustType = "Premium";
Sp_Parameter.Location = "New York";
```

Use this code snippet to populate the Sp_Parameter member variable with a new context that will contain an output parameter returned from the stored procedure. In this example, the output parameter is called Name and contains an Oracle VARRAY.

```java
Sp_Parameter.Name = NewObject();
Sp_Parameter.Name.type = "CUST";
```

The following example shows how to populate the Sp_Parameter member variables with values for the IPAddress and Location input parameters.

```java
Sp_Parameter.IPAddress = "192.168.1.25";
Sp_Parameter.Location = "Singapore";
```

**Calling the CallStoredProcedure function**

When you call CallStoredProcedure, the function calls the procedure in the specified data source and returns the output parameters as member variables in Sp_Parameter.

You pass it the name of the data source, the name of the stored procedure, and the Sp_Parameter variable. Names of the member variables correspond to the names of the output parameters. Make sure that you use the data source name, not the name of a data type.

The following examples demonstrate how to call the CallStoredProcedure function.

In this example, the name of the data source is Oracle_01 and the name of the stored procedure is GetCustomerByID.

```java
CallStoredProcedure("Oracle_01", "GetCustomerByID", Sp_Parameter);
```

In this example, the name of the data source is SYB_03 and the name of the stored procedure is GetCustomersByLocation. The results of the stored procedure are assigned to the MyReturn variable.

```java
MyReturn = CallStoredProcedure("SYB_03", "GetCustomersByLocation", Sp_Parameter);
```

In this example, the name of the data source is DB2DS and the name of the stored procedure is GetHostnameByIP.

```java
CallStoredProcedure("DB2DS", "GetHostnameByIP", Sp_Parameter);
```

**Examples of DB2 SQL stored procedures using parameters**

Examples of DB2 SQL stored procedures using IN, OUT, and, INOUT parameters, and a stored procedure that returns a result set.

**Example of inserting values to a table using IN parameters**

The following example of a stored procedure, insert_data_procedure accepts two IN parameters and inserts the values to a table.
Example of a stored procedure that uses IN and OUT parameters

The following example of a stored procedure, get_ip_info, accepts an IN parameter, which is a HostName and returns the IP address using the OUT parameter IPAddress.

```java
Data_Source = 'DB2DS';
Sp_Name = 'get_ip_info';
Sp_Parameter = NewObject();
Sp_Parameter.HostName = 'mycompany.com';
CallStoredProcedure(Data_Source, Sp_Name, Sp_Parameter);
Log("IP Address is: " + Sp_Parameter.IPAddress);
```

Example of a stored procedure that returns a result set

The following example shows a DB2 SQL stored procedure that returns a result set.

```java
Data_Source = 'DB2DS';
Sp_Name = 'get_all_ip_data';
NetworkData = CallStoredProcedure(Data_Source, Sp_Name, Sp_Parameter);
Num_IP = Length(NetworkData);
log("First IP ": NetworkData[0].IPAddress);
```

In the example, NetworkData is an array with each entry representing a row of data.
- To get the number of rows in the result set, use the length function for example;
  ```java
  Length(NetworkData);
  ```
- To access a specific field value, use the Array[index].<fieldname> syntax for example;
  ```java
  NetworkData[0].IPAddress;
  ```

In this instance, NetworkData[0] contains the values for the first row of the result set. NetworkData[0].IPAddress returns the value for the column IPAddress for that first row.

Examples of DB2 SQL stored procedures that return an array

SQL procedures support parameters and variables of array types. Arrays are a convenient way of passing transient collections of data between an application and a stored procedure or between two stored procedures.

The following example shows a DB2 SQL stored procedure that returns an array.

```java
Data_Source = 'DB2DS';
Sp_Name = 'get_host_names';
Sp_Parameter = NewObject();
```

In the following example, HOST_LIST is an OUT parameter containing an array of host names. When an OUT parameter contains an array, you must specify the value TYPE as an array TYPE = ARRAY before calling the stored procedure using the CallStoredProcedure function.
Sp_Parameter.HOST_LIST = NewObject();
Sp_Parameter.HOST_LIST.TYPE = "ARRAY";
CallStoredProcedure(Data_Source, Sp_Name, Sp_Parameter);

In this example myHosts stores the array returned from the OUT parameter:
myHosts = Sp_Parameter.HOST_LIST;
Num_Hosts = Length(myHosts);
i = 0;
while (i < Num_Hosts) {
    log("Host Name is " + myHosts[i]);
    i = i + 1;
}
Chapter 8. Working with filters

A filter is a text string that sets out the conditions under which Netcool/Impact retrieves the data items.

The use of filters with internal, SQL, LDAP, and some Mediator data types is supported. The format of the filter string varies depending on the category of the data type.

SQL filters

SQL filters are text strings that you use to specify a subset of the data items in an internal or SQL database data type.

For SQL database and internal data types, the filter is an SQL WHERE clause that provides a set of comparisons that must be true in order for a data item to be returned. These comparisons are typically between field names and their corresponding values.

Syntax

For SQL database data types, the syntax of the SQL filter is specified by the underlying data source. The SQL filter is the contents of an SQL WHERE clause specified in the format provided by the underlying database. When the data items are retrieved from the data source, this filter is passed directly to the underlying database for processing.

For internal data types, the SQL filter is processed internally by the policy engine. For internal data types, the syntax is as follows:

```
Field
    Operator
    Value [AND | OR | NOT (Field
    Operator
    Value) ...]
```

where Field is the name of a data type field, Operator is a comparative operator, and Value is the field value.

Attention: Note that for both internal and SQL data types, any string literals in an SQL filter must be enclosed in single quotation marks. The policy engine interprets double quotation marks before it processes the SQL filter. Using double quotation marks inside an SQL filter causes parsing errors.

Operators

The type of comparison is specified by one of the standard comparison operators. The SQL filter syntax supports the following comparative operators:

- >
- <
- =
- <=
- =>
• !=
• LIKE

**Restriction:** You can use the LIKE operator with regular expressions as supported by the underlying data source.

The SQL filter syntax supports the AND, OR and NOT boolean operators.

**Tip:** Multiple comparisons can be used together with the AND, OR, and NOT operators.

**Order of operation**

You can specify the order in which expressions in the SQL are evaluated using parentheses.

**Examples**

Here is an example of an SQL filter:

```sql
Location = 'NYC'
Location LIKE 'NYC.*'
Facility = ' Wandsworth' AND Facility = ' Putney'
Facility = ' Wall St.' OR Facility = ' Midtown'
NodeID >= 123345
NodeID != 123234
```

You can use this filter to get all data items where the value of the Location field is New York:

```sql
Location = 'New York'
```

Using this filter you get all data items where the value of the Location field is New York or New Jersey:

```sql
Location = 'New York' OR Location = 'New Jersey'
```

To get all data items where the value of the Location field is Chicago or Los Angeles and the value of the Level field is 3:

```sql
(Location = 'New York' OR Location = 'New Jersey') AND Level = 3
```

---

**LDAP filters**

LDAP filters are filter strings that you use to specify a subset of data items in an LDAP data type.

The underlying LDAP data source processes the LDAP filters. You use LDAP filters when you do the following tasks:

• Retrieve data items from an LDAP data type using `GetByFilter`.
• Retrieve a subset of linked LDAP data items using `GetByLinks`.
• Delete individual data items from an LDAP data type.
• Specify which data items appear when you browse an LDAP data type in the GUI.

**Syntax**

An LDAP filter consists of one or more boolean expressions, with logical operators prefixed to the expression list. The boolean expressions use the following format:
Attribute
Operator
Value

where Attribute is the LDAP attribute name and Value is the field value.

The filter syntax supports the =, ~=, <, <=, >, >=, and ! operators, and provides
limited substring matching using the * operator. In addition, the syntax also
supports calls to matching extensions defined in the LDAP data source. White
space is not used as a separator between attribute, operator, and value, and those
string values are not specified using quotation marks.

For more information on LDAP filter syntax, see Internet RFC 2254.

Operators

As with SQL filters, LDAP filters provide a set of comparisons that must be true in
order for a data item to be returned. These comparisons are typically between field
names and their corresponding values. The comparison operators supported in
LDAP filters are:

- =
- ~=
- <
- <=
- >
- >=
- !

One difference between LDAP filters and SQL filters is that any Boolean operators
used to specify multiple comparisons must be prefixed to the expression. Another
difference is that string literals are not specified using quotation marks.

Examples

Here is an example of an LDAP filter:

\[(cn=Mahatma Gandhi)\]
\[(!{location=NYC*})\]
\[{&(facility=Wandsworth)(facility=Putney})\]
\[{&(facility=Wall St.)(facility=Midtown)(facility=Jersey City})\]
\[(nodeid>=12345)\]

You can use this example to get all data items where the common name value is
Mahatma Gandhi:

\[(cn=Mahatma Gandhi)\]

Using this example you get all data items where the value of the location attribute
does not begin with the string NYC:

\[(!{location=NYC*})\]

To get all data items where the value of the facility attribute is Wandsworth or
Putney:

\[{&(facility=Wandsworth)(facility=Putney})\]
**Mediator filters**

You use Mediator filters with the GetByFilter function to retrieve data items from some Mediator data types.

The syntax for Mediator filters varies depending on the underlying DSA. For more information about the Mediator syntax for a particular DSA, see the DSA documentation.
Chapter 9. Scheduling policies

You can set up Netcool/Impact to run policies at specific times.

Running policies using schedules

You can use schedules with a policy activator to run one or more policies at specific times.

Procedure

1. Create a schedule.
   The first step in setting up policies to run at specific times is to create a schedule data type in the GUI. For more information, see “Creating a schedule data type.”

2. Create an internal data type that represents each policy as a task.
   After you create the schedule data type, you must create a data type that represents each policy as a task. The task data type can be an internal data type and typically has two user-defined fields. A field that contains a descriptive name for the task and one that contains the name of the policy associated with the task. For more information, see “Creating task data types” on page 84.

3. Create task data items.
   After you create the task data type, the next step is to create a task data item for each policy that you want to schedule. For more information, see “Creating task data items” on page 84.

4. Add the tasks to the schedule.
   After you have created the task data items, the next step is to add the tasks to the schedule that you created at the beginning. This requires you to specify the task that you want to schedule and the date or time at which you want the associated policy to be run. For more information, see “Adding the tasks to the schedule” on page 85.

5. Specify time ranges for each task.

6. Write a top scheduler policy that launches the tasks.
   A top scheduler policy is a policy that is responsible for checking the schedule to see whether any other policy is currently due to be run. For more information, see “Writing a top scheduler policy” on page 85.

7. Create a policy activator and configure it to run the top scheduler.
   The policy activator runs the top scheduler policy at intervals. When the top scheduler policy runs, it checks to see if any other policies are currently “on call” and then runs them. You can configure the policy activator to run at any interval of time. For more accurate timing of scheduled policies, use smaller intervals. For more information, see “Creating a policy activator” on page 86.

8. Start the policy activator.
   To start the policy activator, click the Start Service icon associated with the new policy activator where it is displayed in the Services tab in the toolbar.

Creating a schedule data type

You can create a schedule data type in the GUI Server.
Procedure
1. Click Data Model to open the Data Model tab.
2. Select a cluster from the Cluster list. From the Project list, select Global.
3. In the Data Model tab, click Schedule, right click and select New Data Type to open the New Data Type for Schedule tab.
4. Enter a unique name for the schedule in the Data Type Name field.
5. Click the Save icon to implement to create the schedule data type.

Creating task data types
Use this procedure to create the task data type.

Procedure
1. Click Data Model to open the Data Model tab.
2. Select a cluster and a project from the Cluster and Project lists.
3. In the Data Model tab, click Internal, right click and select New Data Type to open the New Data Type for Internal tab.
4. Enter a unique name for the data type in the Data Type Name field, for example, Tasks.
5. Create new place holder fields that contain a descriptive title for the task and the name of the policy as follows:
   a. Click New Field to open the New field window. Use this window to define the attributes for the data type fields.
   b. Enter a unique ID in the ID field, for example, TaskName or PolicyName.
   c. From the Format list, select String. The Display Name, Description and Field Name fields in this window are optional. For fields in internal data types, the actual name and display name must always be the same as the field ID. If you leave these fields empty, they will be automatically populated with the ID value.
   d. Click OK to save the changes and return to the data type tab. The fields will have actual values when you add values for the Key field and for the task name and policy name fields that you defined when you create the tasks data type in the next section.
6. From the Display Name Field list, select the field that contains the task name. This display name is displayed when you browse data items in the data type. It does not otherwise affect the behavior of the data type.
7. Click the Save icon to implement the changes and to create the task data type.

Creating task data items
Use this procedure to create a task data item.

Procedure
1. Click Data Model to open the Data Model tab.
2. Select a cluster and a project from the Cluster and Project lists.
3. In the Data Model tab, expand the Internal data type, select the task data item, right click and select View Data Items.
4. Click the New icon on the menu.
5. Enter values for the Key field and for the task name and policy name fields that you defined when you create the tasks data type. The value for the Key field can be the same as the task name. However, if the data items are created in the
internal data type or any other data type to be used in the schedule configuration, the Key field must be unique across the data type and all tasks and policies.

6. Click OK.

**Adding the tasks to the schedule**

Use this procedure to add a task to the schedule.

**Procedure**

1. Click Data Model to open the Data Model tab.
2. In the Data Model tab, expand Schedule data source, select the schedule task you created, then right click and select View Data Items to open the Data items: Schedule tab.
3. Click New to open the Schedule Editor.
4. In the Schedule Name field, enter a name for the schedule.
5. Add a description to the Description field.
6. From the Edit Members By Type list, select the name of the task data type that you created and click Edit.
7. In the Select Schedule Members window that opens, select the tasks that you want to schedule and click Add.
8. Click OK.
9. In the Schedule Editor window, select the task that you want to schedule in the Schedule Members list.
10. Select the type of time range you want to associate with the task from the Add New Time Range list and then click New. The possible types of time ranges are Daily, Weekly and Absolute.
11. In the Edit Time Range window that opens, specify the time range and time zone during which you want the policy to run. The exact time at which the policy is run depends on both this time range and the frequency at which the policy activator runs the top scheduler policy. Click OK.
12. Click OK again to exit the Schedule Editor window.

**Writing a top scheduler policy**

A top scheduler policy is a policy that is responsible for checking the schedule to see whether any other policy is currently due to be run.

It is also responsible for launching the policy. The top scheduler policy calls the GetScheduleMember function and retrieves the task data item that is currently "on call." It then obtains the name of the policy associated with the task and runs it using the Activate function.

The following example shows a typical top scheduler policy. In this example, the name of the schedule data type is Schedule and the name of the schedule itself is TasksSchedule. The Tasks data type contains a field named PolicyName that specifies the name of the policy to run.

```plaintext
// Call GetByKey and retrieve the schedule data item that contains
// the schedule of tasks

DataType = "Schedule";
Key = "TasksSchedule";
MaxNum = 1;

Schedules = GetByKey(DataType, Key, MaxNum);
```
// Call GetScheduleMember and retrieve the task that is currently
// "on call"
Tasks = GetScheduleMember(Schedules[0], 0, False, GetDate());

// Call Activate and launch the policy associated with the task
Activate(Null, Tasks[0].PolicyName);

Running policies using the policy activator

You can use a policy activator service to run one policy at specified intervals
during Netcool/Impact run time.

For example, if you want to run a policy named CHECK_SYSTEM_STATUS every 60
minutes during the day, create a policy activator, specify the name of the policy
and the time interval. Then start the service in the GUI Server. If you want to run a
different policy at specific times of day or week, you must use schedules.

Creating a policy activator

Use this procedure to create the policy activator.

Procedure

1. Click Services to open the Services tab.
2. In the Services tab, click the Create New Service icon on the toolbar. Click
   Policy Activator to open the New Policy Activator tab.
3. In the Service Name field, enter a unique name for the policy activator.
4. In the Activation Interval field, enter the interval in seconds at which you
   want the policy activator to run the top scheduler policy
5. From the Policy list, select the top scheduler policy that you created.
6. Select the Startup check box if you want the policy activator to run
   automatically when the server starts.
7. Select the Service Log check box if you want to write the service logs to a file.
8. Click the Save icon on the toolbar to create the policy activator.
9. To check that the service is running, open the policy logger service. On the
   right pane, select the policy activator service the log details display.
Chapter 10. Handling hibernations

Hibernations are policies that have been temporarily put to sleep. While a policy is asleep, it is stored internally at its current state and all processing is paused until it is woken by the hibernating policy activator service or by another policy. IPL and JavaScript languages support hibernation.

Hibernations overview

The Hibernation data type is a system data type that stores hibernating policies.

You do not typically create or modify Hibernation data items using the GUI. However, you can use the GUI to delete stored hibernations in the case that an error condition occurs and the hibernations are not woken by the hibernation policy activator or another policy.

An action key is a string that uniquely identifies a hibernation. When you hibernate a policy, you must specify a unique action key.

The hibernation timeout value is the number of seconds that a policy hibernates before it can be woken by the hibernating policy activator. The hibernation timeout value does not affect the time at which the hibernation can be woken by another policy.

Hibernations are designed to be used in X events in Y time solutions. This type of solution monitors an event source for a certain number of same events to occur within a time frame, it takes the designated event management action (for example, notifying an administrator of a repeating event condition).

You can put a policy into hibernation. You can also activate a hibernating policy or remove a hibernating policy from the hibernation data type. Use the RemoveHibernation function to remove a policy from the hibernation data type and to remove it from the hibernation queue.

Hibernating a policy

To hibernate a policy, you call the Hibernate function, and pass an action key and the number of seconds for it to hibernate.

The action key can be any unique string that you want to use to identify the policy. Typically, you obtain this string by performing any combination of the following tasks:

- Use the value of the Identifier field in an incoming ObjectServer event. The ObjectServer generates a unique Identifier value for each event.
- Use the Random function to generate a random value.
- Use the GetDate function to generate a value that is based on the current system time.

Examples of hibernating a policy

The following examples show how to hibernate a policy and work with IPL and JavaScript languages.
In this example, the action key is the value of the Identifier field in an incoming ObjectServer event. This policy hibernates for 60 seconds before it is woken by the hibernating policy activator.

// Call Hibernate and pass an action key and the timeout // value for the hibernation.

ActionKey = EventContainer.Identifier;
Reason = null;
Timeout = 60;

Hibernate(ActionKey, Reason, Timeout);

A shorter version of this policy is as follows.

Hibernate(EventContainer.Identifier, null, 60);

In this example, the action key is a combination of the current system time and a random value. This policy hibernates for 2 minutes before it is woken by the hibernating policy activator.

// Call Hibernate and pass an action key and the timeout // value for the hibernation.

ActionKey = GetDate() + "_" + Random(9999);
Reason = null;
Timeout = 120;

Hibernate(ActionKey, Reason, Timeout);

A shorter version of this policy is as follows.

Hibernate(GetDate() + Random(9999), null, 120);

Retrieving hibernations

Retrieving hibernations is the way that you get data items from the Hibernation data type.

You must retrieve a hibernation before you can wake it from within a policy or remove it.

You can retrieve hibernations in two ways:
  • Action key search
  • Filter

Retrieving hibernations by action key search

You can use the GetHibernatingPolicies function to retrieve hibernations using a lexicographical search of action key values.

About this task

GetHibernatingPolicies returns an array of Hibernation data items whose action keys fall within the specified start and end action keys.

The following example shows how to retrieve hibernations using an action key search. This search returns all the Hibernation data items whose action keys fall between ActionKeyAAA and ActionKeyZZZ. The example also prints the contents of the policy context to the action tree log.
// Call GetHibernatingPolicies and pass the start action key
// and end action key values.

StartActionKey = "ActionKeyAAA";
EndActionKey = "ActionKeyZZZ";
MaxNum = 10000;

MyHibers = GetHibernatingPolicies(StartActionKey, EndActionKey, MaxNum);
Log(CurrentContext());

A shorter version of this example is as follows.
MyHibers = GetHibernatingPolicies("ActionKeyAAA", "ActionKeyZZZ", 10000);
Log(CurrentContext());

Retrieving hibernations by filter
You can use the GetByFilter function to retrieve hibernations using a filter.

About this task
GetByFilter returns an array of Hibernation data items whose action keys match
the specified filter string. The filter is an SQL filter.

The following example shows how to retrieve hibernations using GetByFilter. In
this example, you retrieve the Hibernation data item whose action key is 76486467.
Then, you print the contents of the current policy context to the policy log.
// Call GetByFilter and pass the name of the data type
// and a filter string.

DataType = "Hibernation";
Filter = "ActionKey = '76486467'";
CountOnly = false;

MyHibers = GetByFilter(DataType, Filter, CountOnly);
Log(CurrentContext());

A shorter version of this example is as follows.
MyHibers = GetByFilter("Hibernation", "ActionKey = '76486467'", false);
Log(CurrentContext());

Waking a hibernation
There are two ways that you can wake a hibernation.

To wake a hibernation, you perform the following tasks:
• Retrieve the hibernation using GetHibernatingPolicies or GetByFilter
• Call ActivateHibernation

You must also run the RemoveHibernation function to remove the policy from the
hibernation queue and to free up memory resources.

Retrieving the hibernation
The first step in waking a hibernation is to retrieve it from the Hibernation data
type using GetHibernatingPolicies or GetByFilter.
About this task

This step is described in the previous section of this guide.

Calling ActivateHibernation

After you retrieve the hibernation, you can call the ActivateHibernation function and pass the data item as an input parameter.

Example

The following example shows how to wake a hibernation.

In this example, you wake a hibernation policy whose action key value is ActionKeyABC.

```c
// Call GetHibernatingPolicies and pass the start action key
// and end action key values.

StartActionKey = "ActionKeyAAA";
EndActionKey = "ActionKeyZZZ";
MaxNum = 10000;

MyHibers = GetHibernatingPolicies(StartActionKey, EndActionKey, MaxNum);
MyHiber = MyHibers[0];

// Call ActivateHibernation and pass the Hibernation data item as
// an input parameter.

ActivateHibernation(MyHiber);
```

Removing hibernations

Use the RemoveHibernation function to remove a policy from the hibernation data type and to remove it from the hibernation queue.

To remove a hibernation from the internal data repository, you call the RemoveHibernation function and pass the action key of the hibernation as an input parameter.

The following example shows how to remove a hibernation. In this example, the action key for the hibernation is ActionKeyABC.

```c
RemoveHibernation("ActionKeyABC");
```

Handling strings and arrays

Read the following information about handling strings and arrays in a policy.

Handling strings

You can use the Netcool/Impact policy to manipulate strings in various ways.

You can perform the following tasks with strings:
- Concatenate strings
- Find the length of a string
- Split a string into substrings
- Extract a substring from another string
- Replace a substring in a string
- Strip a substring from a string
• Trim white space from a string
• Change the case of a string
• Encrypt and decrypt strings

**Concatenating strings**
To concatenate strings, you use the addition operator (+).

**About this task**

You can concatenate two strings or multiple strings at the same time. You can also concatenate a string with a numeric value.

The following example shows how to concatenate strings.

```plaintext
String1 = "This";
String2 = "is a test";
String3 = String1 + " " + String2;
Log(String3);
String4 = "The value of X is " + 5;
Log(String4);
```

When you run this example, it prints the following messages to the policy log:

```
This is a test
The value of X is 5
```

**Finding the length of a string**
You can use the Length function to find the length of a string.

**About this task**

The Length function returns the number of characters in any text string.

The following example shows how to use the Length function.

```plaintext
NumChars = Length("This is a test.");
Log(NumChars);
```

When you run this example, it prints the following message to the policy log:

```
15
```

**Splitting a string into substrings**
You can use the Split function to split a string into substrings.

**About this task**

The Split function takes a string and a set of delimiter characters as input parameters. It returns an array in which each element is a substring.

The following example shows how to use the Split function.

```plaintext
MyString = "One, Two, Three, Four.";
Delimiters = ",,";
MyArray = Split(MyString, Delimiters);
Count = Length(MyArray);
While (Count > 0) {
```
Index = Count - 1;
Log(MyArray[Index]);
Count = Count - 1;
}

When you run this example, it prints the following message to the policy log:
Four
Three
Two
One

**Extracting a substring from another string**
You can use the word position or regular expression matching to extract a substring from another string.

**Extracting a substring using the word position:**

To use the word position to extract a substring, call the Extract function, and pass the string and the word position of the substring.

The following example shows how to extract a string in this way.
MyString = "This is a test.";
MySubstring = Extract(MyString, 2);
Log(MySubstring);

When you run this example, it prints the following message to the policy log:
a

**Extracting a substring using regular expression matching:**

You can use regular expression matching to retrieve a single substring or all substrings from a string.

To extract a single substring, you use the RExtract function. The RExtract function takes a string and a regular expressions pattern as input parameters. It returns the first matching substring that it finds in the string.

To extract all matching substrings, you use the RExtractAll function. As with RExtract, The RExtractAll function takes a string and a regular expressions pattern as input parameters. It returns an array that contains all the matching substrings.

**Replacing a substring in a string**
You can use the Replace function to replace a substring in a string.

**About this task**

The Replace function takes the string, the substring to replace and its replacement as input parameters. The function returns the string after it creates the replacement.

The following example shows how to replace a substring.
MyString = "This is a test.";
Substring1 = "is a";
Substring2 = "is not a";
MyString = Replace(MyString, Substring1, Substring2);
Log(MyString);

When you run this example, it prints the following message to the policy log:
This is not a test.

**Stripping characters from a string**
You can use the Strip function to strip characters from a string.

**About this task**

The Strip function takes the string and list of characters you want to strip as runtime parameters. It returns the string after the characters have been removed.

The following example shows how to strip white spaces and new lines from a string.
MyString = "This is \t \n test";
MyString = Strip(MyString, "\n\t ");
Log(MyString);

When you run the policy it will print this message to the log:
Thisisatest.

**Trimming white space from a string**
You can use the Trim function to trim leading and trailing white space from a string.

**About this task**

The Trim function takes the string as an input parameter and returns it without any leading or trailing white space.

The following example shows how to trim the white space from a string.
MyString = " This is a test. ";
MyString = Trim(MyString);
Log(MyString);

When you run this example, it prints the following message to the policy log:
This is a test.

**Changing the case of a string**
You can use the ToLower function to change the case of a string. You can also use the ToUpper function to change the case of a string to all uppercase.

**Example**

The following example shows how to change a string to lowercase.
Log(ToLower("THIS IS A TEST."));

When you run this example, it prints the following message to the policy log:
this is a test.

The following example shows how to change a string to uppercase.
Log(ToUpper("this is a test."));
When you run this example, it prints the following message to the policy log:
THIS IS A TEST.

**Encrypting and decrypting strings**
The policy language provides a feature that you can use to encrypt and decrypt strings.

**About this task**
This feature is useful if you want to handle password data within a Netcool/Impact policy.

You can use the Encrypt function to encrypt a string. This function takes the string as an input parameter and returns an encrypted version.

The following example shows how to encrypt a string:
MyString = Encrypt("password");

You can decrypt a string that you have previously encrypted using the Decrypt function. This function takes an encrypted string as an input parameter and returns the plaintext version.

The following example shows how to decrypt a string. In this example, the parameter to Decrypt the function is the return value of the Encrypt("password") call.
MyString = Decrypt("AB953E4925B39218F390AD2E9242EB1A");

**Handling arrays**
You can use the Netcool/Impact policy language to find the length of an array and to find distinct values in an array.

**Finding the length of an array**
You can use the Length function to find the number of elements in an array.

**About this task**
The Length function takes the array as an input parameter and returns its number of elements.

The following example shows how to find the number of elements in an array in IPL:
Elements = Length({"One", "Two", "Three"});
Log(Elements);

The following example shows how to find the number of elements in an array in JavaScript:
Elements = Length(["One", "Two", "Three"]);
Log(Elements);

When you run the example in either language, it prints the following message to the policy log:
3

**Finding the distinct values in an array**
You can use the Distinct function to find the distinct values in an array.
About this task

The Distinct function takes the array as an input parameter and returns another array that consists only of the unique, or non-duplicate, elements.

The following example shows how to find the distinct values in an array:

```csharp
MyArray = {"One", "One", "Two", "Three", "Three", "Four"};
MyArray = Distinct(MyArray);
Log(MyArray);
```

When you run this example, it prints the following message to the policy log:

```
{One, Two, Three}
```
Chapter 11. Using Netcool/Impact functions to integrate with Webtop

Using the Netcool/Impact functions in this integration you can use Netcool/Impact to complete the following tasks:

- Provision Webtop content such as maps, entities, and tools.
- Administer Webtop access control in either Webtop 1.3 or Webtop 2.0/2.1/NGF.
- Create NGF pages that contain Webtop content.

Netcool/Impact functions

This solution provides functions that you can use to complete the following tasks:

- Create complex XML structures as IPL objects and turn those objects into XML strings
- Save those XML strings into files locally using JRExec or remotely through SSH command/response
- Push the files to run waapi or the ngf_api locally through JRExec or remotely through SSH command/response
- Optionally delete the files afterwards

These functions are stored in several example policies, which can be found in the WebTop project.

There is example code showing how to create Webtop content, administer Webtop, configure NGF, and create NGF pages (psml files).

Supported operating systems

The Webtop integration is supported by Tivoli Netcool/Impact.

The most up-to-date information about supported hardware, software, browsers, and operating systems is provided by the IBM® Software Product Compatibility Reports at: [http://pic.dhe.ibm.com/infocenter/prodguid/v1r0/clarity/index.html](http://pic.dhe.ibm.com/infocenter/prodguid/v1r0/clarity/index.html)


**Note:** For UNIX users, the integration was not tested on any UNIX other than CentOS 4.5. The only complications might be the behavior of the echo command on different UNIX operating systems. Edit the function `cleanTextForEcho(text)` on the UTIL policy if you must. WAAP requires `$WAAP_HOME` to be set and ngf_api requires `$NCHOME` to be set for the user who is running the Netcool/Impact commands.

**Note:** For Windows users, all interaction with WAAP or NGF_API on a Windows box is through SSH Command/Response. The Windows policy code was developed against a Windows server running openSSH that provides Windows style pathing, for example, `(c:\Program Files)`. openSSH can be downloaded from [http://sshwindows.sourceforge.net/](http://sshwindows.sourceforge.net/) `$NCHOME` and `$WAAP_HOME` must be set on the
Windows box before you start the openSSH Service. Always use short file names in your paths when you work with Windows (c:\progra~1\ibm\netcool).

Setting up the integration

Use this procedure to set up the integration with Webtop.

Procedure

1. Import the policies into your Netcool/Impact cluster.
2. Configure the policies for your environment by editing the getENV function in the WEBTOP, NGF, and PSML policies.

   If you are running Webtop or NGF on Windows, you need to install openSSH on the Windows host. See [http://sshwindows.sourceforge.net/](http://sshwindows.sourceforge.net/)
Chapter 12. Working with IPL to XML functions

Data source adapters allow for access to data from a wide variety of sources. In many cases, the data is retrieved from SQL data sources and delivered as Impact Policy Language (IPL) objects (contexts). One challenge for policy writers has been converting these IPL contexts into XML strings for applications with interfaces that expect data in XML format. To facilitate this task, a set of IPL to XML functions has been developed that can be used to generate an XML string from an IPL object.

IPL to XML functions overview

You use IPL to XML functions to generate an XML string from an IPL object. What effectively happens is a top-level IPL object, referred to as the XML document object, is transformed into XML. IPL objects nested within the document object, referred to as element objects, become XML elements. The functions are used to create the XML document and element objects and to set XML attributes, content, and comments.

XML document object

The XML document object is the base object. XML element objects, attributes, content, and comments are added to the XML document object. It is the XML document object that is converted into an XML string.

For information about how to create the XML document object, see “Creating the XML document object” on page 100.

XML element objects

XML element objects are added to the XML document object or to each other (for nested XML elements). After you created and added an XML element object you can use other functions to add XML attributes, content, and comments to it. For more information about the IPL to XML functions used to create and add XML element objects, see “Adding a sub element” on page 100 and “Creating an unassociated element” on page 101.

Adding XML attributes to element objects

You can use on of three methods to add attributes to XML element objects. The simplest method is to pass an element object, attribute name, and attribute value to the IPLtoXML.addAttribute() function. For more information about this method, see “Adding XML attributes to element objects, simple approach” on page 101.

The second method requires creating a separate XML attribute object and adding that XML attribute object to the XML element object using the IPLtoXML.addAttributeObject() function. For more information about how to use this function, see “Adding XML attributes to element objects that use Attribute objects” on page 102. This method is useful in cases where you expect to use the same attribute and value in many places in your XML.

You use the third method to add several attributes at once using the IPLtoXML.addOrgNodeAttributes() function. Use this function to quickly take data
from a data type lookup and add all the fields to an XML element object as attributes. For more information about this method, see “Adding XML attributes to element objects adding attributes from an OrgNode” on page 103.

**Adding XML content to element objects**

You can add content to any element and append additional content to it later. For more information about how to add and append content to element objects, see “Adding the content to an XML element object” on page 103 and “Appending content to XML element objects” on page 104.

**XML comments**

Using the addCommentToElement function you can add comments to any XML element. The function also puts in the XML commenting code (<!-- -->) for you so do not have to add it manually.

For more information about using this function, see “Adding XML comments to element objects” on page 104.

**Nesting XML elements**

In cases where you created a stand-alone XML element object using the newElement function you can still nest that XML element object using the addElement function. In most cases you, will not be creating stand-alone objects using the newElement function but instead will be creating and nesting XML element objects at the same time using the newSubElement function.

For more information about adding XML element objects to each other, see “Adding XML element objects to each other (nesting)” on page 105.

---

**Creating the XML document object**

**Procedure**

Use this function to create the XML document object:

IPLtoXML.newDocument(myXMLDocumentObject)

The myXMLDocumentObject variable becomes the new XML document object.

**Example**

IPLtoXML.newDocument(REM_Album);

**Adding a sub element**

This function creates an XML element object and nests it within the parent element object in one step.

**Procedure**

To add a sub element, use this function:

IPLtoXML.newSubElement(parentElement, myElement, elementType)

where
parentElement
This element or document object must exist.

myElement
This variable becomes the new XML element of type elementType nested within the parentElement.

elementType
The type of XML element to create.

Example

IPL:
IPLtoXML.newDocument(myCars);
IPLtoXML.newSubElement(myCars, myElement, "Honda");
IPLtoXML.newSubElement(myElement, driver1, "susi");

Generated XML:
<Honda><susi/></Honda>

Creating an unassociated element

Rather than using this function to create an unassociated element you can use the newSubElement function to create and nest an XML element object in one step.

Procedure

To create an unassociated element, use this function:
IPLtoXML.newElement(myElement, elementType)

where

myElement
This variable becomes the new XML element of type elementType.

elementType
The type of XML element to create.

Example

IPL:
IPLtoXML.newElement(myElement, "Honda")

Generated XML:
<Honda/>

Adding XML attributes to element objects, simple approach

Procedure

To add XML attributes to an element object, use this function:
IPLtoXML.addAttribute(myElementObject, attributeName, attributeValue)

where

myElementObject
The element to add the attribute to.
attributeName
The string name of the attribute to add.

attributeValue
The value of the attribute.

Example

IPL:
IPLtoXML.newElement(myForester, "Subaru");
IPLtoXML.addAttribute(myForester, "year", 2003);

Generated XML:
<Subaru year="2003"/>

Note: Use addAttribute when you want to avoid creating the attribute object.

Adding XML attributes to element objects that use Attribute objects

To add XML attributes to an element object that Attribute objects follow this procedure.

Procedure
1. Create the attribute object. Use the following function:
   IPLtoXML.newAttributeObject(attributeObject, attributeName, attributeValue)
   
   where
   
   attributeObject
   This variable becomes the new XML attribute object.
   
   attributeName
   The name of the attribute.
   
   attributeValue
   The value for the attribute.
   
   IPL:
   IPLtoXML.newAttributeObject(carYear, "year", 2003);
   Generated XML:
   year="2003"

   Note: Attribute objects are useful because they can then be reused and added to multiple elements in your code.

2. Add the attribute object to an element object. Use the following function:
   IPLtoXML.addAttributeObject(myElementObject, attributeObject)
   
   where
   
   myElementObject
   The XML element object to which the attribute is added
   
   attributeObject
   The XML attribute object that is added to myElementObject.
   
   IPL:
   IPLtoXML.newElement(myForester, "Subaru");
   IPLtoXML.newAttribute(carYear, "year", 2003);
   IPLtoXML.addAttributeObject(myForester, carYear);
Adding XML attributes to element objects adding attributes from an OrgNode

Procedure

To add attributes from an OrgNode, use this function:
IPLtoXML.addOrgNodeAttributes(elementObject, OrgNode)

where

- **elementObject**
  The element object to add the attributes to.

- **OrgNode**
  An IPL object whose fields you want to add to element as attributes. The object could have come from a lookup or could have been built using `newobject()`.

Example

IPL:
IPLtoXML.newElement(myForester, "Subaru");
og=newobject();
og.color="blue";
og.doors=4;
og.turbo="no";
IPLtoXML.addOrgNodeAttributes(myForester, og);

Generated XML:
<Subaru color="blue" doors="4" turbo="no"/>

Adding the content to an XML element object

Content can be added to any element.

Procedure

To add the content to an XML element object, use this function:
IPLtoXML.setContent(myElementObject, content)

where

- **myElementObject**
  The XML element object to add the content to.

- **content**
  The text string to add to myElementObject as content.

Example

IPL:
IPLtoXML.newElement(myForester, "Subaru");
IPLtoXML.setContent(myForester, "Extra Car");
Appending content to XML element objects

Additional content can be appended to any element.

**Procedure**

To append the content to an XML element object, use this function:
IPLtoXML.appendContent(myElementObject, content)

where

myElementObject
   The XML element object to append the content to.

content
   The text string to append to the existing content.

**Example**

IPL:
IPLtoXML.newElement(myForester, "Subaru");
IPLtoXML.setContent(myForester, "Extra Car");
IPLtoXML.appendContent(myForester, " driven by Dad");

Generated XML:
<Subaru>Extra Car driven by Dad</Subaru>

Adding XML comments to element objects

**Procedure**

To add XML comments to element objects, use this function:
IPLtoXML.addCommentToElement(myElementObject, comment)

where

myElementObject
   The XML element object to add the XML comment to.

comment
   The text string that becomes the XML comment. XML remarking code <!-- --> is added by the function. Do not add it yourself.

**Example**

IPL:
IPLtoXML.newElement(myForester, "Subaru");
addCommentToElement(myForester, "mom drives the element");

Generated XML:
<Subaru><!--mom drives the element--></Subaru>
Adding XML element objects to each other (nesting)

Procedure

To nest an XML element object use this function:
IPLtoXML.addElement(dore, myElementObject)

where

dore    The XML document or element object to add myElementObject to.

myElementObject
    The XML element object that is added to the dore.

Example

IPL:
    IPLtoXML.newElement(myCars, "Cars");
    IPLtoXML.newElement(myForester, "Subaru");
    IPLtoXML.addElement(myCars, myForester);

Generated XML:
    <Cars><Subaru/></Cars>

Generating XML strings from document objects

Procedure

To generate an XML string from the document object, use this function:
IPLtoXML.generateXML(xmlPiece, XML)

where

xmlPiece
    Either an entire XML document object or just one XML element object.

XML
    The variable to hold the XML string that is generated from xmlPiece.

Example

IPL:
    STACK=NewJavaObject("java.util.Stack", {}); //
    this global variable is required for the IPLtoXML conversion
    IPLtoXML.newDocument(carXML);
    IPLtoXML.newElement(myForester, "Subaru");
    IPLtoXML.addElement(carXML, myForester);
    IPLtoXML.addCommentToElement(myForester, "mom drives the element");
    IPLtoXML.generateXML(carXML, Output);

This XML is generated in the Output variable:
    <?xml version="1.0" encoding="UTF-8"?><Subaru<!-- mom drives the element--></Subaru>
Replacement of default XML entities

The function replaceEntities() replaces the following default XML entities in XML content and attributes:

- & ampersand, replaced with &amp;
- < less than, replaced with &lt;
- > greater than, replaced with &gt;
- ' apostrophe, replaced with &apos;
- " quotation mark, replaces with &quot;

If you have additional entities that need to be replaced then edit the function replaceEntities(x) within the IPL to XML policy.

Element ordering in XML

The order in which elements at any particular nesting depth are added to the generated XML is based on the element object variable name.

Let us take the following policy for example:

```
IPLtoXML.newDocument(theWeather);
IPLtoXML.newSubElement(theWeather, system, "weatherSystem");
IPLtoXML.newSubElement(system, sub01, "tornado");
IPLtoXML.newSubElement(system, sub05, "thunderstorm");
IPLtoXML.newSubElement(system, SUB, "hail");
IPLtoXML.addComment(sub05, "bad thunderstorms and a tornado where I live today");
```

The resulting XML looks like this example:

```
<weatherSystem><hail/><tornado/><thunderstorm><!-- bad thunderstorms and a tornado where I live today --></thunderstorm></weatherSystem>
```

The hail, tornado, and thunderstorm elements are all at the same depth in the XML nesting. The hail element was added first, element object name SUB. The tornado element was added second, element object name sub01. The thunderstorm element was added last sub05. If the element order in the XML is important then name the element objects carefully.

Examples of IPLtoXML functions usage

This section contains three examples of usage of IPLtoXML functions.

A simple example

This example shows how the different IPLtoXML functions can be used to generate a simple XML structure.

```
STACK=NewJavaObject("java.util.Stack", {});//required for recursion in generateXML function
IPLtoXML.newDocument(docObj);
IPLtoXML.newElement(cars, "cars");
IPLtoXML.addElement(docObj, cars);
IPLtoXML.addAttribute(cars, "familyName", "Daniel");
IPLtoXML.setContent(cars, "the cars owned by a family");
IPLtoXML.newElement(subaru, "car");
IPLtoXML.newElement(honda, "car");
```
IPLtoXML.addElement(cars, subaru);
IPLtoXML.addElement(cars, honda);
IPLtoXML.newAttributeObject(carLocation, "location", "in the garage");
IPLtoXML.addAttributeObject(subaru, carLocation);
IPLtoXML.addAttributeObject(honda, carLocation);
IPLtoXML.addAttribute(honda, "color", "black");
IPLtoXML.addAttribute(honda, "doors", 4);
IPLtoXML.addAttribute(honda, "driver", "susi");
IPLtoXML.addAttribute(honda, "model", "honda element");
foresterDetails=newobject();
foresterDetails.color="blue";
foresterDetails.doors="4";
foresterDetails.driver="tom";
foresterDetails.model="subaru forester";
IPLtoXML.addOrgNodeAttributes(subaru, foresterDetails);
IPLtoXML.newElement(hondaDriver, "Driver");
IPLtoXML.addAttribute(hondaDriver, "Name", "Susan Daniel");
IPLtoXML.addAttribute(hondaDriver, "Age", 33);
IPLtoXML.addCommentToElement(hondaDriver, "mother of twins");
IPLtoXML.addElement(honda, hondaDriver);
IPLtoXML.generateXML(docObj, xml);
log(xml);

The results of logging the variable XML:
07 Oct 2007 17:11:06,888: SynchronousMessageProcessor:
Parser log: <?xml version="1.0" encoding="UTF-8"?><cars familyName="Daniel">the cars owned by a family<car color="blue" doors="4" driver="tom" location="in the garage" model="subaru forester"/><car color="black" doors="4" driver="susi" location="in the garage" model="honda element"><Driver Age="33" Name="Susan Daniel"/><!-- mother of twins --></car></cars>

ObjectServer event example

This example shows how IPLtoXML is run on an event from the Netcool®/OMNibus ObjectServer. Let us assume that we start with an Netcool/OMNIBus event generated from the Netcool/OMNibus Simnet probe.
1. Create an XML document object:
   IPLtoXML.newDocument(eventXML);
2. Create an XML element object and add it to the XML document object:
   IPLtoXML.newSubElement(eventXML, theEvent, "omniEvent");
3. Add the fields in the Event Container to the theEvent element object as attributes:
   IPLtoXML.addOrgNodeAttributes(theEvent, EventContainer);
4. Use the Java DSA to create a Java memory stack in a global variable called STACK:
   STACK=NewJavaObject("java.util.Stack", {});
   Note: STACK is required every time the generateXML() function is called and must be a global variable (defined outside of any function). IPLtoXML uses recursion and (v4.x) you cannot do recursive functions in the Impact parser without a separate memory stack.
5. Call the GenerateXML() function itself. Pass the function the XML document object, eventXML, and a results variable to put the output into:
   IPLtoXML.GenerateXML(eventXML, results);
The generated XML looks like this example:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<omniEvent
   Acknowledged="0" Agent="LinkMon" AlertGroup="Link"
   AlertKey="" Class="3300" Customer="" EventId=""
   EventReaderName="OMNiBusEventReader" ExpireTime="0"
   FirstOccurrence="1188488450" Flash="0" Grade="0"
   Identifier="link6LinkMon1Link" InternalLast="1188488841"
   KeyField="8689" LastOccurrence="1188488841"
   LocalNodeAlias="" LocalPriObj="" LocalRootObj=""
   LocalSecObj="" Location="" Manager="Simnet Probe"
   NmosCauseType="0" NmosObjInst="0" NmosSerial=""
   Node="link6" NodeAlias="" OwnerGID="0" OwnerUID="65534"
   PhysicalCard="" PhysicalPort="0" PhysicalSlot="0"
   Poll="0" ProcessReq="0"
   ReceivedWhileImpactDown="1" RemoteNodeAlias=""
   RemotePriObj="" RemoteRootObj="" RemoteSecObj=""
   Serial="8689" ServerName="NCOMS" ServerSerial="8689"
   Service="" Severity="0" StateChange="1188488841"
   Summary="Link Up on port" SuppressEscl="0" Tally="0"
   TaskList="0" Type="2" URL="" X733CorrNotif=""
   X733EventType="0" X733ProbableCause="0" X733SpecificProb="/">
</omniEvent>
```

There is a single XML Element called omniEvent. Each of the fields in the original Event Container are XML attributes of omniEvent.

**Example of generating WAAPi XML using IPL**

This example shows how IPL to XML can be used to create an XML string that can be passed to the Netcool/Webtop API (WAAPi).

```java
// create the XML Document object
IPLtoXML.newDocument(waapiXML);
// first element: the methodCall
IPLtoXML.newSubElement(waapiXML, myMethodCall, "methodCall");
// a method element nested in the methodCall element
IPLtoXML.newSubElement(method1, method1, "method");
// attributes for method1
IPLtoXML.addAttribute(method1, "methodName", "entity.createOrReplaceEntity");
// entitygroup element nested in method element
IPLtoXML.newSubElement(method1, entityGroup01, "entitygroup");
// attributes for entitygroup
IPLtoXML.addAttribute(entityGroup01, "name", "support");

// this group was added to NGF
// entity element nested in entitygroup
IPLtoXML.newSubElement(entityGroup01, entity01, "entity");
// entity attributes
IPLtoXML.addAttribute(entity01, "name", "ncentity01");
IPLtoXML.addAttribute(entity01, "filter", "Severity > 4");
IPLtoXML.addAttribute(entity01, "metriclabel", "tom");
IPLtoXML.addAttribute(entity01, "metricshow", "Average");
IPLtoXML.addAttribute(entity01, "metricof", "Severity");
// entitylist nested in entitygroup
IPLtoXML.newSubElement(entityGroup01, entitylist01, "entitylist");
// entitylist attributes
IPLtoXML.addAttribute(entitylist01, "name", "ncentitylist01");
IPLtoXML.addAttribute(entitylist01, "list", "AllEvents");
IPLtoXML.addAttribute(entitylist01, "view", "basic");
IPLtoXML.addAttribute(entitylist01, "metriclabel", "tom");
```

The resulting XML string:
This string can be passed to WAAPI via JRExec or Command/Response to create the Webtop entity.
Chapter 13. Working with functions

The Impact Policy Language (IPL) and JavaScript support built-in functions and user-defined functions.

Unless stated otherwise the same functions can be used for IPL and JavaScript languages. There are differences in the syntax that is used in IPL and JavaScript.

Note: When you enter values in the function builder GUI for the parameters to a function and if the value is a string literal, you must add quotation marks around the value you enter in the function builder. If the value of the parameter refers to a variable defined earlier in the policy, then do not place quotation marks around it.

You use variables to pass values to functions. The variable is updated after the function is complete. As a result, you can use only variables to pass values to functions.

Activate

The Activate function runs another policy.

After the policy finishes running, Netcool/Impact returns to the first policy and processes any subsequent statements that it contains.

You can run a policy by name or by data item.

To run a policy by name, call Activate and pass the name of the policy to the function as an input parameter.

To run a policy by data item, you first retrieve the item from the internal data repository and then call Activate, and pass it to the function as an input parameter. Policies are stored in the repository in the internal Policy data type. You can retrieve policy data items by calling the GetByKey, GetByFilter, or GetByLinks functions. For data items of type Policy, the value of the KEY field is the same as the policy name. You can use this value in a key expression when you call GetByKey or use it in an SQL filter string when you call GetByFilter.

When you call the Activate function, the secondary policy inherits the variables set in the original policy. These include the EventContainer variable and those variables that store data items retrieved from internal or external data types. These variables are not passed back to the original policy after the second policy finishes running.

Syntax

The Activate function has the following syntax:

Activate([DataItem], [PolicyName])
Parameters

The Activate function has the following parameters.

Table 11. Activate function parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataItem</td>
<td>Data item</td>
<td>Data item that stores the policy to be activated. Pass a null value for this parameter if you are activating the policy by name.</td>
</tr>
<tr>
<td>PolicyName</td>
<td>String</td>
<td>Name of the policy to trigger. Pass a null value for this parameter if you are activating the policy by data item.</td>
</tr>
</tbody>
</table>

Example

The following example shows how to use the Activate function to run a policy by name.

// Call Activate and pass the name of the policy as an input // parameter
Activate(null, "POLICY_01");

The following example shows how to use the Activate function to run a policy by data item.

// Call GetByKey and pass the name of the data type and // the key expression as input parameters
DataType = "Policy";
Key = "POLICY_01";
MaxNum = 1;

MyPolicy = GetByKey(DataType, Key, MaxNum);

// Call Activate and pass the policy data item as an input // parameter
Activate(MyPolicy[0], null);

ActivateHibernation

The ActivateHibernation function continues running a policy that was previously put to sleep by using the Hibernate function. You must also run the RemoveHibernation function to remove the policy from the hibernation queue and to free up memory resources.

The policy is continued at the statement that follows the Hibernate function call. After the policy finishes running, Netcool/Impact returns to the original policy and processes any remaining statements that it contains.

Before you run a hibernating policy, you must first retrieve it from the internal data repository. Hibernating policies are stored in the repository as data items in the internal Hibernation data type. You retrieve a hibernation data item by calling the GetHibernatingPolicies or the GetByFilter function. If you call GetByFilter, you use an SQL filter to specify the action key value that identifies the hibernating policy. After you have retrieved the data item, call ActivateHibernation and pass it to the function as an input parameter.
Syntax

The `ActivateHibernation` function has the following syntax:

`ActivateHibernation(Hibernation)`

Parameters

The `ActivateHibernation` function has the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Hibernation</code></td>
<td>Data Item</td>
<td>Data item that stores the hibernating policy.</td>
</tr>
</tbody>
</table>

Examples

The following example shows how to continue running a hibernating policy using the `GetHibernatingPolicies` and `ActivateHibernation` functions.

```c
// Call GetHibernatingPolicies and pass the start and end action keys // as input parameters
StartActionKey = "ActionKeyAAAA";
EndActionKey = "ActionKeyZZZZ";
MaxNum = 1;
MyHiber = GetHibernatingPolicies(StartActionKey, EndActionKey, MaxNum);
// Call ActivateHibernation and pass the Hibernation data item as an // input parameter
ActivateHibernation(MyHiber[0]);
```

The following example shows how to continue running a hibernating policy using the `GetByFilter` and `ActivateHibernation` functions.

```c
// Call GetByFilter and pass the name of the Hibernation data type, // and a filter that specifies an action key that identifies the hibernation
DataType = "Hibernation";
Filter = "ActionKey = 'ActionKey0001'";
CountOnly = false;
MyHiber = GetByFilter(DataType, Filter, CountOnly);
// Call ActivateHibernation and pass the Hibernation data item as an // input parameter
ActivateHibernation(MyHiber[0]);
```

AddDataItem

The `AddDataItem` function adds a data item to a data type.

You can use `AddDataItem` with internal, SQL database, and some Mediator data types.

Before you add a new data item, you must first create a new context by using the `NewObject` function. Then, you assign values to the member variables, where each variable corresponds to a field in the data type. After you assign these values, call
AddDataItem and pass the data type name and the context to the function as input parameters. When you call AddDataItem, the values of the variables are used to populate fields in the new item.

**Syntax**

The AddDataItem function has the following syntax:

```
[DataItem =] AddDataItem(DataType, ContextToCopy)
```

**Parameters**

The AddDataItem function has the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataType</td>
<td>String</td>
<td>Name of the data type.</td>
</tr>
<tr>
<td>ContextToCopy</td>
<td>Context</td>
<td>Name of the context whose member variables contain initial field values for the data item.</td>
</tr>
</tbody>
</table>

**Return value**

The AddDataItem function can optionally return the context that was used to create the new data item.

**Example**

The following example shows how to add a data item to a data type by using the AddDataItem function. This example uses an internal or SQL database data type named Node. Data items in this type have the following fields, Id, Name, Location, and the key field NodeName.

If you are creating a new data item for an SQL data source, you must enter the key field into the context when you create the new data item.

```csharp
// Call NewObject and populate the member variables of the context with initial // field values for the data item

MyContext = NewObject();
MyContext.Id = "000123";
MyContext.Name = "ORACLE_01";
MyContext.Location = "Raleigh";
MyContext.NodeName = "node1";

// Call AddDataItem and pass the name of the data type and the context as // input parameters

DataType = "Node";
MyContext = AddDataItem(DataType, MyContext);
```

**BatchDelete**

The BatchDelete function deletes a set of data items from a data type.

You can use BatchDelete with SQL database data types. You cannot use this function with internal, LDAP, or Mediator data types.
You can specify which items to delete using an SQL filter or by passing the items to the function in an array.

To delete data items using an SQL filter, call BatchDelete and pass the name of the data type and the filter string to the function as input parameters. The filter string specifies which data items to delete. It uses the SQL filter syntax, which is similar to the syntax of the WHERE clause in an SQL SELECT statement. For more information about SQL filters, see "SQL filters" on page 79.

To delete data items by passing them in an array, you first retrieve them from the data type by calling GetByFilter, GetByKey, or GetByLinks. Then call BatchDelete and pass the name of the data type and the data item array to the function as input parameters.

Syntax

The BatchDelete function has the following syntax:

BatchDelete(DataType, [DeleteFilter], [DeleteDataItems])

Parameters

The BatchDelete function has the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataType</td>
<td>String</td>
<td>Name of the data type.</td>
</tr>
<tr>
<td>DeleteFilter</td>
<td>String</td>
<td>SQL filter string that specifies which data items to delete. Optional.</td>
</tr>
<tr>
<td>DeleteDataItems</td>
<td>Array</td>
<td>Array of data items to delete. Optional.</td>
</tr>
</tbody>
</table>

Examples

The following example shows how to delete a set of data items using an SQL filter. In this example, you delete all of the data items in a data type named Customer where the value of the Location field is Raleigh.

// Call BatchDelete and pass the name of the data type and a filter string as // input parameters

DataType = "Customer";
Filter = "Location = 'Raleigh'";
BatchDelete(DataType, Filter, null);

The following example shows how to delete a set of data items by passing them to the BatchDelete function in an array. In this example, you delete all of the data items in the data type Server where the value of the Facility field is SE_0014.

// Call GetByFilter and pass the name of the data type and a filter string as // input parameters

DataType = "Server";
Filter = "Facility = 'SE_0014'";
CountOnly = false;
MyServers = GetByFilter(DataType, Filter, CountOnly);
// Call BatchDelete and pass the array of data items as an input parameter
BatchDelete(DataType, null, MyServers);

**BatchUpdate**

The BatchUpdate function updates field values in a set of data items in a data type.

You can use BatchUpdate with SQL database data types. You cannot use this function with internal, LDAP, or Mediator data types.

To update the field values, call BatchUpdate and pass the name of the data type, a filter string, and an update expression to the function as input parameters. The filter string specifies which data items to update. It uses the SQL filter syntax, which is similar to the syntax of the WHERE clause in an SQL SELECT statement. The update expression is a comma-separated list of field assignments similar to the contents of the SET clause in an SQL UPDATE statement. For more information about SQL filters, see “SQL filters” on page 79.

**Syntax**

The BatchUpdate function has the following syntax:

NumberofUpdates = BatchUpdate(DataType, Filter, UpdateExpression)

**Parameters**

The BatchUpdate function has the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DataType</strong></td>
<td>String</td>
<td>Name of the data type.</td>
</tr>
<tr>
<td><strong>Filter</strong></td>
<td>String</td>
<td>SQL filter string that specifies which data items to update.</td>
</tr>
<tr>
<td><strong>UpdateExpression</strong></td>
<td>String</td>
<td>Expression that specifies the fields to update and the corresponding updated values. The expression is a comma-separated list of field assignment similar to the SET clause in an SQL UPDATE statement.</td>
</tr>
</tbody>
</table>

**Return value**

This function returns a Num value, that is the number of rows that were updated.

**Example**

The following example shows how to update field values in a set of data items. In this example, you update the Location and Facility fields of items in a data type named Server.

// Call BatchUpdate and pass the name of the data type, a filter string and // an update expression as input parameters

```java
DataType = "Server";
Filter = "Location = 'New York'";
```
UpdateExpression = "Location = 'Raleigh', Facility = 'SE_0014';
BatchUpdate(DataType, Filter, UpdateExpression);

**BeginTransaction**

The BeginTransaction is a local transactions function that is used in SQL operations.

You use this function to start the transaction. This function is used a policy in conjunction with other local transactions functions.

*For more information about the local transactions functions, see [Chapter 6, “Using local transactions in a policy,” on page 49.](#)*

**Arguments**

The BeginTransaction() function takes no arguments.

**Note:** The ObjectServer does not support the use of the BeginTransaction function.

**CallDBFunction**

The CallDBFunction function is a local transactions function that is used in SQL operations.

You can use CallDBFunction with SQL database data types. You cannot use this function with internal, LDAP, or Mediator data types.

To call the function, call CallDBFunction and pass the name of a data type, a filter string, and the function expression as input parameters. The data type identifies the underlying SQL database where the function is to be run. The function expression is the function call that is to be run by the database. CallDBFunction returns the value that results from the function.

Using CallDBFunction is equivalent to running the following SQL statement against the database:

```
SELECT function FROM table WHERE filter
```

where *function* is the specified function expression, *table* is the data type name, and *filter* is the filter string.

**Syntax**

CallDBFunction has the following syntax:

```
Integer | Float | String | Boolean = CallDBFunction(DataType, Filter, Metric)
```

**Parameters**

The CallDBFunction function has the following parameters.

*Table 16. CallDBFunction function parameters*

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataType</td>
<td>String</td>
<td>Name of a data type associated with the underlying SQL database.</td>
</tr>
</tbody>
</table>
Table 16. CallDBFunction function parameters (continued)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filter</td>
<td>String</td>
<td>Filter string that specifies which data items in the data type to run the function against. Not required for all types of database functions. To run the function without consideration for specific rows of data in the associated database table, enter a filter string such as 0=0 that always evaluates to true.</td>
</tr>
<tr>
<td>Metric</td>
<td>String</td>
<td>Database function expression.</td>
</tr>
</tbody>
</table>

Return value

CallDBFunction returns the value that resulted from the database function.

Examples

The following example shows how to call a database function named NOW() and return the results of the function for use in a policy.

```plaintext
// Call CallDBFunction and pass the name of a data type, a filter // string and the function expression

DataType = "Server";
Filter = "0 = 0";
Metric = "NOW()";

DBTime = CallDBFunction(DataType, Filter, Metric);
```

CallStoredProcedure

The CallStoredProcedure function calls a database stored procedure.

You can use this function with Sybase, Microsoft SQL Server, DB2SQL, and Oracle databases.

For detailed instructions on calling stored procedures from within a policy, see Chapter 7, “Calling stored procedures from within a policy,” on page 53.

Syntax

The CallStoredProcedure function has the following syntax:

```
[Array =] CallStoredProcedure(DataSource, ProcedureName, Sp_Parameter)
```

Parameters

The CallStoredProcedure function has the following parameters.

Table 17. CallStoredProcedure function parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataSource</td>
<td>String</td>
<td>Name of the data source associated with the database.</td>
</tr>
</tbody>
</table>
Table 17. CallStoredProcedure function parameters (continued)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProcedureName</td>
<td>String</td>
<td>The ProcedureName parameter can be just the stored procedure name or it can also be the fully qualified name using the following naming convention: Catalog.Schema.ProcedureName. For example, AdventureWorks.HumanResources.uspUpdateEmployeePersonalInfo. In instances where you try to call the procedure with just the procedure name, for example, uspUpdateEmployeePersonalInfo and you get an exception when you run the function; you can use the fully qualified name instead. For example, AdventureWorks.HumanResources.uspUpdateEmployeePersonalInfo.</td>
</tr>
<tr>
<td>Sp_Parameter</td>
<td>Context</td>
<td>Context named Sp_Parameter that contains the input parameters for the stored procedure as a set of name/value pairs. You cannot substitute any other name for this parameter.</td>
</tr>
</tbody>
</table>

**Return value**

Array of contexts that contain the output for the stored procedure. Returned for Sybase stored procedures only.

**ClassOf**

The ClassOf function returns the data type of a variable.

**Syntax**

The ClassOf function has the following syntax:

\[
\text{String} = \text{ClassOf}(\text{Var})
\]

**Parameters**

The ClassOf function has the following parameter.

Table 18. ClassOf function parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Var</td>
<td>Variable</td>
<td>Variable whose format you want to obtain.</td>
</tr>
</tbody>
</table>

**Return value**

Data type name for the variable.

**Note:**

- If you pass an integer variable to ClassOf it returns as long in IPL and returns as double in JavaScript.
- If you pass a context variable to ClassOf, it returns as BindingsVarGetSettable in IPL and returns as JavaScriptScriptableWrapper in JavaScript.
- If you pass an OrgNode variable to ClassOf, it returns as OrgNode in IPL and returns as VarGetSettable in JavaScript.
Example

The following example shows how to return the data type of a variable:

```plaintext
MyString = "This is a string.";
MyType = ClassOf(MyString);
Log(MyType);
```

This example prints the following message to the policy log:

```
Parser Log: String
```

CommandResponse

Use the CommandResponse function to run interactive and non-interactive programs on both local and remote systems.

The CommandResponse function sends a series of commands to a system by using telnet, ssh, or tn3270 and then returns responses from the system to the policy for handling. The default communication protocol is telnet.

When you call the function, CommandResponse connects to a remote port on the system using the connection information that you specify and returns a new context that identifies the session. To send a command, you set the value of the context's SendCommand variable to the text of the command. Then, you assign one or more substrings that match the expected response to the context's ExpectList variable. If the value of the system response matches one or more of the substrings in the ExpectList array, the value of the context's ResponseReceived variable is set to the full text of the response. To end the remote session, you set the value of the context's Disconnect variable to true.

The CommandResponse function is similar to the functionality provided by the Expect utility. For information about concepts that are related to this tool, see the Expect website at [http://expect.nist.gov](http://expect.nist.gov)

If you are using the JRExecAction and Command Response functions, you are encouraged to sanitize strings before you use their values in these functions. You can use the functions Escape and Illegal to sanitize string input.

Syntax

The CommandResponse function has the following syntax:

```plaintext
Session = CommandResponse(Host, UserName, Password|UserCredentials, InitialPrompt|Options, Port, Timeout, Expiration)
```

Parameters

The CommandResponse function has the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>String</td>
<td>Host name or IP address of the remote system.</td>
</tr>
<tr>
<td>UserName</td>
<td>String</td>
<td>User name for a remote system account.</td>
</tr>
<tr>
<td>Password</td>
<td>String</td>
<td>Password for the remote system account. Not required if UserCredentials parameter is specified.</td>
</tr>
</tbody>
</table>
Table 19. CommandResponse function parameters (continued)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InitialPrompt</td>
<td>String</td>
<td>Substring that matches the last line in the initial response that is returned by the remote application. Using all or a subset of the expected service command line prompt generally provides the best results. If the prompt contains a trailing space, include it in the string that is passed as this parameter. For SSH and telnet if Options is specified, InitialPrompt is still required within it, for example Options.InitialPrompt.</td>
</tr>
<tr>
<td>Options</td>
<td>Context</td>
<td>Context that specifies an initial response string and other settings specific to the remote system connection. For more information, see Options.</td>
</tr>
<tr>
<td>Port</td>
<td>Integer</td>
<td>Port that is used by the remote application. 23 (telnet) is the default.</td>
</tr>
<tr>
<td>Timeout</td>
<td>Integer</td>
<td>Command timeout in seconds. If the remote system does not respond to a command in less than the timeout value, the session disconnects. Default is 60.</td>
</tr>
<tr>
<td>Expiration</td>
<td>Integer</td>
<td>Entire session expiration timeout in seconds. Must be greater than Timeout. Default is 600.</td>
</tr>
</tbody>
</table>

User credentials

You can specify user credentials and other options for the remote service by using the UserCredentials context.

You can set the following member variables in this context.

Table 20. UserCredentials context member variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Password</td>
<td>Password for the remote system account.</td>
</tr>
<tr>
<td>PassPhrase</td>
<td>An ssh passphrase. Required only if you are using public key authentication with ssh.</td>
</tr>
<tr>
<td>KeyFile</td>
<td>Location and name of the private key file on the system where Netcool/Impact is running. Required only if you are using public key authentication with ssh.</td>
</tr>
</tbody>
</table>

Options

You can specify session initiation options for telnet, ssh, and tn3270 connections by using the Options context.

You can set the following member variables in this context.

Table 21. Options context member variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service</td>
<td>Specifies the service that is running on the remote system. Possible values are telnet, ssh, and tn3270. Default is telnet.</td>
</tr>
</tbody>
</table>
Table 21. Options context member variables (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AutoInitiate</td>
<td>Specifies whether Netcool/Impact should automatically connect and login using the supplied host, port, user name, and password. Possible values are true or false. If set to true, Netcool/Impact automatically connects and logs in to the remote application. If set to false, the policy must manually initiate a connection and login by setting the CommandResponse context variables Connect and Login, successively, to true. Netcool/Impact ignores the actual values that are set to the Connect and Login variables. This syntax is used to provide consistency with other features of the policy language.</td>
</tr>
<tr>
<td>CommandTerminator</td>
<td>Specifies the syntax that is used to terminate a command sent to the remote system.</td>
</tr>
<tr>
<td>LoginPrompt</td>
<td>Specifies the login prompt sent by the service that is running on the remote system.</td>
</tr>
<tr>
<td>PasswordPrompt</td>
<td>Specifies the password prompt sent by the service that is running on the remote system.</td>
</tr>
</tbody>
</table>

 Defaults

The CommandResponse function uses the following connection defaults.

Table 22. CommandResponse defaults

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port</td>
<td>23</td>
</tr>
<tr>
<td>Timeout</td>
<td>60 (1 minute)</td>
</tr>
<tr>
<td>Expiration</td>
<td>600 (10 minutes)</td>
</tr>
<tr>
<td>Options.AutoInitiate</td>
<td>true</td>
</tr>
<tr>
<td>Options.CmdTerminator</td>
<td>\n</td>
</tr>
<tr>
<td>Options.LoginPrompt</td>
<td>Login</td>
</tr>
<tr>
<td>Options.PasswordPrompt</td>
<td>Password</td>
</tr>
</tbody>
</table>

 Return value

The CommandResponse function returns a context that identifies the remote session. This context has the following member variables.

Table 23. Session context member variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SendCommand</td>
<td>String</td>
<td>Used to send a command to the remote system.</td>
</tr>
<tr>
<td>ExpectList</td>
<td>String[]</td>
<td>Array of substrings that match the possible expected responses for a command.</td>
</tr>
<tr>
<td>MatchedPrompt</td>
<td>String</td>
<td>Substring from the ExpectList array that matched the response from the remote system.</td>
</tr>
<tr>
<td>ResponseReceived</td>
<td>String</td>
<td>Response that is received from the remote system.</td>
</tr>
</tbody>
</table>
Telnet

To connect to a remote system using telnet, you call CommandResponse and pass the host name or IP address, a user name and password, an initial prompt substring, and the port number that is used by the telnet service on the remote host. The default is 23. You can also pass a session timeout and expiration value in seconds. CommandResponse connects to the remote port on the system and returns a new context that identifies the session.

The following example shows how to send a command to a remote system using telnet. In this example, you connect to the telnet application that is running on port 23 on localhost and send an ls -l /tmp command.

```
// Call CommandResponse and pass the required values as
// input parameters
Host = localhost;
UserName = "demouser1";
Password = "demouser";
Port = 23;
Timeout = 30;
Expiration = 60;
InitialPrompt = "[demouser1@localhost ~]$ ";

Session = CommandResponse(Host, UserName, Password, InitialPrompt, Port, Timeout, Expiration);

// Set the value of the SendCommand variable to the text of the command
// that you want to execute on the remote system
Session.SendCommand = "ls -l /tmp";

// Set the value of the ExpectList variable to a substring that matches
// the expected value of the string returned by the remote host
Session.ExpectList = {"[demouser1@localhost ~]$ ";

Log(Session.ResponseReceived);

// Disconnect from the remote host
Session.Disconnect = true;
```

SSH

To connect to a remote system using ssh, you call CommandResponse and pass the host name or IP address, a UserCredentials context, an Options context, and the port number that is used by the ssh service on the remote host (default is 22). You can also pass a session timeout and expiration value in seconds.

For ssh connections, the UserCredentials context specifies a login password for the remote service and, optionally, an SSH passphrase, and the location of an RSA or DSA private key file on the system where the Netcool/Impact server is running. The Options context specifies the service to use (in this case, ssh) and the initial response options for the connection. Use of public key authentication is optional.

The CommandResponse function only supports the SSH2 protocol. SSH1 is not supported.

If you want to use public key authentication with ssh and the CommandResponse function, you must first generate a public/private key pair using the ssh-keygen
utility. The following command session example shows how to generate the key pair. In this example, the utility creates the key files in the $IMPACT_HOME/ssh directory.

```
cd $IMPACT_HOME
mkdir .ssh
chmod 700 .ssh
ssh-keygen -q -f .ssh/id_rsa -t rsa
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
```

Do not provide an empty passphrase at the prompt.

After you generate the public/private key pair, you must copy the public key to the remote system where you plan to connect using ssh. You can identify the public key file that is generated by the ssh-keygen utility by its file name suffix, which is .pub. In the previous example, ssh-keygen creates a public key file named id_rsa.pub in the $IMPACT_HOME/ssh directory.

To copy the public key to the remote system, first transfer the file to a temporary directory on the system. Then append the file contents to the authorized keys file in the home directory of the user account on the system that you plan to use for authentication. This file is named authorized_keys and is in the $HOME/.ssh directory.

When you run the ssh-keygen utility, you specify the name and location of the private key file as a command argument. In the previous example, the name and location are ./ssh/id_rsa. You pass this name and location as part of the Options context when you call the CommandResponse function.

For more information about ssh and public key authentication, see the OpenSSH website at http://www.openssh.com.

The following example shows how to send a command to a remote system using ssh. In this example, you use a UserCredentials context to specify the password for the login user and an Options context to specify the service, which is ssh, the command prompt terminator, and an initial prompt string.

```
// Call CommandResponse and set the required values as input parameters

Host = "localhost";
UserName = "demouser1";
Port = 22;
Timeout = 30;
Expiration = 60;

// Create new UserCredentials context to pass to CommandResponse
// and populate the Password member variable

UserCredentials = NewObject();
UserCredentials.Password = "p4ssw0rd";

// Create new Options context to pass to CommandResponse and
// populate the Service, CmdTerminator and InitialPrompt member
// variables

Options = NewObject();
Options.Service = "ssh";
Options.CmdTerminator = "\n";
Options.InitialPrompt = "[demouser1@localhost ~]# ";
```
Session = CommandResponse(Host, UserName, UserCredentials, Options, Port, Timeout, Expiration);

// Send a command to the ssh session
Session.SendCommand = "ls -l /tmp";

// Set the value of the ExpectList variable to a substring that
// matches the expected response
Session.ExpectList = {"[demouser1@localhost ~]$ "};

// Print the response to the policy log
Log(Session.ResponseReceived);

// Close the ssh session
Session.Disconnect = true;

The following example shows how to send a command to a remote system by
using ssh and public key authentication. In this example, you use a
UserCredentials context to specify the password for the login user, the
authentication passphrase, and the location of the private key file.
You use an Options context to specify the service, which is ssh and an initial prompt string.
// Call CommandResponse and set the required values as input parameters

Host = "localhost";
UserName = "demouser";

// Create new UserCredentials context to pass to CommandResponse
// and populate the Password, KeyFile and PassPhrase member variables
UserCredentials = NewObject();
UserCredentials.Password = "p4ssw0rd";
UserCredentials.KeyFile = "./.ssh/id_rsa";
UserCredentials.PassPhrase = "p4ssphr4se";

// Create new Options context to pass to CommandResponse and
// populate the Service and InitialPrompt member
// variables
Options = NewObject();
Options.Service = "ssh";
Options.InitialPrompt = "[demouser1@localhost ~]$ ";

Session = CommandResponse(Host, UserName, UserCredentials, Options, null, null, null);

// Send a command to the ssh session
Session.SendCommand = "ls -l /tmp";

// Set the value of the ExpectList variable to a substring that
// matches the expected response
Session.ExpectList = {"[demouser1@localhost ~]$ "};

// Print the response to the policy log
Log(Session.ResponseReceived);
// Close the ssh session
Session.Disconnect = true;

tn3270

To connect to a remote system by using tn3270, you call CommandResponse and pass the host name or IP address, a UserCredentials context, an Options context, and the port number that is used by the tn3270 service on the remote host. The default is 23. You can also pass a session timeout and expiration value in seconds.

For tn3270 connections, the UserCredentials context specifies a login password for the remote service. The Options context specifies the service to use (in this case, tn3270) and the initial response options for the connection.

IPL and JavaScript provide a set of key strings that you can use to send special characters to the tn3270 session. You can use these key strings to move the input cursor and perform other actions when you send information using tn3270. For example, you can use key strings to pass a tab character at the end of the user name provided to CommandResponse for account login. If you are using tn3270 to interact with IBM Tivoli zNetview, this moves the cursor to the end of the user name string before it is entered.

You can use the following key strings with tn3270 and the CommandResponse function.

Table 24. CommandResponse key strings

<table>
<thead>
<tr>
<th>Special character</th>
<th>Key string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enter</td>
<td>[enter]</td>
</tr>
<tr>
<td>Tab</td>
<td>[tab]</td>
</tr>
<tr>
<td>F1</td>
<td>[pf1]</td>
</tr>
<tr>
<td>F2</td>
<td>[pf2]</td>
</tr>
<tr>
<td>F3</td>
<td>[pf3]</td>
</tr>
<tr>
<td>F4</td>
<td>[pf4]</td>
</tr>
<tr>
<td>F5</td>
<td>[pf5]</td>
</tr>
<tr>
<td>F6</td>
<td>[pf6]</td>
</tr>
<tr>
<td>F7</td>
<td>[pf7]</td>
</tr>
<tr>
<td>F8</td>
<td>[pf8]</td>
</tr>
<tr>
<td>F9</td>
<td>[pf9]</td>
</tr>
</tbody>
</table>

The following example shows how to interact with zNetview on a remote system using tn3270. You use a UserCredentials context to specify the password for the login user and an Options context to specify the service, which is tn3270, the login, password, and initial prompts, and the command prompt terminator.

// Call CommandResponse and set the required values as input parameters

Host = "localhost";
UserName = "demouserTAB_KEY";
Port = 23;
Timeout = 120;
Expiration = 240;
// Create new UserCredentials context to pass to CommandResponse
// and populate the Password member variable
UserCredentials = NewObject();
UserCredentials.Password = "Your password";

// Create new Options context to pass to CommandResponse and
// populate the Service, AutoInitiate, LoginPrompt, PasswordPrompt
// and CmdTerminator member variables
Options = NewObject();
Options.Service = "tn3270";
Options.AutoInitiate = false;
Options.LoginPrompt = "OPERATOR ID ==>";
Options.PasswordPrompt = "PASSWORD ==>";
Options.InitialPrompt = "Action===>";
Options.CmdTerminator = "ENTER_KEY";

Session = CommandResponse(Host, UserName, UserCredentials, Options, Port, Timeout, Expiration);
Session.Connect = true;
Session.ExpectList = {"SELECTION ==>" };
Session.SendCommand = "netview";
Session.Login = true;
Session.SendCommand = "nldm list";
Session.ExpectList = {"CMD==>"};
Log(Session.ResponseReceived);

// Close the tn3270 session
Session.Disconnect = true;

CommitChanges

The CommitChanges function is used in connection with the GetByFilter and
GetByKey functions to force updates in a database.

Using CommitChanges with the GetByFilter and GetByKey functions

The GetByFilter function retrieves data items from a data type using a filter as the
query condition. The GetByKey function retrieves data items from a data type using
a key expression as the query condition.

When data items are assigned to the built-in DataItem or OrgNode variable, they are
not immediately updated but are stored in a queue to optimize the number of calls
to the database. So, for example, if you update multiple fields in the DataItems
variable there will only be one call to update the underlying database, when a
function call is made.

To force all queued updates, call the CommitChanges() function in your policy.

Arguments

The CommitChanges() function takes no arguments.
CommitTransaction

The CommitTransaction function is a local transactions function that is used in SQL operations.

You use this function to commit the changes to the database. If the RollbackTransaction() function is not called, the CommitTransaction() function commits the changes to the database. If the RollbackTransaction() is called, the changes are undone and an internal flag is set to Auto Commit any future SQL operations.

You must always call the CommitTransaction() function to complete a transaction even if the RollbackTransaction() function is called.

For more information about the local transactions functions, see Chapter 6, “Using local transactions in a policy,” on page 49

Arguments

The CommitTransaction() function takes no arguments.

Note: The ObjectServer does not support the use of the CommitTransaction function.

ConvertObjectsToJSON

This function converts Netcool/Impact Objects to a one level JSON string.

Syntax

The ConvertObjectsToJSON function has the following syntax:

result = ConvertObjectsToJSON(Impact_Objects);

Parameters

The ConvertObjectsToJSON function has the following parameter.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Impact Object or Array Of Impact</td>
<td>An Impact Object.</td>
<td>Converts Impact Objects to JSON string that is</td>
</tr>
<tr>
<td>Objects</td>
<td></td>
<td>limited to small hierarchy.</td>
</tr>
</tbody>
</table>

Return value

The return value is a JSON String.

Examples

An IPL policy example to add an Object.

MyContext = NewObject();
MyContext.Identifier = GetDate();
subobj = NewObject();
MyContext.sub = subobj;
MyContext.sub.test = true;
mycars = {"Saab", "Volvo", "BMW"};
MyContext.sub.cars=mycars;
Log("MyContext: " + MyContext);
converted=ConvertObjectsToJSON(MyContext);
Log(converted);

A JavaScript policy example.

```javascript
EventContext = NewObject();
EventContext.servername = "NCOMS";
EventContext.serverserial = 555;
Obj = NewObject();
Obj.id = "0";
EventContext.added = Obj;
eventlist = [];
event = NewObject();
event.id = 1;
event.servername = "NCOMS";
event.serverserial = 555;

eventlist[0] = event;

event = NewObject();
event.id = 2;
event.servername = "NCOMS_BK";
event.serverserial = 666;

eventlist[1] = event;

event = NewObject();
event.id = 3;
event.servername = "NCOMS_VIRTUAL";
event.serverserial = 777;

eventlist[2] = event;

anotherlist = [];
o = NewObject();
o.Product = "Impact";
anotherlist[0] = o;
o = NewObject();
o.Company = "IBM";

anotherlist[1] = o;
o = NewObject();
o.Division = "Tivoli";

anotherlist[2] = o;

Log("anotherlist : " + anotherlist);
o = NewObject();
o.anotherlist = anotherlist;
eventlist[3] = o;
Log("eventlist : " + eventlist);
EventContext.eventlist = eventlist;

JSONResult = ConvertObjectsToJSON(EventContext);
Log(" JSONResult: " + JSONResult);
```

**ConvertXMLToImpactObjects**

The `ConvertXMLToImpactObjects` function converts any XML string to a nested structure of Impact objects.

The Impact object structure can be traversed to access any of the data in the XML string. The `ConvertXMLToImpactObjects` function can be used to convert the result.
of web services calls into a structure that can be easily traversed. The function can
also be used to parse any other XML data, without requiring xsd or schema
information. Text inside a CDATA section, defined by the characters <![CDATA [ ]]>),
is added to the Impact objects that are created by the ConvertXMLToImpactObjects
function. Text inside the following XML tags is not added to the Impact objects.

Comments
ProcessingInstruction
DocType
Entity
Notation

The XML parser does use the rules in the DocType, Entity, and Notation tags to
assign substitutions in the XML, but the XML parser does not add the text from
these tags to the Impact objects.

The rules for converting elements and attributes to objects are as follows:
• All elements except for leaf elements are represented as arrays of Impact objects.
• Attributes of elements are referenced by the name of the attribute on the element
 object.
• Leaf elements will not be arrays unless there are multiple elements with the
 same name.
• If a leaf element has no attributes, the text data of the leaf element is referenced
 by the name of the leaf element.
• If a leaf element has attributes, text data of the element is referenced by the
 CDATA field in the element object.

Syntax

The ConvertXMLToImpactObjects function has the following syntax.
Result = ConvertXMLToImpactObjects(xmlstring);

Parameters

The ConvertXMLToImpactObjects function has the following parameter.

Table 26. ConvertXMLToImpactObjects function parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmlstring</td>
<td>String</td>
<td>A string of xml data.</td>
</tr>
</tbody>
</table>

Examples

This function can be used to convert the result of web services calls into a structure
that can be easily traversed. The following XML string is used as an example to
explain how the conversion is done.

xmns:xsd="http://www.w3.org/2001/XMLSchema">
<GetWeatherInformationResult>
  <WeatherDescription>
    <WeatherID>1</WeatherID>
    <Description region="US">Thunder Storms</Description>
    <PictureURL>http://ws.cdyne.com/WeatherWS/Images/thunderstorms.gif</PictureURL>
  </WeatherDescription>
</GetWeatherInformationResult>
If this string is in a variable in the policy called xmlstring, you can call:

```
Result = ConvertXMLToImpactObjects(xmlstring);
```

Result object contains the xml data in the Impact objects.

```
Result.GetWeatherInformation[0].GetWeatherInformationResult[0].
WeatherDescription[0].WeatherID
```

Returns the value 1.

```
Result.GetWeatherInformation[0].xmlns
```

Returns the value http://ws.cdyne.com/WeatherWS/

```
Result.GetWeatherInformation[0].GetWeatherInformationResult[0].
WeatherDescription[1].Description.CDATA
```

Returns the value Partly Cloudy.

To iterate through arrays of elements, you can use a while loop:

```
WeatherDescriptionArray = Result.GetWeatherInformation[0].
GetWeatherInformationResult[0].
WeatherDescription;
// WeatherDescriptionArray is an array of "WeatherDescription" element objects).
  i = 0;
  while (WeatherDescriptionArray[i] <> NULL) {
    WeatherDescriptionObject = WeatherDescriptionArray[i];
    log("PictureURL for element "+i+" is "+
      + WeatherDescriptionObject.PictureURL);
    i = i + 1;
  }
```

When element or attribute names contain non-alphanumeric characters, you need to provide a different syntax to access the data in a policy. In the previous example, there is an attribute on the Result.GetWeatherInformation element called xmlns:xsd. Since this attribute name contains a colon, you cannot reference the data by using Result.GetWeatherInformation[0].xmlns:xsd. However, you can access it by using Result.GetWeatherInformation[0]["xmlns:xsd"].

---

**CurrentContext**

The CurrentContext function returns the current policy context.

The policy context consists of all of the currently defined variables in the policy, including EventContainer, DataItems, DataItem, and Num.

**Important:** This function must be used only in a Log statement and it cannot be assigned to a variable.

**Syntax**

The CurrentContext function has the following syntax:

```
CurrentContext()
```
Example

This example shows how to return the current policy context.
Log(CurrentContext());

This example prints the member variables of the context and their values to the policy log.

Decrypt

The Decrypt function decrypts a string that has been previously encrypted by using Encrypt or the nci_crypt tool.

Syntax

The Decrypt function has the following syntax:
String = Decrypt(Expression)

Parameters

The Decrypt function has the following parameter.

Table 27. Decrypt function parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>String</td>
<td>String to decrypt.</td>
</tr>
</tbody>
</table>

Return value

Decrypted string.

Example

This example shows how to decrypt a string for example a server name. The encrypt value uses the servers private key for encryption. The output is server specific.

MyString = Encrypt("Password");
Log("Encrypted Value " + MyString);

MyDecrypt = Decrypt(MyString);
Log("Decrypted Value " + MyDecrypt);

Output

Parser log: Encrypted Value {aes}52A2390EA2D80E0191E735915CCAC5FE
Parser log: Decrypted Value Password

DeleteDataItem

The DeleteDataItem function deletes a single data item from a data type.

To delete multiple data items, use the BatchDelete function. You can use DeleteDataItem with internal and SQL database data types.

Before you delete a data item, you must first retrieve it from the data type by calling GetByFilter, GetByKey, or GetByLinks. Then, call DeleteDataItem and pass it to the function as an input parameter.
**Syntax**

The `DeleteDataItem` function has the following syntax:

```
DeleteDataItem(DataItem)
```

**Parameters**

The `DeleteDataItem` function has the following parameter.

*Table 28. DeleteDataItem function parameters*

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DataItem</code></td>
<td>Data Item</td>
<td>Data item that you want to delete.</td>
</tr>
</tbody>
</table>

**Examples**

The following example shows how to delete a data item from a data type using the `DeleteDataItem` function. In this example, you retrieve the item from the data type using the `GetByFilter` function.

```c
// Call GetByFilter and pass the name of the data type and a filter string as input parameters

DataType = "Server";
Filter = "Name = 'ORA_01'";
CountOnly = false;

MyServers = GetByFilter(DataType, Filter, CountOnly);
MyServer = MyServers[0];

// Call DeleteDataItem and pass the data type as an input parameter

DeleteDataItem(MyServer);
```

**Deploy**

The Deploy function copies data sources, data types, policies, and services between server clusters.

You can use this function to write automated deployment policies that copy Impact Server data between test and production environments.

**Syntax**

The Deploy function has the following syntax:

```
Deploy(TargetCluster, Username, Password, Elements, ElementsOfType, CheckpointID)
```

**Parameters**

The Deploy function has the following parameters.

*Table 29. Deploy function parameters*

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TargetCluster</code></td>
<td>Name of the destination server cluster.</td>
</tr>
<tr>
<td><code>Username</code></td>
<td>Valid Netcool/Impact user name.</td>
</tr>
<tr>
<td><code>Password</code></td>
<td>Valid Netcool/Impact password.</td>
</tr>
</tbody>
</table>
Table 29. Deploy function parameters (continued)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements</td>
<td>String or array of strings that specify which project components to copy between server clusters.</td>
</tr>
<tr>
<td>ElementsOfType</td>
<td>String that specifies which type of project components to copy between server clusters. You can specify Project, DataSource, DataType, Policy, and Service.</td>
</tr>
<tr>
<td>CheckpointID</td>
<td>If you are using CVS and SVN version control system for Netcool/Impact, you can specify a checkpoint label. This label is applied to all project components when checked into the version control system for the target cluster. If you are not using Subversion or you do not want to use a checkpoint label, use the null value for this parameter.</td>
</tr>
</tbody>
</table>

In addition to the parameters, Deploy also optionally reads the following variables from the policy-level scope.

Table 30. Deploy function optional variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TargetNameserverHost</td>
<td>Host name or IP address of the system where the Name Server is running. This is typically a system where you are running the name server and GUI server as hosted applications.</td>
</tr>
<tr>
<td>TargetNameserverPort</td>
<td>HTTP port that is used by the Name Server. The default is 9080.</td>
</tr>
<tr>
<td>TargetNameserverLocation</td>
<td>URL path on the Java application server where the name server is located. The default is /nameserver/services.</td>
</tr>
<tr>
<td>NameserverSslEnabled</td>
<td>Specifies whether the communication with the name server is realized over SSL. Value can be true or false. The default is false.</td>
</tr>
</tbody>
</table>

You use these variables when you want to deploy project data from a Netcool/Impact cluster that uses one name server to a cluster that uses a different Name Server.

**Examples**

The following example shows how to copy a project and all its data sources, data types, policies, and services to a server cluster named NCI_PROD_01. In this example, the name of the project is PROJECT_01. Both clusters use the same instance of the Name Server.

```java
TargetCluster = "NCI_PROD_01";
Username = "impactadmin";
Password = "impactpass";
Elements = "PROJECT_01";
ElementsOfType = "Project";
CheckpointID = null;
Deploy(TargetCluster, Username, Password, Elements, ElementsOfType, CheckpointID);
```

The following example shows how to copy policies named POLICY_01, POLICY_02, and POLICY_03 to a server cluster named NCI_PROD_02. Both clusters use the same instance of the Name Server.

```java
TargetCluster = "NCI_PROD_02";
Username = "impactadmin";
Password = "impactpass";
Elements = "";
ElementsOfType = "Policy";
CheckpointID = null;
Deploy(TargetCluster, Username, Password, Elements, ElementsOfType, CheckpointID);
```
TargetCluster = "NCI_PROD_02";
Username = "impactadmin";
Password = "impactpass";
Elements = {"POLICY_01", "POLICY_02", "POLICY_03"};
ElementTypes = "Policy";
CheckpointID = null;

Deploy(TargetCluster, Username, Password, Elements, ElementTypes, CheckpointID);

The following example shows how to copy a project and all its data sources, data types, policies, and services to a server cluster named NCI_PROD_03. In this example, the name of the project is PROJECT_01. The target cluster here uses a different instance of the Name Server.

TargetCluster = "NCI_PROD_03";
Username = "impactadmin";
Password = "impactpass";
Elements = "PROJECT_01";
ElementTypes = "Project";
CheckpointID = null;

// Specify the host and port where the nameserver for the target cluster is located
TargetNameserverHost = "192.168.1.1";
TargetNameserverPort = 9080;

Deploy(TargetCluster, Username, Password, Elements, ElementTypes, CheckpointID);

DirectSQL

The DirectSQL function runs an SQL operation against the specified database and returns any resulting rows to the policy as data items.

You can use the DirectSQL function only with SQL database data types.

You can use this function to perform SELECT queries with JOIN clauses and to perform other operations that cannot be carried out using GetByKey, GetByFilter, or GetByLinks. This function supports SELECT, UPDATE and DELETE statements.

Syntax

The DirectSQL function has the following syntax:

```
[Array =] DirectSQL(DataSource, Query, CountOnly, ForceSelect
```

Parameters

The DirectSQL function has the following parameters.

**Table 31. DirectSQL function parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataSource</td>
<td>String</td>
<td>Name of the data source associated with the SQL database.</td>
</tr>
<tr>
<td>Query</td>
<td>String</td>
<td>SQL operation to run against the database.</td>
</tr>
<tr>
<td>CountOnly</td>
<td>Boolean</td>
<td>Pass a false value for this parameter. Provided only for backwards compatibility.</td>
</tr>
<tr>
<td>ForceSelect</td>
<td>Boolean</td>
<td>Force the DirectSQL function to run the query as a SELECT statement even though it does not start with a SELECT keyword.</td>
</tr>
</tbody>
</table>
Return value

The DirectSQL function returns an array of data items where each data item represents a row returned from the database by the SQL query. Fields in the data items have a one-to-one correspondence with fields in the returned rows.

Examples

The following example shows how to run an SQL SELECT operation with a JOIN clause against a database.

```java
// Call DirectSQL and pass the name of the data source and the SQL SELECT statement as input parameters
DataSource = "MYSQL_01";
Query = "SELECT * FROM Customer LEFT JOIN Server ON " + "Customers.Location = Server.Location";
CountOnly = false;
MyCustomers = DirectSQL(DataSource, Query, CountOnly);
```

The following example shows how to run an SQL UPDATE operation against a database.

```java
// Call DirectSQL and pass the name of the data source and the SQL statement as input parameters
DataSource = "MYSQL_02";
Query = "UPDATE Customer SET Affected = true WHERE Location = 'New York'";
CountOnly = false;
DirectSQL(DataSource, Query, CountOnly);
```

The following example shows how to run an SQL DELETE operation against a database.

```java
// Call DirectSQL and pass the name of the data source and the SQL statement as input parameters
DataSource = "MYSQL_03";
Query = "DELETE FROM Customer WHERE Location = 'New York'";
CountOnly = false;
DirectSQL(DataSource, Query, CountOnly);
num_rows_deleted = Num;
Log("Number of deleted rows: " + num_rows_deleted);
```

This query deletes the specified records and returns the number of rows deleted.

Caching on SELECT Statement

The DirectSQL function supports caching when used to run a SELECT statement that returns a result set from a database. Caching is configured separately for each data source.

To enable caching, you must edit the DirectSQL properties file. This file is named `servername_directsql.props`, where `servername` is the name of the Impact Server. The file is located in the `$IMPACT_HOME/etc` directory. The following table shows the properties in this file.

You must replace the string in each property with the data source number as represented in the data source list. The data source list is a file named `servername_datasourcelist` and is located in the `$IMPACT_HOME/etc` directory.
Table 32. DirectSQL Caching Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>impact.datasource.n.enablecaching</td>
<td>Specifies whether caching is enabled for this data source. Value can be true or false.</td>
</tr>
<tr>
<td>impact.datasource.n.cachesize</td>
<td>Maximum number of rows per query to be cached.</td>
</tr>
<tr>
<td>impact.datasource.n.querycachesize</td>
<td>Maximum number of queries to be cached.</td>
</tr>
<tr>
<td>impact.datasource.n.cacheinvalidation</td>
<td>Amount of time in seconds before data items in the cache are considered stale and must be refreshed from the data source.</td>
</tr>
<tr>
<td>impact.datasource.n.querycacheinvalidation</td>
<td>Amount of time in seconds before queries in the cache are considered stale and must be refreshed from the data source.</td>
</tr>
</tbody>
</table>

**Distinct**

The Distinct function returns an array of distinct elements from another array.

**Syntax**

The Distinct function has the following syntax:

\[ Array = \text{Distinct}(Array, [UniqueClause]) \]

**Parameters**

The Distinct function has the following parameters.

Table 33. Distinct function parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>Array</td>
<td>Array whose distinct elements you want to obtain.</td>
</tr>
<tr>
<td>UniqueClause</td>
<td>String</td>
<td>A string expression that specifies which fields must be different in all data items to return items in the array as part of the distinct result set. The format of this expression is one or more field names separated by the plus sign (+). Optional.</td>
</tr>
</tbody>
</table>

**Return value**

An array of distinct elements.

**Note:** In JavaScript, Integers return as Float instead of integers, for example 1 is 1.0.

**Examples**

The following example shows how to return an array of distinct elements from another array using IPL.

\[ \text{MyArray} = \text{Distinct}(["a", "a", "b", "b", "c"]); \]
\[ \text{Log(MyArray);} \]
This example prints the following message to the policy log:
Parser Log: \{a, b, c\}

The following example shows how to return an array of distinct elements from another array using JavaScript
MyArray=Distinct(["a", "a", "b", "b", "c"]);
Log(MyArray);

This example prints the following message to the policy log:
Parser log: ["a", "b", "c"]

The following example shows how to return an array of distinct elements from an array of data items. In this example, you use the UniqueClause parameter to specify that all distinct elements returned must have different value in the Node and Class fields.
MyArray = Distinct(DataItems, "Node+Class");

---

**Encrypt**

The Encrypt function encrypts a string.

**Syntax**

The Encrypt function has the following syntax:

\[ String = Encrypt(Expression) \]

**Parameters**

The Encrypt function has the following parameter.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>String</td>
<td>String to encrypt.</td>
</tr>
</tbody>
</table>

**Return value**

An encrypted string.

**Example**

The following example shows how to encrypt a string. The encrypt value uses the servers private key for encryption. The output is server specific.

MyString = Encrypt("Password");
Log(MyString);

This example prints the following message to the policy log:
Parser Log: AB953E4925B39218F390AD2E812E81A

---

**Escape**

This function escapes special characters in an input string in a policy.
**Syntax**

The Escape function has the following syntax:

\[
\text{OutputString} = \text{Escape} (\text{InputString}, \text{ArrayOfCharsToEscape})
\]

**Parameters**

The Escape function has the following parameters:

*Table 35. Escape function parameters. A table of the Escape function parameters.*

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InputString</td>
<td>String</td>
<td>The Input String to escape special characters</td>
</tr>
<tr>
<td>ArrayOfCharsToEscape</td>
<td>String</td>
<td>The Array of special characters to Escape in the Input String</td>
</tr>
</tbody>
</table>

**Return value**

An escaped string.

**Examples**

Example 1

MyString = "That's all folks!";
Log("Original - MyString: " + MyString);
MyString = Escape(MyString, {"\", "});
Log("Escaped - MyString: " + MyString);

March 12, 2013 10:46:01 PM EDT[PolicyLogger][EscapeTest][pool-1-thread-13]
Parser log: Original - MyString: That's all folks!
March 12, 2013 10:46:01 PM EDT[PolicyLogger][EscapeTest][pool-1-thread-13]
Parser log: Escaped - MyString: That\'s all folks!

Example 2

MyPath = "C:\Program Files\IBM\Tivoli";
Log("Original - MyPath: " + MyPath);
MyPath = Escape(MyPath, {"\", "});
Log("Escaped - MyPath: " + MyPath);

March 12, 2013 10:46:01 PM EDT[PolicyLogger][EscapeTest][pool-1-thread-13]
Parser log: Original - MyPath: C:\Program Files\IBM\Tivoli
March 12, 2013 10:46:01 PM EDT[PolicyLogger][EscapeTest][pool-1-thread-13]
Parser log: Escaped - MyPath: C:\\Program Files\\IBM\\Tivoli

**Eval**

The Eval function evaluates an expression by using the specified context.

**Syntax**

The Eval function has the following syntax:

\[
\text{Integer} | \text{Float} | \text{String} | \text{Boolean} = \text{Eval} (\text{Expression}, \text{Context})
\]
Note: In JavaScript, the Float variable returns extra precision, for example, 10.695671999999998 instead of 10.695672. In IPL, integer division of 10/5 is 2.0. In JavaScript, integer division of 10/5 is 2.

Parameters

The Eval function has the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>String</td>
<td>Expression to evaluate.</td>
</tr>
<tr>
<td>Context</td>
<td>Context</td>
<td>Context to use in evaluating the expression.</td>
</tr>
</tbody>
</table>

Return value

Result of the evaluated expression.

Example

The following example shows how to evaluate an expression using the given context.

MyContext = NewObject();
MyContext.a = 5;
MyContext.b = 10;
MyResult = Eval("a + b", MyContext);
Log(MyResult);

This example prints the following message to the policy log:
Parser Log: 15

EvalArray

The EvalArray function evaluates an expression by using the specified array.

Syntax

The EvalArray function has the following syntax:

```
Integer | Float | String | Boolean = EvalArray(Expression, Array)
```

Parameters

The EvalArray function has the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>String</td>
<td>Expression to evaluate.</td>
</tr>
<tr>
<td>Array</td>
<td>array</td>
<td>Array to use in evaluating the expression.</td>
</tr>
</tbody>
</table>

Return value

Result of the evaluated expression.
Note: In JavaScript, Integers return as Float instead of integers, for example 1 is 1.0.

Example

The following example shows how to evaluate an expression using the given array.

MyNode1 = NewObject();
MyNode1.Name = "ORA_01";
MyNode1.Location = "New York";

MyNode2 = NewObject();
MyNode2.Name = "ORA_02";
MyNode2.Location = "New York";

MyNodes = {MyNode1, MyNode2};

MyEval = EvalArray("Name: " + Name + ", Location: " + Location', MyNodes);
Log(MyEval);

This example prints the following message to the policy log:
Parser Log: Name: ORA_01, Location: New York, Name: ORA_02, Location: New York

Exit

You use the Exit function to stop a function anywhere in a policy or to exit a policy.

The Exit function works differently in IPL and JavaScript. In IPL, when you use Exit in a user-defined function it exits that function, and the policy continues. In JavaScript, when you use Exit in a user-defined function in a policy it exits the entire policy. If you want to stop a function in a JavaScript policy you must use the return command in the policy.

Syntax

The Exit function has the following syntax:

Exit()

Examples

In this example, the value of the X variable is tested. If X is greater than ten, the policy terminates. If X is less than ten, it prints a message to the policy log. The following example is valid for IPL and JavaScript.

function testX() {
  X = 15;
  if (X > 10) {
    Log("Exiting if statement");
    Exit();
  } else {
    Log("X is less than 10.");
  }
}

testX();
Log("End of policy");

Output for IPL
Exiting if statement
End of policy

Output for JavaScript
Exiting if statement

The following example shows the use of the Exit function in IPL:
Log("Entering Policy TestExit....");
SetGlobalVar("exitFunction","false");
SetGlobalVar("exitPolicy","false");

function testExit(test){
    SetGlobalVar("exitPolicy",test);
    if (test = true){
        Log("Exiting function TestExit....");
        Exit();
    }else{
        Log("Staying in the Policy TestExit....");
    }
}

//Passing true will exit the function testExit and exit the policy
//on the second call(below)to Exit.
//Passing false will allow the function and policy to finish to the end.
testExit(false);
if(""+GetGlobalVar("exitPolicy")== "true"){
    Log("Exiting policy....");
    Exit();
}
Log("If you see this message, the policy continued to the end....");

The following example shows the use of the Exit and return functions in JavaScript:
Log("Entering Policy TestExit....");
SetGlobalVar("exitFunction","false");
SetGlobalVar("exitPolicy","false");

function testExit(test){
    SetGlobalVar("exitPolicy",test);
    if (test == true){
        Log("Exiting function TestExit AND policy....");
        Exit();
    }else{
        Log("Staying in the Policy TestExit....");
        return;
        Log("I will not see this log statement as we have already returned");
    }
}

//Passing true will immediately exit the function testExit AND the policy.
//Passing false will allow the function and policy to finish to the end.
testExit(true);
Log("If you see this message, the policy continued to the end....");

---

Extract

The Extract function extracts a word from a string.
Syntax

The `Extract` function has the following syntax:

\[ \text{String} = \text{Extract(Expression, Index, [Delimiter])} \]

Parameters

The `Extract` function has the following parameters.

**Table 38. Extract function parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>String</td>
<td>String expression from which to extract strings.</td>
</tr>
<tr>
<td>Index</td>
<td>Integer</td>
<td>Word position of the substring, where 0 indicates the first word.</td>
</tr>
<tr>
<td>Delimiter</td>
<td>array</td>
<td>Array of characters that separate words in the string. Default is an array with the space character as the single element.</td>
</tr>
</tbody>
</table>

Return value

The extracted substring.

Example

The following example shows how to extract a word from a string.

```plaintext
MyString = "This is a test.";
MyWord = Extract(MyString, 1, " ");
Log(MyWord);

MyString = "This|is|a|test.";
MyWord = Extract(MyString, 3, "|");
Log(MyWord);
```

This example prints the following message to the policy log:

Parser Log: is
Parser Log: test.

Float

The `Float` function converts an integer, string, or Boolean expression to a floating point number.

Syntax

The `Float` function has the following syntax:

\[ \text{Float} = \text{Float(Expression)} \]

Parameters

The `Float` function has the following parameter.

**Table 39. Float function parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>Integer</td>
<td>String</td>
</tr>
</tbody>
</table>
Return value

The converted floating point number.

Example

The following example shows how to convert integers, strings, and Boolean expressions to float point numbers using IPL.

```
MyFloat = Float(25);
Log(MyFloat);

MyFloat = Float("25.12");
Log(MyFloat);

MyFloat = Float(true);
Log(MyFloat);
```

This example prints the following message to the policy log:

```
Parser Log: 25.0
Parser Log: 25.12
Parser Log: 1.0
```

The following example shows how to convert integers, strings, and Boolean expressions to float point numbers using JavaScript.

```
MyFloat = Float(25);
Log(MyFloat);

MyFloat = Float("25.12");
Log(MyFloat);

MyFloat = Float(true);
Log(MyFloat);
```

JavaScript does not add any decimal points to the results for integers and Boolean expressions that are used with the Float function.

This example prints the following message to the policy log:

```
Parser Log: 25
Parser Log: 25.12
Parser Log: 1
```

FormatDuration

The FormatDuration function converts a duration in seconds into a formatted date/time string.

Syntax

The FormatDuration function has the following syntax:

```
String = FormatDuration(Seconds)
```
Parameters

The FormatDuration function has the following parameter.

Table 40. FormatDuration function parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seconds</td>
<td>Integer</td>
<td>Number of seconds.</td>
</tr>
</tbody>
</table>

Return value

Formatted date/time string.

Example

The following example shows how to convert seconds into a formatted date/time strings.

Seconds = "41927";
Duration = FormatDuration(Seconds);
Log(Duration);

This example prints the following to the policy log:
Parser Log: 11:38:47s

GetByFilter

The GetByFilter function retrieves data items from a data type by using a filter as the query condition.

To retrieve data items using a filter condition, you call GetByFilter and pass the data type name and the filter string as input parameters. The syntax for the filter string varies depending on whether the data type is an internal, SQL database, LDAP, or Mediator data type.

GetByFilter returns an array of references to the retrieved data items. If you do not assign the returned array to a variable, the function assigns it to the built-in DataItems variable and sets the value of the Num variable to the number of data items in the array.

You can use GetByFilter with internal, SQL database, and LDAP data types. You can also use GetByFilter with some Mediator data types.

Important: When data items are assigned to the built-in DataItem variable, they are not immediately updated but are stored in a queue to optimize the number of calls to the database. So, for example, if you update multiple fields in the DataItems variable there will only be one call to update the underlying database, when a function call is made. To force all queued updates, call the CommitChanges() function in your policy. The CommitChanges() function does not take any arguments.

Syntax

The GetByFilter function has the following syntax:

[Array =] GetByFilter(DataType, Filter, [CountOnly])
Parameters

The `GetByFilter` function has the following parameters.

**Table 41. GetByFilter function parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DataType</code></td>
<td>String</td>
<td>Name of the data type.</td>
</tr>
<tr>
<td><code>Filter</code></td>
<td>String</td>
<td>Filter expression that specifies which data items to retrieve from the data type.</td>
</tr>
<tr>
<td><code>CountOnly</code></td>
<td>Boolean</td>
<td>Pass a false value for this parameter. Provided for compatibility with earlier versions only.</td>
</tr>
</tbody>
</table>

**Return value**

Array of references to the retrieved data items. Optional.

**Examples**

The following example shows how to retrieve data items from an internal or SQL database data type.

```plaintext
// Call GetByFilter and pass the name of the data type
// and an SQL database filter expression

DataType = "Admin";
Filter = "Level = 'Supervisor' AND Location LIKE 'NYC.*";
CountOnly = false;

MyAdmins = GetByFilter(DataType, Filter, CountOnly);
```

The following example shows how to retrieve data items from an LDAP data type.

```plaintext
// Call GetByFilter and pass the name of the data type
// and an LDAP filter expression

DataType = "Customer";
Filter = "(|(facility=NYC)(facility=NNJ))";
CountOnly = false;

MyCustomers = GetByFilter(DataType, Filter, CountOnly);
```

The following example shows how to retrieve data items from a Mediator data type.

```plaintext
// Call GetByFilter and pass the name of the data type
// and the Mediator filter expression

DataType = "SWNetworkElement";
Filter = "ne_name = 'DSX1 PNL-01 (ORP)";
CountOnly = false;

MyElements = GetByFilter(DataType, Filter, CountOnly);
```

**GetByKey**

The `GetByKey` function retrieves data items from a data type by using a key expression as the query condition.
To retrieve data items by key, you call `GetByKey` and pass the name of the data item and a key expression. The key expression varies depending on whether you want the data items to match a single key or multiple keys.

`GetByKey` returns an array of references to the retrieved data items. If you do not specify a return variable, the function assigns the array to the built-in `DataItems` variable and sets the value of the `Num` variable to the number of data items in the array.

You can use `GetByKey` with internal, SQL database, and LDAP data types. You can also use `GetByKey` with some Mediator data types.

**Important:** When data items are assigned to the built-in `DataItem` variable, they are not immediately updated but are stored in a queue to optimize the number of calls to the database. So, for example, if you update multiple fields in the `DataItems` variable there will only be one call to update the underlying database, when a function call is made. To force all queued updates, call the `CommitChanges()` function in your policy. The `CommitChanges()` function does not take any arguments.

### Syntax

The `GetByKey` function has the following syntax:

```
[Array =] GetByKey(DataType, Key, [MaxNum])
```

### Parameters

The `GetByKey` function has the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DataType</code></td>
<td>String</td>
<td>Name of the data type.</td>
</tr>
<tr>
<td><code>Key</code></td>
<td>Integer</td>
<td>Float</td>
</tr>
<tr>
<td><code>MaxNum</code></td>
<td>Integer</td>
<td>Maximum number of data items to retrieve. Default is 1. Optional.</td>
</tr>
</tbody>
</table>

### Return value

Array of references to the retrieved data items. Optional.

### Examples

The following example shows how to retrieve data items from a data type using a single key.

```csharp
// Call GetByKey and pass the name of the data type,
// the key expression and the maximum number of data items
// to return.

DataType = "Customer";
Key = "12345";
MaxNum = 1;

MyCustomers = GetByKey(DataType, Key, MaxNum);
```
The following example shows how to retrieve data items from a data type that has multiple key fields.

Example using IPL.
// Call GetByKey and pass the name of the data type, // the key expression and the maximum number of data // items to return.

```
DataType = "Node";
Key = {"R12345", "D98776");
MaxNum = 1;
MyCustomers = GetByKey(DataType, Key, MaxNum);
```

Example using JavaScript.
// Call GetByKey and pass the name of the data type, // the key expression and the maximum number of data // items to return.

```
DataType = "Node";
Key = ["R12345", "D98776");
MaxNum = 1;
MyCustomers = GetByKey(DataType, Key, MaxNum);
```

GetByLinks

The GetByLinks function retrieves data items in target data types that are linked to one or more source data items.

To retrieve data items by link, you must first retrieve source data items using GetByKey, GetByFilter, or another call to GetByLinks. Then you call GetByLinks and pass an array of target data types and the sources. The function returns an array of data items in the target data types that are linked to the source data items.

GetByLinks returns an array of references to the retrieved data items. If you do not specify a return variable, the function assigns the array to the built-in DataItems variable and sets the value of the Num variable to the number of data items in the array.

Important: When data items are assigned to the built-in DataItem variable, they are not immediately updated but are stored in a queue to optimize the number of calls to the database. So, for example, if you update multiple fields in the DataItems variable there will only be one call to update the underlying database, when a function call is made. To force all queued updates, call the CommitChanges() function in your policy. The CommitChanges() function does not take any arguments.

Syntax

The GetByLinks function has the following syntax:
```
[Array =] GetByLinks(DataTypes, [LinkFilter], [MaxNum], DataItems)
```
Parameters

The GetByLinks function has the following parameters.

Table 43. GetByLinks function parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataTypes</td>
<td>Array</td>
<td>Array of target data type names. The function returns data items of these data types that are linked to the source data items specified.</td>
</tr>
<tr>
<td>LinkFilter</td>
<td>String</td>
<td>Filter expression that specifies which linked data items to retrieve. Optional.</td>
</tr>
<tr>
<td>MaxNum</td>
<td>Integer</td>
<td>Maximum number of data items to retrieve. Default is 1. Optional.</td>
</tr>
<tr>
<td>DataItems</td>
<td>Array</td>
<td>Array of source data items.</td>
</tr>
</tbody>
</table>

Return value

Array of data items in the target data type that are linked to the source data item. Optional.

Examples

The following example shows how to retrieve data items linked to another data item.

Example using IPL.

// Call GetByLinks and pass the target data type, // the maximum number of data items to retrieve and // the source data item.

DataTypes = {"Location"};
Filter = "";
MaxNum = "10000";
DataItems = MyCustomers;

MyLocations = GetByLinks(DataTypes, Filter, MaxNum, DataItems);

Example using JavaScript.

// Call GetByLinks and pass the target data type, // the maximum number of data items to retrieve and // the source data item.

DataTypes = ["Location"];
Filter = "";
MaxNum = "10000";
DataItems = MyCustomers;

MyLocations = GetByLinks(DataTypes, Filter, MaxNum, DataItems);

The following example shows how to retrieve data items linked to another data item. In this example, the function filters the data items using the value of the Location field.

Example using IPL.

// Call GetByLinks and pass the target data type, // the link filter, the maximum number of data items // to retrieve and the source data item.
DataTypes = {"Operators"};
Filter = "Location = 'New York'";
MaxNum = "10000";
DataItems = MyCustomers;

MyOperators = GetByLinks(DataTypes, Filter, MaxNum, DataItems);

Example using JavaScript.
// Call GetByLinks and pass the target data type,
// the link filter, the maximum number of data items
// to retrieve and the source data item.
DataTypes = ["Operators"];
Filter = "Location = 'New York'";
MaxNum = "10000";
DataItems = MyCustomers;

MyOperators = GetByLinks(DataTypes, Filter, MaxNum, DataItems);

GetByXPath

The GetByXPath function provides a way to parse an XML string or get an XML string through a URL specified as parameter.

The data to be retrieved is specified as an XPath expression. When Netcool/Impact interacts with different systems there are scenarios where the data in Netcool/Impact is in XML format.

GetByXPath can be used in the following scenarios to retrieve the data for each source:
• The response from a Web services call. The GetByXPath function retrieves the data from a web service response.
• The response to a REST API call provides data in XML format. The GetByXPath function retrieves the information from the response.
• If the data read from a JMS DSA is an XML string, the GetByXPath function retrieves the data from the XML.

More information about XPath Expression is available on the W3C website and the W3Schools website:
• Go to [http://www.w3.org](http://www.w3.org) in the standards section, select XML Technology, then select Transformation. Under Current status select XPath, and then a topic that is titled *XML Path Language (XPath) Version 1.0.*
• Go to [http://www.w3schools.com](http://www.w3schools.com) and search for XPath tutorial and select XPath Syntax.

Syntax

The GetByXPath function has the following syntax:
result=GetByXPath(inputString,namesspacemapping,xPathExpression);
Parameters

The GetByXPath function has the following parameters.

**Table 44. GetByXPath function parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input String</strong></td>
<td>String</td>
<td>The input XML string from which the data must be retrieved.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The input string can be a URL. The URL is identified as a string that starts with &quot;file&quot; or &quot;http&quot;. If the URL is specified, the XML document is retrieved from the URL. The content is then used to extract the data. If authentication is required for the HTTP method, use the GetHTTP method to retrieve the XML string output, and use the output in the GetByXPath function.</td>
</tr>
</tbody>
</table>
| **Namespace Mapping** | IPL Object | XPath expressions can contain namespace prefixes. This IPL object provides a way to specify the mapping between the prefix to the real namespace that corresponds to the prefix. The syntax for Namespace Mapping is objectName.<prefixName>=<Namespace URI>  
  • <prefixName> is the prefix to be specified in the XPathExpression for an XML node.  
  • <Namespace URI> is the corresponding URI in the XML document.  
  For example, nsMapping.xsi="http://www.w3.org/2001/XMLSchema-instance";  
  • xsi is the prefix that is used in the XPath Expression.  
  • URI is the URI corresponding to the prefix in the XML document. |
| **XPathExpression** | String | The XPath expression that is applied against the input XML string and the data is retrieved. |

Return value

Define the return value in your XML policy:

```xml
xmlResult=GetByXPath(xmlStr,nsMapping,xPathExpr)
xmlResult.Result.<NodeName>="<ArrayOfResultValues>
```

To access the result, use the following command:

```xml
xmlResult.Result.<NodeName>[<IndexValue>]
```

The following example shows a simple policy that contains the GetByXPath function:

```xml
xPathExpr = "//tns:State/text() //tns:City/text() //tns:Temperature/text()";
xmlResult=GetByXPath(""+WSInvokeDLResult, nsMapping, xPathExpr);
```

To return the City and State values for this policy:

```xml
State=xmlResult.Result.State[0];
City=xmlResult.Result.City[0];
```

[0] is used for the index because the policy contains only one value for City.
If the return value is a string, the type is String. If the return value is a number, the type is Double. If it is a Boolean, the type is Boolean.

If a function is specified, only one function per execution is supported.

The functions can be specified as parameter inside the XPath expressions as well.

**Example 1**

Namespace mapping where no namespaces are defined in the *input string*.

```java
nsMapping = NewObject();
```

**Example 2**

The following examples are of namespaces that are defined in the *input string*.

Example 2a:

```java
nsMapping = NewObject();
nSMapping.tns="urn:www-collation-com:1.0";
nSMapping.xsi="http://www.w3.org/2001/XMLSchema-instance";
nSMapping.col1="urn:www-collation-com:1.0";
```

Example 2b:

```java
nsMapping= NewObject();
nSMapping.xsi="http://www.w3.org/2001/XMLSchema-instance";
nSMapping.soapenv= "http://schemas.xmlsoap.org/soap/envelope/";
nSMapping.xsd="http://www.w3.org/2001/XMLSchema";
```

**Example 3**

An *input string*:

```xml
<com:getCramerObjectDetailsResponse
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:com="http://interfaces/sessions/ejb/ice/cramer/com"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
  <return>
    <cramerObjectDetails>
      <entry>
        <key xsi:type="xsd:string">AKEY</key>
        <value xsi:type="xsd:string">null</value>
      </entry>
      <entry>
        <key xsi:type="xsd:string">Class ID</key>
        <value xsi:type="xsd:string">4</value>
      </entry>
      <entry>
        <key xsi:type="xsd:string">DEVICE1</key>
        <value xsi:type="xsd:string">null</value>
      </entry>
      <entry>
        <key xsi:type="xsd:string">Model Type</key>
        <value xsi:type="xsd:string">1761493060</value>
      </entry>
      <entry>
        <key xsi:type="xsd:string">Class Name</key>
        <value xsi:type="xsd:string">CNAME</value>
      </entry>
      <entry>
        <key xsi:type="xsd:string">manufacturer</key>
```

Example 4

An XPathExpression inside a policy:

```java
XPathExpr = "//cramerObjectDetails/entry[key="Model Type"]//value/text()";
xmlResult = GetByXPath(xmlStr, nsMapping, xPathExpr);
Log(xmlResult);
```

```java
xPathExpr = "//cramerObjectDetails/entry/key/text() | //cramerObjectDetails/entry/value/text() ";
xmlResult=GetByXPath(xmlStr, nsMapping, xPathExpr);
Log(xmlResult);
```

```java
xPathExpr = "count(//entry)";
xmlResult=GetByXPath(xmlStr,nsMapping,xPathExpr);
Log(xmlResult);
```

Log result 1:
(value={1761493060})

Log result 2:
(key={AKEY, Class ID, DEVICE1, Model Type, Class Name, manufacturer, ResourceId},
value={null, 4, null, 1761493060, CNAME, DEVICE2, 81520})

Log result 3:
(FunctionResult=7.0)

In the log result 1, from the first expression, the text of the value node is retrieved when the text of the key node is Model Type.

In the log result 2, from the second expression, the text value of both key and value nodes is retrieved in XML.

Example 5

This example demonstrates how to use an XPathExpr parameter in a policy

```xml
<GetSportsResultsByZIPResult>
Success>true</Success>
<ResponseText>New results found</ResponseText>
<Team1>Sweden</Team1>
<Team2>Norway</Team2>
<Team3>Iceland</Team3>
<News/>
</GetSportsResultsByZIPResult>
</GetSportsResultsByZIPResponse>
```

If the namespace is specified in the XML, you must define the name space mapping:
nsMapping= NewObject();
nsMapping.tns = "http://www.example.com/SportsWS/";
nsMapping.xsd="http://www.example.com/XMLSchema";
nsMapping.soap="http://www.example.com/soap-envelope";
nsMapping.xsi="http://www.example.com/XMLSchema-instance";
xPathExpr = "//tns:Team1/text()";
xmlResult=GetByXPath(""+WSInvokeDLResult, nsMapping, xPathExpr);
Log(xmlResult);

Note that the name space is specified in both nsMapping.tns and xPathExpr = "//tns:Team1/text()".

Use the following to access the return value:
xmlResult.Result.Team1[0]

The returned value is Sweden

GetClusterName

You use the GetClusterName function inside a policy to identify which cluster is running the policy.

Syntax

GetClusterName();

Parameters

This function does not take any parameters.

Return value

String type, the name of the cluster.

GetDate

The GetDate function returns the date/time as the number of seconds expired since the start of the UNIX epoch.

Syntax

The GetDate function has the following syntax:

\[ \text{Integer} = \text{GetDate()} \]

Return value

Number of seconds since the start of the UNIX epoch.

Example

The following example shows how to return the date/time as the number of seconds in UNIX time.

\[ \text{Seconds} = \text{GetDate()} ; \]
\[ \text{Time} = \text{LocalTime}(\text{Seconds}, "MM/dd/yyyy HH:mm:ss \text{ zz}") ; \]
\[ \text{Log}(\text{Seconds}) ; \]
\[ \text{Log}(\text{Time}) ; \]

This example prints the following to the policy log:
GetFieldValue

Use this function to get the value of static or non-static public fields in a Java class. For non-static fields, use the variable \textit{FieldName} for a Java class or \textit{TargetObject} for a Java object. For a static Java class field, use the variable \textit{ClassName}.

Adding Java archive (JAR) files to the shared library directory

Before you can use this policy function, you must make the Java classes available to Netcool/Impact during run time. To make the Java classes available, complete the following steps:

1. Copy the Java classes to the \$NCHOME/dsalib directory.
2. Restart the Impact Server to load the JAR files.

You must repeat this procedure for each Impact Server. This is necessary because the Java class files in the \$NCHOME/dsalib directory are not replicated between servers.

Syntax

\texttt{GetFieldValue( ClassName, TargetObject, FieldName );}

Parameters

\textit{Table 45. GetFieldValue function parameters}

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{ClassName}</td>
<td>Name of the Java class. For a non-static method call, this parameter would be set to null.</td>
</tr>
<tr>
<td>\textit{TargetObject}</td>
<td>Name of the instantiated Java object. For a static method, this parameter would be set to null.</td>
</tr>
<tr>
<td>\textit{FieldName}</td>
<td>Name of the public field variable in the Java class to retrieve the value for.</td>
</tr>
</tbody>
</table>

Returns

Value of the field.

Example

Get the value of the static field named \textit{out} of the Java class, \textit{java.lang.System}:

\texttt{fieldvalue = GetFieldValue("java.lang.System", null, "out");}

Get the value of the non-static field \textit{firstname} from a hypothetical Java class, \textit{com.ibm.DeveloperAccount}. Since it is a non-static field, we are retrieving the field data from an instantiated object of the class. We assume that the constructor needs only a simple ID number to retrieve the account instance:

Example using IPL.

\texttt{dev_acct = NewJavaObject("com.ibm.DeveloperAccount", \{765224\});}

\texttt{first_name = GetFieldValue(null, dev_acct, "firstname");}
Example using JavaScript.

```javascript
dev_acct = NewJavaObject("com.ibm.DeveloperAccount", [765224]);
first_name = GetFieldValue(null, dev_acct, "firstname");
```

GetGlobalVar

This function retrieves the global value that is saved by previous SetGlobalVar calls.

It should be used with a corresponding SetGlobalVar() call. The call only retrieves its own copy of the global variable.

**Syntax**

```javascript
GetGlobalVar(variablename)
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>variablename</td>
<td>Name of the variable whose value you want to retrieve.</td>
</tr>
</tbody>
</table>

**Example**

You can define in your policy a global flag to indicate if a particular exception happens when you run your policy. The flag is set to false originally. The exception handler in your policy sets the flag to true if an exception occurs. Your main policy can check the flag to decide if the exception happens after an action call such as SendJMSMessage().

```javascript
function GetGlobalVarTest(){
  Log("\nrunTimeFlag in getglobalvartest: " + GetGlobalVar("runTimeFlag"));
}

function SetGlobalVarGetGlobalVarTest(){
  Handle java.lang.NullPointerException {
    Log("\nNull pointer exception: runTimeFlag is " + GetGlobalVar("runTimeFlag"));
  }
  Handle java.lang.Exception {
    Log("\nException thrown: runTimeTest is " + GetGlobalVar("runTimeFlag"));
  }
  Date = 1235414139;
  SetGlobalVar("runTimeFlag", Date);
  SendJMSMessage(com.sun.appserv.naming.S1ASCtxFactory, "jms/ConnectionFactory", "this is to test the exception handler");
  Log("\nrunTimeFlag in saveglobalvar and getglobalvar test: " + GetGlobalVar("runTimeFlag"));
  GetGlobalVarTest();
}
```
GetHTTP

You can use the GetHTTP function to retrieve any HTTP URL or to post content to a web page.

You can use it to:
- Retrieve web pages to get information
- Complete a form on a web page
- Run cgi, servlets, or other server scripts on the web server
- Connect through a proxy server

To enable SSL connections between Netcool/Impact servers and external servers, refer to the Netcool/Impact Administration Guide, within the security chapter go to the 'Enabling SSL connections with external servers' topic.

Syntax

GetHTTP(HTTPHost, HTTPPort, Protocol, Path, ChannelKey, Method, AuthHandlerActionTreeName, FormParameters, FilesToSend, HeadersToSend, HttpProperties)

Parameters

This function has the following parameters:

Table 47. GetHTTP function parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTPHost</td>
<td>The IP address of the host to which the call is being made.</td>
</tr>
<tr>
<td>HTTPPort</td>
<td>The port number of the host to which the call is being made.</td>
</tr>
<tr>
<td>Protocol</td>
<td>The protocol used in the call. Set the Protocol parameter to http or https.</td>
</tr>
<tr>
<td>Path</td>
<td>The remaining part of the URL being called that follows the port number.</td>
</tr>
<tr>
<td>ChannelKey</td>
<td>Can be any text. Setting the ChannelKey to different values for different GetHTTP invocations will mean that the different invocations will use different HTTP connections even if an existing connection is free.</td>
</tr>
<tr>
<td>Method</td>
<td>The method used with the function and the method value is GET by default. You can also set it to POST, PUT, or DELETE.</td>
</tr>
<tr>
<td>AuthHandlerActionTreeName</td>
<td>The name of an authorization handler action tree. This parameter is used for compatibility with earlier versions. The default is DefaultAuthHandler.</td>
</tr>
<tr>
<td>FormParameters</td>
<td>Name-value pairs that are URL encoded and added to the URL being accessed.</td>
</tr>
<tr>
<td>FilesToSend</td>
<td>This parameter is used in the POST method to send files.</td>
</tr>
<tr>
<td>HeaderToSend</td>
<td>Contains HTTP Header information in name-value pairs that needs to be added to the HTTP packet that is sent.</td>
</tr>
</tbody>
</table>
Table 47. GetHTTP function parameters (continued)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HttpProperties</td>
<td>A NewObject that contains name-value pairs. The valid variables are:</td>
</tr>
<tr>
<td></td>
<td>- <strong>ConnectionTimeout</strong>: Sets the timeout until a connection is established. The default value of zero means the timeout is not used.</td>
</tr>
<tr>
<td></td>
<td>- <strong>ResponseTimeout</strong>: Sets the default socket timeout (SO_TIMEOUT) in milliseconds which is the timeout for waiting for data. A timeout value of zero is interpreted as an infinite timeout.</td>
</tr>
<tr>
<td></td>
<td>- <strong>AuthenticationHost</strong>: The host the credentials apply to. The host can be set to null if the credentials are applicable to any host.</td>
</tr>
<tr>
<td></td>
<td>- <strong>AuthenticationPort</strong>: The port the credentials apply to. The port can be set to a negative value if the credentials are applicable to any port.</td>
</tr>
<tr>
<td></td>
<td>- <strong>AuthenticationScheme</strong>: The authentication scheme the credentials apply to. The authentication scheme can be set to null if the credentials are applicable to any authentication scheme.</td>
</tr>
<tr>
<td></td>
<td>- Basic: Basic authentication is the original and most compatible authentication scheme for HTTP. It is also the least secure as it sends the user name and password unencrypted to the server.</td>
</tr>
<tr>
<td></td>
<td>- Digest: Digest authentication was added in the HTTP 1.1 protocol. Digest authentication is more secure than basic authentication as it never transfers the actual password across the network. Instead, digest authentication uses it to encrypt a &quot;nonce&quot; value sent from the server.</td>
</tr>
<tr>
<td></td>
<td>- <strong>AuthenticationRealm</strong>: The realm the credentials apply to. The realm can be set to null if the credentials are applicable to any realm.</td>
</tr>
<tr>
<td></td>
<td>- <strong>UserId</strong>: The user name.</td>
</tr>
<tr>
<td></td>
<td>- <strong>Password</strong>: The password.</td>
</tr>
<tr>
<td></td>
<td>- <strong>Content</strong>: Content is generated by using methods such as creating an xml string or JSON string. You can use this property with the HTTP POST and HTTP PUT methods.</td>
</tr>
<tr>
<td></td>
<td>- <strong>ContentType</strong>: The default ContentType is text/xml. You can use this property with the HTTP POST and HTTP PUT methods.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| HttpProperties (continued) | • UseProxy: Use this property to specify whether you want to connect to a web server using a proxy server. Valid values are: true and false. If you set this property to true, you must specify values for the ProxyHost and ProxyPort properties.  
• ProxyHost: If you set UseProxy to true, use this property to specify the name of the proxy host.  
• ProxyPort: If you set UseProxy to true, use this property to specify the port on the proxy host through which to make the connection.  
• UseProxyAuth: Use this property to specify whether you want to perform authentication with the proxy server. If you set UseProxyAuth to true, use the ProxyUserId and ProxyPassword properties to specify the user name and password to authenticate with the proxy server.  
• ProxyUserId: If you set UseProxy to true, use this property to specify the user name to log on to the proxy server.  
• ProxyPassword: If you set UseProxy to true, use this property to specify the password.  
• DecryptPassword: If the password has been encrypted using the Encrypt() function or the nci_crypt script, set this property to true.  
• ProxyRealm: Specify the realm if the proxy server requires one.  
• ProxyAuthScheme: Specify the authentication scheme if the proxy server requires one. |

### Using the GetHTTP function to log a page

The following example logs an external URL.
```plaintext
HTTPHost="maps.googleapis.com";
HTTPPort=80;
Protocol="http";
Path="/maps/api/geocode/xml?address=07002&sensor=false";
ChannelKey="";
Method="";
AuthHandlerActionTreeName="";
FormParameters=null;
FilesToSend=null;
HeadersToSend=null;
HttpProperties=null;
HttpProperties.ContentType="application/xml";
x=GetHTTP(HTTPHost, HTTPPort, Protocol, Path, ChannelKey, Method, AuthHandlerActionTreeName, FormParameters, FilesToSend, HeadersToSend, HttpProperties);
Log(x);
Log(ThePage); // "ThePage" is useful for diagnosing http errors
```

### Using the GetHTTP function to connect through a proxy server

The following example connects to a server using a proxy host.
```plaintext
function sendRequestToServerWithProxy(Host,Port,Protocol,Username,Password, Path, body) {
    body = ""+ body;
    HTTPHost=Host;
```
HTTPPort=Port;
Log("Connecting to Server: " + Host + " on port: " + Port + " with user: " + Username);
ChannelKey="";
Method="GET";
AuthHandlerActionTreeName="";
Log("Content info: " + body);
HttpProperties=NewObject();
HttpProperties.Content=body;
Headers = NewObject();
Headers['Connection']='close';
HttpProperties.UserId = Username;
HttpProperties.Password = Password;

//optional Password can be encrypted using Encrypt() function or nci_crypt script,
//if so the following property must be added:
HttpProperties.DecryptPassword = true;

// following section enables proxy redirect.
HttpProperties.UseProxy = true;
HttpProperties.ProxyHost = "localhost";
HttpProperties.ProxyPort = 8080;

GetHTTP(HTTPHost, HTTPPort, Protocol, Path, ChannelKey, Method, AuthHandlerActionTreeName, null, null, Headers, HttpProperties);

return ThePage;
}

Path="/full/path/to/server/actios?parameter1=val1&parameter2=val2";
result=sendRequestToServerWithProxy("localhost",16310,"http","username","password",Path,null);
Log("result: " + result);

//make sure to replace all required fields with actual values.

Authenticating with a proxy server

The following example is the same as the previous one, but it includes authenticating with the proxy server.

function sendRequestToServerWithProxy(Host,Port,Protocol,Username,Password, Path, body) {
    body = ""+ body;
    HTTPHost=Host;
    HTTPPort=Port;
    Log("Connecting to Server: " + Host + " on port: " + Port + " with user: " + Username);
    ChannelKey="";
    Method="GET";
    AuthHandlerActionTreeName="";
    Log("Content info: " + body);
    HttpProperties=NewObject();
    HttpProperties.Content=body;
    Headers = NewObject();
    Headers['Connection']='close';
    HttpProperties.UserId = Username;
    HttpProperties.Password = Password;

    //optional Password can be encrypted using Encrypt() function or nci_crypt script,
    //if so the following property must be added:
    HttpProperties.DecryptPassword = true;

    // following section enables proxy redirect
    // and includes authenticating with the proxy server.
    HttpProperties.UseProxy = true;
    HttpProperties.ProxyHost = "localhost";
HttpProperties.ProxyPort = 8080;
HttpProperties.UseProxyAuth = true;
HttpProperties.ProxyUserId = ProxyUsername;
HttpProperties.ProxyPassword = ProxyPassword;

// the following optional line is only required
// if endpoint authentication is required
// or alternatively pass the endpoint username and password via
// the HttpProperties.UserId and Password properties
Headers["Authorization"] = 'Basic ' + Base64.encode(endpointuser+":"+endpointpassword);

GetHTTP(HTTPHost, HTTPPort, Protocol, Path, ChannelKey, Method, AuthHandlerActionTreeName, null, null, Headers, HttpProperties);
return ThePage;
}

Path="/full/path/to/server/actions?parameter1=val1&parameter2=val2";
result=sendRequestToServerWithProxy("localhost",16310,"http","username","password",Path,null);
Log("result: " + result);

// make sure to replace all required fields with actual values.

Authentication

Setting the HttpProperties.AuthenticationScheme is no longer required to authenticate with the GetHTTP function. When you do not specify the following variables, AuthenticationScheme, AuthenticationHost, AuthenticationPort, AuthenticationRealm, the GetHTTP function uses any scheme, any host, any port, and any realm. To do basic authentication, all you need is to assign HttpProperties the name-value pairs for a user name and password.

For example:
HttpProperties = NewObject();
HttpProperties.UserId = "<username>";
HttpProperties.Password = "<password>";

GetHTTP and HTTP POST and HTTP PUT methods

GetHTTP supports sending content through the HTTP POST and HTTP PUT methods. For example, you can use these methods to post content to a product that has a REST (Representational State Transfer) interface to get a response.

HttpProperties has two variables, Content and ContentType, which you can use with the HTTP POST and HTTP PUT methods.

The content is generated by using methods such as creating an xml string or JSON string.

For HTTP POST set the following parameters:
HttpProperties = NewObject();
HttpProperties.Content = "<Content String>";
HttpProperties.ContentType = "<The Content Type that server is expecting>";
HttpProperties.UserId = "<username>";
HttpProperties.Password = "<password>";

An example of ContentType is <application/rdf+xml>. The default ContentType is text/xml for HTTP POST and HTTP PUT.
When you set the content for HttpProperties to send through HTTP POST, the FilesToSend and FormParameters must be set to null when calling GetHTTP.

For HTTP PUT, you can specify Content and ContentType for HttpProperties. The FilesToSend and FormParameters must be set to null when calling GetHTTP.

Output values of a GetHTTP call:
ResultCode -> Status Code associated with HTTP call
HeadersSent -> Http Request Headers
HeadersReceived -> Http Response Headers
ThePage -> Response Body of HTTP call
ErrorReason -> Returns the status text associated with the latest response.

GetHibernatingPolicies

The GetHibernatingPolicies function retrieves data items from the Hibernation data type by performing a search of action key values.

To retrieve data items from the Hibernation data type, you call GetHibernatingPolicies and pass two action key values as input parameters. The function performs a search of action keys for all Hibernation data items and returns data items whose action keys fall between the two specified values.

GetHibernatingPolicies returns an array of retrieved hibernation data items. If you do not specify a return variable, the function assigns the array to the built-in DataItems variable and sets the value of the Num variable to the number of data items in the array.

Syntax

The GetHibernatingPolicies function has the following syntax:

```
[Array =] GetHibernatingPolicies(StartActionKey, EndActionKey, [MaxNum])
```

Parameters

The GetHibernatingPolicies function has the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StartActionKey</td>
<td>String</td>
<td>Starting action key to be used in the lexicographical search.</td>
</tr>
<tr>
<td>EndActionKey</td>
<td>String</td>
<td>Ending action key to be used in the lexicographical search.</td>
</tr>
<tr>
<td>MaxNum</td>
<td>Integer</td>
<td>Maximum number of hibernations to return. Default is 1. Optional.</td>
</tr>
</tbody>
</table>

Return value

Array of matching Hibernation data items. Optional.

Example

The following example shows how to retrieve the first Hibernation data items whose action keys fall between the values ActionKey0001 and ActionKey1000.
// Call GetHibernatingPolicies and pass ActionKey0001 and ActionKey1000 as input parameters

StartActionKey = "AlertKey0001";
EndActionKey = "AlertKey1000";
MaxNum = 1;

MyHibers = GetHibernatingPolicies(StartActionKey, EndActionKey, MaxNum);

// Call ActivateHibernation and pass the Hibernation
// as a input parameter

ActivateHibernation(MyHibers[0]);

---

**GetHostAddress**

You use the GetHostAddress function inside a policy to get the IP address of the system where the Netcool/Impact server is running.

**Syntax**

GetHostAddress();

**Return value**

String type, the IP address of the system.

**Parameters**

This function does not take any parameters.

**Example**

host_address = GetHostAddress();
Log("Host Address " + host_address);

**Output**

Parser log: Host Address 67.63.55.3

---

**GetScheduleMember**

The GetScheduleMember function retrieves schedule members that are associated with a particular time range group and time.

To retrieve a schedule from a time range group, you call GetScheduleMember and pass the name of the time range group and the time in seconds. You can retrieve a specific member by position in the schedule, or retrieve all schedule members.

GetScheduleMember returns an array of retrieved schedule member data items. If you do not specify a return variable, the function assigns the array to the built-in DataItems variable and sets the value of the Num variable to the number of data items in the array.

**Syntax**

The GetScheduleMember function has the following syntax:

```
[Array =] GetScheduleMember(Schedule, [Position], [ReturnAll], Time)
```
Parameters

The GetScheduleMembers function has the following parameters.

Table 49. GetScheduleMember function parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schedule</td>
<td>Data Item</td>
<td>Data item that contains the time range group to query for the schedule member.</td>
</tr>
<tr>
<td>Position</td>
<td>Integer</td>
<td>Position of the schedule member in the schedule. Ignored if ReturnAll is specified. Optional.</td>
</tr>
<tr>
<td>ReturnAll</td>
<td>Boolean</td>
<td>If true, the function returns all matching schedule members. If false, the function returns the member specified by the Position parameter. Optional.</td>
</tr>
<tr>
<td>Time</td>
<td>Integer</td>
<td>The time during which the schedule member is active. Expressed as the number of seconds since the beginning of the UNIX epoch. You can obtain this number by calling GetDate. Note: When using the function-building GUI to enter the values, the Time parameter must be entered in the following format: dow mon dd hh:mm:ss zzz yyyy Where: • dow is the day of the week (Sun, Mon, Tue, Wed, Thu, Fri, Sat). • mon is the month (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec). • dd is the day of the month (01 through 31), as two decimal digits. • hh is the hour of the day (00 through 23), as two decimal digits. • mm is the minute within the hour (00 through 59), as two decimal digits. • ss is the second within the minute (00 through 59, as two decimal digits. • zzz is the time zone (and may reflect daylight saving time). If time zone information is not available, then zzz is empty, that is, it consists of no characters. • yyyy is the year, as four decimal digits. The text box in the GUI will show the current time by default in the correct format, so that string can be changed to the desired time.</td>
</tr>
</tbody>
</table>

Return value

Array of the retrieved schedule member data items. Optional.

Examples

The following example shows how to retrieve the schedule member at position 0 of a time range group, using the current time.
GetScheduleMember

The following example shows how to retrieve all schedule members from time range group, using the current time.

```java
TimeRange = GetByKey("Schedule", "Weekday_Shift_01", 1);
Position = 0;
CurrentTime = GetDate();

Members = GetScheduleMember(TimeRange[0], Position, null, CurrentTime);
```

Call GetScheduleMember and pass the time range group, the member position and the current time.

```java
TimeRange = GetByKey("Schedule", "Weekday_Shift_01", 1);
Position = 0;
CurrentTime = GetDate();

Members = GetScheduleMember(TimeRange[0], Position, null, CurrentTime);
```

GetServerName

You use the GetServerName function inside a policy to identify which server is running the policy.

**Syntax**

```java
GetServerName();
```

**Parameters**

This function does not take any parameters.

**Return value**

String type, the name of the Impact Server.

GetServerVar

You use this function to retrieve the global value that is saved by previous SetServerVar.

SetServerVar() and GetServerVar() functions can be used as a way to cache and share variable values across different policies or threads on the server. You have full control of these variables and are responsible for cleaning up global variables.

**Syntax**

```java
GetServerVar(variablename)
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>variablename</td>
<td>Name of the variable.</td>
</tr>
</tbody>
</table>
Example

For an example of using the GetServerVar function, see "SetServerVar" on page 215.

Hibernate

The Hibernate function causes a policy to hibernate.

To hibernate a policy, you call Hibernate and pass an action key and the number of seconds for the policy to hibernate as input parameters. The action key can be any unique key you want to use to identify the policy.

When a policy hibernates, it stops running and is stored as a data item of type Hibernation. At intervals, the hibernating policy activator queries the Hibernation data type and wakes those policies whose timeout value has expired. You can also wake a hibernating policy from within another policy using the ActivateHibernation function. JavaScript also supports hibernation functions.

Note: If you have a Hibernate call in a policy, and that policy is in a chain, the policy will exit when the Hibernate is reached and the policy chain will not continue.

Syntax

The Hibernate function has the following syntax:

```javascript
Hibernate(ActionKey, [Reason], Timeout)
```

Parameters

The Hibernate function has the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActionKey</td>
<td>String</td>
<td>String that uniquely identifies the hibernating policy.</td>
</tr>
<tr>
<td>Reason</td>
<td>String</td>
<td>String that describes the reason for hibernating the policy. Optional.</td>
</tr>
<tr>
<td>Timeout</td>
<td>Integer</td>
<td>Number of seconds before the hibernating policy is available to be wakened by the hibernating policy activator.</td>
</tr>
</tbody>
</table>

Example

The following example shows how to cause a policy to hibernate for one minute.

```javascript
// Call Hibernate and pass an action key and the number of seconds to hibernate as input parameters
ActionKey = "ActionKey" + GetDate();
Timeout = 60;
Hibernate(ActionKey, null, Timeout);
RemoveHibernation(ActionKey);
```
Illegal

If the input in the policy has malicious content, the Illegal function throws an exception in a policy.

Syntax

The Illegal function has the following syntax:

\[
\text{Illegal(InputString, ArrayOfIllegalChars)}
\]

Parameters

The Illegal function has the following parameters:

*Table 52. Illegal function parameters. Table of parameters for the Illegal function*

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InputString</td>
<td>String</td>
<td>The input String to check for malicious content.</td>
</tr>
<tr>
<td>ArrayOfIllegalChars</td>
<td>String[]</td>
<td>The Array of Illegal Characters or Words to Avoid in the Input String.</td>
</tr>
</tbody>
</table>

Return value

The function throws a PolicyException if illegal words or characters exist in the input string.

Example

The following example throws an exception because it contains words `delete` and `rm`.

```plaintext
MyString = "Test delete Hello-World-rm!";
Log("MyString: "+MyString);
Illegal(MyString, {"delete", "rm"});
```

Int

The Int function converts a float, string, or Boolean expression to an integer.

This function truncates any decimal fraction value associated with the number. For example, the value of Int(1234.67) is 1234.

Syntax

The Int function has the following syntax:

\[
\text{Integer = Int(Expression)}
\]
Parameters

The Int function has the following parameter.

Table 53. Int function parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>Float</td>
<td>String</td>
</tr>
</tbody>
</table>

Return value

The converted integer number.

**Important**: A float value converted to an int value is truncated not rounded.

Example

The following example shows how to convert float, string, and boolean expressions to an integer.

```
MyInt = Int(123.45);
Log(MyInt);

MyInt = Int(123.54);
Log(MyInt);

MyInt = Int("456");
Log(MyInt);

MyInt = Int(false);
Log(MyInt);
```

This example prints the following message to the policy log:

```
Parser Log: 123
Parser Log: 123
Parser Log: 456
Parser Log: 0
```

JavaCall

You use this function to call the method MethodName in the Java object TargetObject with parameters, or, to call the static method MethodName in the Java class ClassName with parameters.

**Adding Java archive (JAR) files to the shared library directory**

Before you can use this policy function, you must make the Java classes available to Netcool/Impact during run time. To make the Java classes available, complete the following steps:

1. Copy the Java classes to the $NCHOME/dsalib directory.
2. Restart the Impact Server to load the JAR files.

You must repeat this procedure for each Impact Server. This is necessary because the Java class files in the $NCHOME/dsalib directory are not replicated between servers.

Syntax

```
JavaCall( ClassName, TargetObject, MethodName, Parameters )
```
Parameters

Table 54. JavaCall function parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClassName</td>
<td>Name of the Java class. When you are using a non-static method call, this parameter is set to null.</td>
</tr>
<tr>
<td>TargetObject</td>
<td>Name of the instantiated Java object. When you are using a static method, this parameter is set to null.</td>
</tr>
<tr>
<td>MethodName</td>
<td>Name of the Java method in the Java class you are calling.</td>
</tr>
<tr>
<td>Parameters</td>
<td>An array of parameter values the method requires.</td>
</tr>
</tbody>
</table>

Returns

Value that the method returns, if any.

Examples

Call a method println() from the Java System object represented by variable out to print a line of text message to system stdout:

Get a system property named app. Call the java.lang.System.getProperty(String key) method with the following line:

IPL example:

```
propValue = JavaCall("java.lang.System", null, "getProperty", { "app" });
```

The same example in JavaScript:

```
propValue = JavaCall("java.lang.System", null, "getProperty", [ "app" ]);   
```

In this example, use this function to check JVM properties by calling methods on class java.lang.System from your policy. This IPL example, prints the value of a JRE system property named app:

```
propvalue = JavaCall("java.lang.System", null, "getProperty", { "app" });
log("Property "app" is " + propvalue);
```

The same example in JavaScript:

```
propvalue = JavaCall("java.lang.System", null, "getProperty", [ "app" ]);    
Log("Property "app" is " + propvalue);
```

In the following IPL example, create a Java object of class Vector and call its methods:

```
// Create an new instance of Java class java.util.Vector
vector = NewJavaObject("java.util.Vector", null);
//Add "111111" to vector.
JavaCall(null, vector, "add", { "111111" });
// Retrieve element at position 0.
log("The first element is " + JavaCall(null, vector, "get", { 0 }));
// Add element "222222" to position 0
JavaCall(null, vector, "add", { 0, "222222" });
// Print out the element at position 0.
// It should now be "222222", not "111111".
log("The first element is " + JavaCall(null, vector, "get", { 0 }));
// Add element "333333" to vector.
```

Chapter 13. Working with functions 169
JavaCall(null, vector, "add", ["33333"]); // Print out the current size of vector. The value should be 3.
log("Vector size is " + JavaCall(null, vector, "size", {}));

If you are using JavaScript, for a JavaCall that needs an integer argument you must use the `Integer.parseInt` JavaCall to create an actual integer.

// Create a new instance of Java class `java.util.Vector`
vector = NewJavaObject("java.util.Vector", null);
index = JavaCall("java.lang.Integer", null, "parseInt", ["0"]);
// Add "111111" to vector.
JavaCall(null, vector, "add", ["111111"]);
// Retrieve element at position 0.
Log("The first element is " + JavaCall(null, vector, "get", [index])
// Add element "22222" to position 0
JavaCall(null, vector, "add", [index, "22222"]);
Log("The first element is " + JavaCall(null, vector, "get", [index])

---

**JRExecAction**

The JRExecAction function runs an external command by using the JRExec server.

To run an external command, call `JRExecAction` and pass the name of the command, an array of strings that contain any command-line arguments and a timeout value.

If you are using the JRExecAction and Command Response functions, you are encouraged to sanitize strings before you use their values in these functions. You can use the functions `Escape` and `Illegal` to sanitize string input.

**Syntax**

The JRExecAction function has the following syntax:

\[ \text{JRExecAction(Command, Parameters, ExecuteOnQueue, TimeOut)} \]

**Parameters**

The JRExecAction function has the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command</td>
<td>String</td>
<td>Name of the external command, script, or program to run.</td>
</tr>
<tr>
<td>Parameters</td>
<td>Array</td>
<td>Array of parameters to pass to the command.</td>
</tr>
<tr>
<td>ExecuteOnQueue</td>
<td>Boolean</td>
<td>To place the command on the JRExec server queue, set this parameter to true. Commands on the queue are executed by the JRExec server in parallel mode. To run the command in parallel with any other commands that might currently be running, set this parameter to true. Set this parameter to false for most uses of this function.</td>
</tr>
<tr>
<td>TimeOut</td>
<td>Integer</td>
<td>Number of seconds to wait after sending the command before timing out.</td>
</tr>
</tbody>
</table>

These parameters are only available after you execute the function:
Table 56. JRExecAction function output parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExecOutput</td>
<td>String</td>
<td>Return value of the script or command.</td>
</tr>
<tr>
<td>ExitCode</td>
<td>Integer</td>
<td>Exit code of the script or command.</td>
</tr>
</tbody>
</table>

Examples

The following example shows how to run an external command named `myscript` using the JRExec server using IPL.

```java
// Call JRExecAction and pass the name of the command
// the input parameters and a timeout value
Command = "/usr/local/bin/myscript";
Parameters = {"param1", "param2", "param3"};
TimeOut = 5;
MyResult = JRExecAction(Command, Parameters, false, TimeOut);
// Output Values:
Log("ExecOutput: " + ExecOutput);
Log("ExitCode: " + ExitCode);
```

The following example shows how to run an external command named `myscript` using the JRExec server using JavaScript.

```javascript
// Call JRExecAction and pass the name of the command
// the input parameters and a timeout value
Command = "/usr/local/bin/myscript";
Parameters = ["param1", "param2", "param3"];
TimeOut = 5;
MyResult = JRExecAction(Command, Parameters, false, TimeOut);
// Output Values:
Log("ExecOutput: " + ExecOutput);
Log("ExitCode: " + ExitCode);
```

Keys

The `Keys` function returns an array of strings that contain the field names of the specified data item.

Syntax

The `Keys` function has the following syntax:

```
Array = Keys(DataItem)
```

Parameters

The `Keys` function has the following parameter.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataItem</td>
<td>Data item</td>
<td>The data item whose keys you want to return.</td>
</tr>
</tbody>
</table>

Return value

Array of strings that contain the field names of the data item.
Example

The following example shows how to return an array of strings that contain the field names.

```plaintext
MyNodes = GetByFilter("Node", "0=0", false);
MyNode = MyNodes[0];
Fields = Keys(MyNode);
Log(Fields);
```

This example prints the names of all the fields in the Node data item.

Length

The Length function returns the number of elements or fields in an array or the number of characters in a string.

Syntax

The Length function has the following syntax:

```plaintext
Integer = Length(Array | String)
```

Parameters

The Length function has the following parameter.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>Array or String</td>
<td>Array whose elements to count, or string whose characters to count.</td>
</tr>
</tbody>
</table>

Return value

Number of elements or characters.

Example

The following example shows how to return the number of elements in an array.

```plaintext
MyNodes = GetByFilter("Node", "Location = 'New York'", false);
NumNodes = Length(MyNodes);
Log(NumNodes);
```

This example prints the number of data items in the MyNodes array to the policy log.

Load

You use this function to load a JavaScript library into your JavaScript policy.

After you load a JavaScript library you can call its defined functions in your JavaScript policy.

The Load function has the following usage:

```plaintext
Load(libraryname)
```
where libraryname is the name of an Impact JavaScript policy, or a filename of an external JavaScript library, without the .js extension. To be able to load a library, you must first copy it over to the $NCHOME/jslib directory. After you load the library, you can call its functions by referencing their names.

Assume, for example, that your MyLibrary.js JavaScript policy has the following function defined:

```javascript
function myfunc() {
    Log("Running myfunc");
}
```

You can load the MyLibrary policy into another JavaScript policy, and call its myfunc function using the following code:

```javascript
Load("MyLibrary");
myfunc();
```

### LocalTime

The LocalTime function returns the number of seconds since the beginning of the UNIX epoch as a formatted date/time string.

#### Syntax

The LocalTime function has the following syntax:

```javascript
Date = LocalTime(Seconds, Pattern)
```

#### Parameters

The LocalTime function has the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seconds</td>
<td>Integer</td>
<td>Number of seconds.</td>
</tr>
<tr>
<td>Pattern</td>
<td>String</td>
<td>Date/time pattern. Optional. If not specified, the default date/time pattern is used.</td>
</tr>
</tbody>
</table>

#### Return value

A formatted date/time string.

#### Example

The following example shows how to return the given number of seconds in various formats.

```javascript
// Return date/time string using default format

Seconds = GetDate();
Time = LocalTime(Seconds);
Log(Time);

// Return date/time strings using specified formats

Seconds = GetDate();
Time = LocalTime(Seconds, "MM/dd/yy");
Log(Time);
```
Seconds = GetDate();
Time = LocalTime(Seconds, "HH:mm:ss");
Log(Time);

This example prints the following message to the policy log:
Parser Log: Nov 11 2003, 15:44:38 EST
Parser Log: 06/19/03
Parser Log: 13:11:24

---

Log

The Log function prints a message to the policy log.

To print a message to the policy log, you call this function and pass the expression that you want to print and, optionally, a log level.

The log level specifies the level of severity for the message, with 1 being the lowest and 3 being highest. The policy logger service configuration specifies the level of severity that the message must meet to be printed in the log. For example, if you configure the policy logger with a severity level of 2, only messages with a log level of 2 or less are printed. Messages with a log level of 0 are always logged.

Syntax

The Log function has the following syntax:
`Log([LogLevel], Expression)`

Parameters

The Log function has the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LogLevel</td>
<td>Integer</td>
<td>An Integer between 0 and 3 that specifies the level of severity for the message. Default is 0. Optional.</td>
</tr>
<tr>
<td>Expression</td>
<td>Integer</td>
<td>Message to print to the log.</td>
</tr>
<tr>
<td>Expression</td>
<td>Float</td>
<td></td>
</tr>
<tr>
<td>Expression</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>Expression</td>
<td>Boolean</td>
<td></td>
</tr>
</tbody>
</table>

Examples in IPL

The following example shows how to print a message to the policy log in IPL.
Log("This is a test.");

This example prints the following message to the policy log:
Parser Log: This is a test.

The following example shows how to print a message to the policy log with a severity level of 2 in IPL.
Log(2, "This is another test.");

This example prints the message to the policy log only if the policy logger service is configured with a log level of 2 or greater.
Examples in IPL and JavaScript

This is the format of a logged string in IPL:

```plaintext
MyContext = NewObject();
MyContext.DEPT = 'Dept A';
MyContext.LOCID = 5;
log("Policy Context (IPL):" + MyContext);
```

This example prints the following message to the policy log:

```
Policy Context (IPL):"Created by parser"=(LOCID=5, DEPT=Dept A)
```

This is the format of a logged string in JavaScript:

```plaintext
MyContext = NewObject();
MyContext.DEPT = 'Dept A';
MyContext.LOCID = 5;
Log("Policy Context (JS):" + MyContext);
```

This example prints the following message to the policy log:

```
Policy Context (JS):{DEPT:"Dept A", LOCID:5}
```

Merge

The Merge function merges two contexts or event containers by adding the member variables of the source context or event container to those of the target.

Syntax

The Merge function has the following syntax:

```
[Target] = Merge(Target, Source, [Exclude])
```

Parameters

The Merge function has the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target</td>
<td>Context</td>
<td>Target context or event container.</td>
</tr>
<tr>
<td>Source</td>
<td>Context</td>
<td>Source context or event container.</td>
</tr>
<tr>
<td>Exclude</td>
<td>Array</td>
<td>Array of strings that contain the names of member variables to exclude from the merge. Optional.</td>
</tr>
</tbody>
</table>

Return value

The merged contexts or event containers. Optional.

Examples

The following example shows how to merge two contexts.

```plaintext
MyContext1 = NewObject();
MyContext1.s1 = "square";
MyContext1.s2 = "triangle";
```
MyContext2 = NewObject();
MyContext2.s3 = "circle";
MyContext2.s4 = "rectangle";

Merge(MyContext1, MyContext2, null);
Log(MyContext1);

This policy prints the following message to the policy log:
Parser log: "Created by parser"=(s2=triangle,s1=square,s3=circle,s4=rectangle)

The following example shows how to merge two contexts and exclude some member variables from the merge using IPL.

MyContext1 = NewObject();
MyContext1.s1 = "square";
MyContext1.s2 = "triangle";

MyContext2 = NewObject();
MyContext2.s3 = "circle";
MyContext2.s4 = "rectangle";

Merge(MyContext1, MyContext2, ["s3"]);
Log(MyContext1);

This example prints the following message to the policy log:
Parser log: "Created by parser"=(s2=triangle,s1=square,s4=rectangle)

The following example shows how to merge two contexts and exclude some member variables from the merge using JavaScript.

MyContext1 = NewObject();
MyContext1.s1 = "square";
MyContext1.s2 = "triangle";

MyContext2 = NewObject();
MyContext2.s3 = "circle";
MyContext2.s4 = "rectangle";

Merge(MyContext1, MyContext2, ["s3"]);
Log(MyContext1);

This example prints the following message to the policy log:
Parser log: {s1:"square", s2:"triangle", s4:"rectangle"}

---

**NewEvent**

The NewEvent function creates a new event container.

**Syntax**

The NewEvent function has the following syntax:

```
EventContainer = NewEvent(EventReader)
```
Parameters

The NewEvent function has the following parameter.

Table 62. NewEvent function parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EventReader</td>
<td>String</td>
<td>Name of the event reader associated with the event source.</td>
</tr>
</tbody>
</table>

Return value

Event container that stores the new event.

Example

The following example shows how to create a new event container, and how to populate its event field variables and the EventReaderName variable.

```java
MyEvent = NewEvent("OMNIbusEventReader");
// Set the EventReaderName member variable, only required if the policy // is run by a means other than an event reader service. This specifies the // name of the event reader to use to send the event
MyEvent.EventReaderName = "OMNIbusEventReader";
// Set the event field variables
MyEvent.Identifier = "XYZ123";
MyEvent.Node = "DB_SERVER_01";
MyEvent.Class = "99999";
MyEvent.Manager = "Netcool/Impact";
MyEvent.Acknowledged = 0;
MyEvent.Severity = 5;
MyEvent.Type = 0;
```

NewJavaObject

The NewJavaObject function is used to call the constructor for a Java class.

Adding Java archive (JAR) files to the shared library directory

Before you can use this policy function, you must make the Java classes available to Netcool/Impact during run time. To make the Java classes available, complete the following steps:

1. Copy the Java classes to the $NCHOME/dsalib directory.
2. Restart the Impact Server to load the JAR files.

You must repeat this procedure for each Impact Server. This is necessary because the Java class files in the $NCHOME/dsalib directory are not replicated between servers.

Syntax

NewJavaObject( ClassName, Parameters )
**Parameters**

**Table 63. NewJavaObject function parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClassName</td>
<td>Name of the Java class you are instantiating a Java object for.</td>
</tr>
<tr>
<td>Parameters</td>
<td>An array of parameter values a constructor for this class requires.</td>
</tr>
</tbody>
</table>

**Returns**

The instantiated object.

**Examples**

To create a Java String Object "This is a string!" and assign it to variable `str`, then in an IPL policy, put in the following line:

```java
str = NewJavaObject("String", {"This is a string!"});
```

This IPL example of code creates a Java object of class `java.util.Hashtable` and then adds, retrieves, and removes data from it:

```java
// Create a new instance of Java HashTable class.
my_hash = NewJavaObject("java.util.Hashtable", null);
// Add table entry {"one", "aaaa"} to my_hash.
JavaCall(null, my_hash, "put", {"one", "aaaa"});
// Add entry ("two", "bbbb") to table my_hash.
JavaCall(null, my_hash, "put", {"two", "bbbb"});
// Add entry ("three", "ccccc") to table.
JavaCall(null, my_hash, "put", {"three", "ccccc"});
// Print the table entry value indexed by the key "three"
log("Check hashtable value indexed by key "three". Value is " +
JavaCall(null, my_hash, "get", {"three"});)
// Remove the entry indexed by the key "one" from the table
JavaCall(null, my_hash, "remove", {"one"});
log("After remove call, my_hash becomes " + my_hash);
```

To create a Java String Object "This is a string!" and assign it to variable `str`, then in a JavaScript policy, put in the following line:

```javascript
str = NewJavaObject("String", ["This is a string!"]);
```

This JavaScript example of code creates a Java object of class `java.util.Hashtable` and then adds, retrieves, and removes data from it:

```javascript
// Create a new instance of Java HashTable class.
my_hash = NewJavaObject("java.util.Hashtable", null);
// Add table entry ("one", "aaaa") to my_hash.
JavaCall(null, my_hash, "put", {"one", "aaaa"});
// Add entry ("two", "bbbb") to table my_hash.
JavaCall(null, my_hash, "put", {"two", "bbbb"});
// Add entry ("three", "ccccc") to table.
JavaCall(null, my_hash, "put", {"three", "ccccc"});
// Print the table entry value indexed by the key "three"
log("Check hashtable value indexed by key "three". Value is " +
JavaCall(null, my_hash, "get", ["three"]);
// Remove the entry indexed by the key "one" from the table
JavaCall(null, my_hash, "remove", ["one"]);
log("After remove call, my_hash becomes " + my_hash);
```

The Impact policy does not support file or directory operations. The Java API, however, supports these operations in its `java.io.*` library. You can access this...
library and all other functions the Java API provides by using the Java Policy functions. This piece of code, for example, calls java.io.File class, opens a directory, and outputs a list of the files in the directory:

This example applies to IPL.
```
homeDir = NewJavaObject("java.io.File", {"/home/user/"});
file_list = JavaCall(null, homedir, "list", {});
Log("file_list is " + file_list);
```

This example applies to JavaScript.
```
homeDir = NewJavaObject("java.io.File", ["/home/user/"]);
file_list = JavaCall(null, homedir, "list", []);
Log("file_list is " + file_list);
```

### NewObject

The NewObject function creates a new context.

**Syntax**

The NewObject function has the following syntax:
```
Context = NewObject()
```

**Return value**

The new context.

**Example**

The following example shows how to create a new context.
```
MyContext = NewObject();
```

### ParseDate

The ParseDate function converts a formatted date/time string to the time in seconds since the beginning of the UNIX epoch. 1st January 1970 00:00:00 (UTC).

**Syntax**

The ParseDate function has the following syntax:
```
Integer = ParseDate(Date, [Pattern])
```

**Parameters**

The ParseDate function has the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td>String</td>
<td>Formatted date/type string.</td>
</tr>
<tr>
<td>Pattern</td>
<td>String</td>
<td>String that contains the formatting pattern. Optional. If not specified, default format is used.</td>
</tr>
<tr>
<td>Timezone</td>
<td>String</td>
<td>An optional parameter format that is based on the java.util.TimeZone class. You can use the parameter to provide the timezone that the date and time should be converted to.</td>
</tr>
</tbody>
</table>
Return value

The time in seconds.

Examples

The following example shows how to convert various formatted date/time strings to the time in seconds.

// Convert date/time string using default format

DateString = "Nov 11 2003, 15:44:38 EST";
Time = ParseDate(DateString);
Log(Time);

// Convert date/time strings using specified formats

DateString = "06/19/03";
Time = ParseDate(DateString, "MM/dd/yy");
Log(Time);

DateString = "13:11:24";
Time = ParseDate(DateString, "HH:mm:ss");
Log(Time);

This example prints information similar to the following message to the policy log:


The following example shows how to pass the optional parameter Timezone to the ParseDate function.

Format

Integer = ParseDate(Date, [Pattern], [Timezone])

Example

DateString = "2014-04-01 13:11:24";
Time = ParseDate(DateString, "yyyy-MM-dd HH:mm:ss", "Australia/Sydney");
Log(Time);

Output:

1396318284

ParseJSON

This function converts a JSON string into a Netcool/Impact Object.

Syntax

The ParseJSON function has the following syntax:

result = ParseJSON(JSONString);
Parameters

The ParseJSON function has the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSON String</td>
<td>String</td>
<td>A formatted JSON String.</td>
</tr>
</tbody>
</table>

Return value

The return of the function can be accessed as an Object.property format.

\[\text{items[index].fieldName}\]
\[\text{items[index].propertyArray[index].property}\]

The return from the function is an Impact Object representing the JSON string.

1) JSON Object is returned as Impact Object
2) JSON Array is returned as Array Of Impact Objects
3) Array returned as Java Array.

Example

```plaintext
content="{"+
  \"glossary\": {" +
  \"title\": \"example glossary\"," +
  \"GlossDiv\": {" +
  \"title\": \"S\"," +
  \"GlossList\": {" +
  \"GlossEntry\": {" +
  \"ID\": \"SGML\"," +
  \"SortAs\": \"SGML\"," +
  \"GlossTerm\": \"Standard Generalized Markup Language\"," +
  \"Acronym\": \"SGML\"," +
  \"Abbrev\": \"ISO 8879:1986\"," +
  \"GlossDef\": {" +
  \"para\": \"A meta-markup language, used to create markup languages\" +
  * such as DocBook.\"\"," +
  \"GlossSeeAlso\": [{\"GML\", \"XML\",1}]+
  "]}+" +
  \"GlossSee\": \"markup\" +
  " +
  ]}+" +
  " +
  "}+" +

jsonObj = ParseJSON(content);
Log("jsonObj: "+ jsonObj);
GlossSeeAlsoArray=jsonObj.glossary.GlossDiv.GlossList;
GlossSeeAlsoArray=GlossSeeAlsoArray.GlossEntry.GlossDef.GlossSeeAlso;
Log(" GlossSeeAlsoArray: "+ GlossSeeAlsoArray);
LenG=Length(GlossSeeAlsoArray);
indexG=0;
while(indexG < LenG) {
  Log("GlossSeeAlsoArray["+indexG+"] = " + GlossSeeAlsoArray[indexG]);
  indexG = indexG+1;
}
```

The policy output is similar to the following example:

October 15, 2014 9:12:31 AM EDT [PolicyLogger] [TestExample] [pool-3-thread-478]
Parser log: jsonObj: "Context"="glossary="Context"="title=example glossary," +
GlossDiv="Context"=
(GlossList="Context"="GlossEntry="Context"="SortAs=SGML,
GlossDef="Context"="GlossSeeAlso="GML, XML, 1),
para=A meta-markup language, used to create markup languages such as DocBook.),
An example of a JSON string that is an array and starts with a square bracket is converted to an Impact object.

```
JSONArray='[
    {
    "color": "red", "value": "#f00"
    },
    {
    "color": "green", "value": "#0f0"
    },
    {
    "color": "blue", "value": "#00f"
    },
    {
    "color": "cyan", "value": "#0ff"
    },
    {
    "color": "magenta", "value": "#f0f"
    },
    {
    "color": "yellow", "value": "#ff0"
    },
    {
    "color": "black", "value": "#000"
    }
];
parsed=ParseJSON(JSONArray);
Log(parsed.arrayItems[0].color);
```

Policy output.
Parser log: red

## Random

The Random function returns a random integer between zero and the given upper bound.

### Syntax

The Random function has the following syntax:

```plaintext
Integer = Random(UpperBound)
```

### Parameters

The Random function has the following parameter.

*Table 66. Random function parameters*

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UpperBound</td>
<td>Integer</td>
<td>Highest possible integer to be returned.</td>
</tr>
</tbody>
</table>

### Return value

Random integer.

### Example

The following example shows how to return a random integer between 1 and 10.

```plaintext
UpperBound = 9;
MyRandom = Random(UpperBound) + 1;
Log(MyRandom);
```

This example prints the following message to the policy log:
Parser Log: 6
RDF functions

You can use RDF functions to make Netcool/Impact compatible with open services for lifecycle collaboration (OSLC).

**RDFModel**

You can use the RDFModel function to create an RDF model without any input parameters.

To create an empty RDF model, you call the RDFModel function without entering any input parameters. The function returns an empty RDF model.

**Syntax**

The RDFModel function has the following syntax:

[Model =] RDFModel()

**Parameters**

The RDFModel function has no input parameters.

**RDFModelToString**

You can use the RDFModelToString function to export an RDF model to a string in a particular language.

When you create or write an RDF model, you can use the RDFModelToString function to export a model to a string in a particular language. You can define a model object and a string that contains the language that is used as input parameters. If the language string is null or an empty string, the default language RDF/XML is used. The following language strings are supported:

- RDF/XML
- RDF/XML-ABBREV
- TURTLE
- TTL
- N3

RDFModelToString returns a string.

**Syntax**

The RDFModelToString function has the following syntax:

[String =] RDFModelToString(Model, Language)

**Parameters**

The RDFModelToString function has the following parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>Model</td>
<td>Model object to output</td>
</tr>
<tr>
<td>Language</td>
<td>String</td>
<td>Language type</td>
</tr>
</tbody>
</table>
The following example updates the namespaces in a model:

```java
//Retrieve the RDF from OSLC provider through the GetHTTP method
HTTPHost="example.com";
HTTPPort=9081;
Protocol="https";
Path="/NCICLUSTER_NCI_oslc/data/resourceShapes/alerts";
ChannelKey="tom";
Method="";
AuthHandlerActionTreeName="";
FormParameters=NewObject();
FilesToSend=NewObject();
HeadersToSend=NewObject();
HttpProperties = NewObject();
HttpProperties.UserID="impactadmin";
HttpProperties.Password="passw0rd";
x=GetHTTP(HTTPHost, HTTPPort, Protocol, Path, ChannelKey, Method,
AuthHandlerActionTreeName,FormParameters, FilesToSend, HeadersToSend,
HttpProperties);
//Create Model from RDF payload
drf=RDFParse(x);

//Retrieve all statements from model
allStatements=RDFSelect(rdf,null,null,null);

//Output RDF to log using N3
log(RDFModelToString(rdf, "N3"));

//Output RDF to log using the default language (RDF/XML)
log(RDFModelToString(rdf, null));
```

### RDFModelUpdateNS

You can use the RDFModelUpdateNS function to insert, update, or remove a namespace from an RDF model.

When you create an RDF model, you can use the RDFModelUpdateNS function to insert, update, or remove a namespace from the model. You can define a model object, prefix string, and a URI string as input parameters. If the URI is null or an empty string, the function removes the prefix string from the model. If the URI contains a string with a non-empty value and the prefix exists, the URI is updated. If the prefix does not exist, a new prefix and URI is added to the model. RDFModelUpdateNS returns this model.

### Syntax

The RDFModelUpdateNS function has the following syntax:

```
[Model =] RDFModelUpdateNS (Model, Prefix, URI)
```

### Parameters

The RDFModelUpdateNS function has the following parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>Model</td>
<td>Model object to update</td>
</tr>
<tr>
<td>Prefix</td>
<td>String</td>
<td>Contains the prefix to be updated in the model</td>
</tr>
<tr>
<td>URI</td>
<td>String</td>
<td>Contains the URI to associate with prefix</td>
</tr>
</tbody>
</table>
The following example updates the namespaces in a model:

```java
//Create model
data model = RDFModel();

//Update or insert namespace to model
RDFModelUpdateNS(model, "oslc", "http://open-services.net/ns/core#");
RDFModelUpdateNS(model, "rdfs", "http://www.w3.org/2000/01/rdf-schema#");
RDFModelUpdateNS(model, "dcterms", "http://purl.org/dc/terms/");
```

The following piece of code deletes an existing model's namespace:

```java
//Retrieve the RDF from OSLC provider through the GetHTTP method
HTTPHost="example.com";
HTTPPort=9081;
Protocol="https";
Path="/NCICLUSTER_NCI_oslc/data/resourceShapes/alerts";
ChannelKey="tom";
Method="";
AuthHandlerActionTreeName="";
FormParameters=NewObject();
FilesToSend=NewObject();
HeadersToSend=NewObject();
HttpProperties = NewObject();
HttpProperties.UserId="impactadmin";
HttpProperties.Password="passw0rd";
x=GetHTTP(HTTPHost, HTTPPort, Protocol, Path, ChannelKey, Method, AuthHandlerActionTreeName, FormParameters, FilesToSend, HeadersToSend, HttpProperties);

//Create Model from RDF payload
model=RDFParse(x);

//Delete namespace from model that has prefix 'oslc'
RDFModelUpdateNS(model, "oslc", null);
```

**RDFNodeIsResource**

You can use the RDFNodeIsResource function to help other functions read and parse objects that are also an RDF resource. You can define an RDF node as a input parameter in this function. If the object is an RDF resource, the function returns a true value. If the object is an RDF literal, the function returns a false value. Other functions can use the model returned by the RDFNodeIsResource function to continue reading and parsing the RDF object.

**Syntax**

The RDFNodeIsResource function has the following syntax:

```java
[Boolean =] RDFNodeIsResource (Object)
```

**Parameters**

The RDFNodeIsResource function has the following parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>RDF node</td>
<td>RDF object type check</td>
</tr>
</tbody>
</table>

The following example shows statements based on an RDF that is retrieved by the GetHTTP function:
//Retrieve the RDF from OSLC provider through the GetHTTP method
HTTPHost="example.com";
HTTPPort=9081;
Protocol="https";
Path="/NCICLUSTER_NCI_oslc/data/resourceShapes/alerts";
ChannelKey="tom";
Method="";
AuthHandlerActionTreeName="";
FormParameters=NewObject();
FilesToSend=NewObject();
HeadersToSend=NewObject();
HttpProperties = NewObject();
HttpProperties.UserId="impactadmin";
HttpProperties.Password="passw0rd";
x=GetHTTP(HTTPHost, HTTPPort, Protocol, Path, ChannelKey, Method,
AuthHandlerActionTreeName, FormParameters, FilesToSend, HeadersToSend,
HttpProperties);

//Create Model from RDF payload
rdf=RDFParse(x);

//Retrieve all statements from model
allStatements=RDFSelect(rdf,null,null,null);

//Output subject, predicate, and objects from all statements returned, whose
//object is a literal, to the log
Size=Length(allStatements);
log(Size);
Count=0;
While (Count < Size) {
    if (!RDFNodeIsResource (allStatements [Count].object)) {
        log (allStatements [Count].subject + " " + allStatements [Count].predicate + " "+
+ allStatements [Count].object + ");
    }
    Count = Count + 1;
}

**RDFNodeIsAnon**

You can use the RDFNodeIsAnon function to assist in reading and parsing an RDF. The RDFNodeIsAnon takes in a subject/object containing an RDFNode as an input parameter and returns true if the resource is anonymous. If the return value is false, the RDF resource is not anonymous. The model generated by the function can then be used by other functions to continue reading and parsing the RDF.

RDFNodeIsAnon returns true or false, depending if the RDFNode is anonymous

**Syntax**

The RDFNodeIsAnon function has the following syntax:

[Boolean =] RDFNodeIsAnon (Node)

**Parameters**

The RDFNodeIsAnon function has the following parameter:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node</td>
<td>RDFNode</td>
<td>Subject or object to check for</td>
</tr>
</tbody>
</table>
//Retrieve the RDF from OSLC provider through the GetHTTP method
HTTPHost="example.com";
HTTPPort=9081;
Protocol="http";
Path="/NCICLUSTER_NCI_oslc/data/resourceShapes/alerts";
ChannelKey="tom";
Method="";
AuthHandlerActionTreeName="";
FormParameters=newobject();
FilesToSend=newobject();
HeadersToSend=newobject();
HttpProperties = NewObject();
HttpProperties.UserId="impactadmin";
HttpProperties.Password="passw0rd";
x=GetHTTP(HTTPHost, HTTPPort, Protocol, Path, ChannelKey, Method,
AuthHandlerActionTreeName,FormParameters, FilesToSend, HeadersToSend,
HttpProperties);  

//Create Model from RDF payload
rdf=RDFParse(x);  

//Retrieve all statements from model
allStatements=RDFSelect(rdf,null,null,null);  

//Output subject, predicate, and objects from all statements returned,
//whose object is a literal, to the log
Size=Length(allStatements);
Log(Size);
Count=0;
While (Count < Size) {
  if (!RDFNodeIsAnon(allStatements[Count].subject)) {
    Log (allStatements [Count].subject + " " + allStatements [Count].predicate + " " + allStatements [Count].object + ".");
  }
  Count = Count + 1;
}

RDFParse
You can use the RDFParse function to help other functions read and parse an RDF object. It retrieves the data from a string that contains an RDF payload and returns a model that contains the RDF payload passed to it. Other functions can use this model to further read and parse an RDF object.

Syntax
The RDFParse function has the following syntax:
[Model =] RDFParse(Payload)

Parameters
The RDFParse function has the following parameters:

Parameter
Payload

Type   String

Description
Payload containing the RDF
The following example provides statements based on an RDF that is retrieved by the GetHTTP function:

```java
//Retrieve the RDF from OSLC provider through the GetHTTP method
HTTPHost="example.com";
HTTPPort=9081;
Protocol="https";
Path="/NCICLUSTER_NCI_oslc/data/resourceShapes/alerts";
ChannelKey="tom";
Method="";
AuthHandlerActionTreeName="";
FormParameters=NewObject();
FilesToSend=NewObject();
HeadersToSend=NewObject();
HttpProperties = NewObject();
HttpProperties.UserId="impactadmin";
HttpProperties.Password="passw0rd";
x=GetHTTP(HTTPHost, HTTPPort, Protocol, Path, ChannelKey, Method, 
AuthHandlerActionTreeName, FormParameters, FilesToSend, HeadersToSend, 
HttpProperties);
//Create Model from RDF payload
rdf=RDFParse(x);
```

**RDFRegister**

You can use the RDFRegister function to help you to register service providers or OSLC resources with the registry server.

Before you can register a service provider or resource, you must use the other RDF policy functions to build an RDF model that meets the OSLC and Registry Services requirements.

After you build the RDF model, use the RDFRegister function to register the RDF with the resource registry contained in the Registry Services integration service.

If the service provider or OSLC resource is registered successfully, the RDFRegister function returns the resource location of the registration record. The following variables and their return values are also returned to provide more information:
- `ResultCode` contains the result code for the response.
- `HeadersReceived` contains the headers received in the response.
- `HeadersSent` contains the headers sent in the response.
- `ResponseBody` contains the response body text.

If the query parameters are set in the URL and you use the RDFRegister policy function to register a service provider, you must manually add the location of the service provider to the policy. For example:

```java
RDFStatement(newModel, manu[0].subject, 
"http://open-services.net/ns/core#serviceProvider", serviceProviderURL, true);
```

If you use the query string inside the path, you must also ensure that the `FormParameters` parameter is set to null. For example:

```java
FormParameters=null;
```

Finally, you must ensure that the policy contains pagination information. For example:

```java
Path="/NCICLUSTER_NCI_oslc/data/mysql1?oslc.paging=true&oslc.pageSize=100";
```
If unsuccessful, the return value of the resource location registration record is null. Error code information is returned in the ErrorReason and ResultCode variables.

**Syntax**

The RDFRegister function has the following syntax:

```java
[ String = ] RDFRegister(URI, Username, Password, Model)
```

where Username can be a null or void string to specify that no authentication is required.

**Parameters**

The RDFRegister function has the following parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>URI</td>
<td>String</td>
<td>Registry Services server creation factory URI</td>
</tr>
<tr>
<td>Username</td>
<td>String</td>
<td>User name for the Registry Services server</td>
</tr>
<tr>
<td>Password</td>
<td>String</td>
<td>Password for the Registry Services server</td>
</tr>
<tr>
<td>Model</td>
<td>Model</td>
<td>Model that contains the RDF</td>
</tr>
</tbody>
</table>

**Example**

The following example manually registers a service provider and a set of resources that have been exposed by the OSLC server provider in Netcool/Impact.

The Registry Services server information is as follows:

RegistryServerProviderCFUri="http://<registry_services_server>:16310/oslc/pr/collection";
RegistryServerResourceCFUri="http://<registry_services_server>:16310/oslc/rr/registration/collection";
RegistryServerUsername="system";
RegistryServerPassword="manager";

The Netcool/Impact server information is as follows:

HTTPHost="<impact_server>";
HTTPPort=9080;
Protocol="http";
Path="/NCICLUSTER_NCI_oslc/provider/provider01";
Path="/NCICLUSTER_NCI_oslc/data/computer";
ChannelKey="";
Method="GET";
AuthHandlerActionTreeName="";
FormParameters=NewObject();
FilesToSend=NewObject();
HeadersToSend=NewObject();
HttpProperties = NewObject();
HttpProperties.UserId="impactadmin";
HttpProperties.Password="passw0rd";
HttpProperties.AuthenticationScheme="basic";

Get the service provider RDF from Netcool/Impact:
Create an RDF model that is based on the service provider response:

```java
serviceProviderModel = RDFParse(serviceProviderResponse)
```

Register the service provider in the provider registry:

```java
serviceProviderURL = RDFRegister(RegistryServerProviderCFUri,
RegistryServerUsername, RegistryServerPassword, serviceProviderModel);
log("Provider Registry-Service Provider URL: " + serviceProviderURL);
```

Get all the computer system resources from Netcool/Impact:

```java
allResources = GetHTTP(HTTPHost, HTTPPort, Protocol, Path2, ChannelKey,
Method, AuthHandlerActionTreeName, null, FilesToSend, HeadersToSend,
HttpProperties);
```

Create an RDF model that is based on the resource response:

```java
allResourceModel = RDFParse(allResources);
```

Register each computer system and a set of properties with the resource registry:

```java
statements = RDFSelect(allResourceModel, null,
"http://jazz.net/ns/ism/event/impact#data/computer/ID", null);
size = Length(statements);
count = 0;
while (count < size) {
    Path3 = statements[count].subject;
    //Get the individual computer system resource
    resourceResponse = GetHTTP(HTTPHost, HTTPPort, Protocol, Path3, ChannelKey,
Method, AuthHandlerActionTreeName, null, FilesToSend, HeadersToSend,
HttpProperties);
    resourceModel = RDFParse(resourceResponse);
    newModel = RDFModel();
    manu = RDFSelect(resourceModel, null,
"http://open-services.net/ns/crtv#manufacturer", null);
    model = RDFSelect(resourceModel, null,
"http://open-services.net/ns/crtv#model", null);
    serial = RDFSelect(resourceModel, null,
"http://open-services.net/ns/crtv#serialNumber", null);
    RDFModelUpdateNS(newModel, "crtv", "http://open-services.net/ns/crtv#");
    RDFModelUpdateNS(newModel, "oslc", "http://open-services.net/ns/core#");
    RDFStatement(newModel, manu[0].subject,
"http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
"http://open-services.net/ns/core#ComputerSystem", true);
    RDFStatement(newModel, manu[0].subject, manu[0].predicate, manu[0].object,
RDFNodeIsResource(manu[0].object));
    RDFStatement(newModel, manu[0].subject, model[0].predicate, model[0].object,
RDFNodeIsResource(model[0].object));
    RDFStatement(newModel, manu[0].subject, serial[0].predicate,
serial[0].object, RDFNodeIsResource(serial[0].object));
    RDFStatement(newModel, manu[0].subject, "http://open-services.net/ns/core#serviceProvider",
serviceProviderURL, true);
    Register the resource in the resource registry:
```
```
resourceURL = RDFRegister(RegistryServerResourceCFUri, 
RegistryServerUsername, RegistryServerPassword, newModel);
log("Resource Registry-Resource URL: " +resourceURL);

count=count+1;
}

RDFUnRegister

To remove the registration record of a service provider or resource from the 
registry server, use the RDFUnRegister function to supply the location of the 
registration record, the Registry Services server username and password, and the 
registration record that you want to remove.

Before you can remove the registration record of a service provider, you must 
remove all the registration records for the associated OSLC resources.

If successful, the RDFUnRegister function returns the message code 204 and the 
value true. The following variables and their return values are also returned to 
provide additional information:
• ResultCode contains the result code for the response.
• HeadersReceived contains the headers received in the response.
• HeadersSent contains the headers sent in the response.
• ResponseBody contains the response body text.

If unsuccessful, the return value of the resource location registration record is false. 
Error code information is returned in the ErrorReason and ResultCode variables.

Syntax

The RDFUnRegister function has the following parameters:
[ String =] RDFUnRegister(URI, Username , Password)

where Username can be a null or void string to specify that no authentication is 
required.

Parameters

Table 72. RDFUnRegister function parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>URI</td>
<td>String</td>
<td>Location that contains the registration record for the resource or service provider</td>
</tr>
<tr>
<td>Username</td>
<td>String</td>
<td>User name for the Registry Services server</td>
</tr>
<tr>
<td>Password</td>
<td>String</td>
<td>Password for the Registry Services server</td>
</tr>
</tbody>
</table>

Example of how to remove the registration of a service provider

The following example demonstrates how to remove the registration of the service 
provider.

The service provider location is:
http://<registryserver>:16310/oslc/providers/6577
Use the RDFUnRegister function to remove the registration. For example:

```plaintext
// Registry server information
ServiceProviderUri="http://<registryserver>:16310/oslc/providers/6577";
RegistryServerUsername="system";
RegistryServerPassword="manager";
result = RDFUnRegister(ServiceProviderUri, RegistryServerUsername, RegistryServerPassword);
```

Example of how to remove the registration of an OSLC resource

The following example demonstrates how to use the policy function to remove the registration of an OSLC resource.

```plaintext
registrationURL = "http://oslcregistryserver.com:16310/oslc/registration/1351071987349";
providerURL = "http://oslcregistryserver.com:16310/oslc/providers/1351071987343";
RegistryServerUsername="smadmin";
RegistryServerPassword="password";
returnString = RDFUnRegister(registrationURL, RegistryServerUsername, RegistryServerPassword);
```

RDFSelect

You can use the RDFSelect function to assist in reading and parsing an RDF. To retrieve statements based on an RDF model, you call the RDFSelect function and pass the RDF model that is created by the RDFParse function. You can filter based on subject, predicate, and object.

The RDFSelect function returns an array of statements that are based on the filter, retrieving values for the subject, predicate, and object variables. You can use it to create RDF statements or triples. You can also use it to filter statements. If you do not want to filter your results, you specify null or empty values for the input parameters.

Syntax

The RDFSelect function has the following syntax:

```plaintext
[Array = ] RDFSelect(Model, Subject, Predicate, Object)
```

Parameters

The RDFSelect function has the following parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>Model</td>
<td>The model that contains the RDF payload</td>
</tr>
<tr>
<td>Subject</td>
<td>String</td>
<td>Filters for the subject value in RDF statements</td>
</tr>
<tr>
<td>Predicate</td>
<td>String</td>
<td>Filters for the predicate value in RDF statements</td>
</tr>
<tr>
<td>Object</td>
<td>String</td>
<td>Filters for the object value in RDF statements</td>
</tr>
</tbody>
</table>

The following example provides statements based on an RDF that is retrieved by the GetHTTP function:

```plaintext
// Retrieve the RDF from OSLC provider through the GetHTTP method
HTTPHost="example.com";
HTTPPort=9081;
```
The following piece of code provides all statements that contain a particular subject name:

```java
//Retrieve the RDF from OSLC provider through the GetHTTP method
HTTPHost="example.com";
HTTPPort=9081;
Protocol="https"; Path="/NCICLUSTER_NCI_oslc/data/resourceShapes/alerts"; ChannelKey="tom";
Methods=""; AuthHandlerActionTreeName="";
FormParameters=NewObject();
FilesToSend=NewObject();
HeadersToSend=NewObject();
HttpProperties = NewObject();
HttpProperties.UserId="impactadmin";
HttpProperties.Password="passw0rd";
x=GetHTTP(HTTPHost, HTTPPort, Protocol, Path, ChannelKey, Method, AuthHandlerActionTreeName, FormParameters, FilesToSend, HeadersToSend, HttpProperties);
//Create Model from RDF payload
rdf=RDFParse(x);
//Retrieve statements containing subject from model
statements=RDFSelect(rdf,"http://ibm.com/ns/netcool-impact/data/SCR_Components#MYCLASS",null,null);
//Output subject, predicate, and objects from all statements returned to the log
Size=Length statements; log(Size);
Count=0; While (Count < Size) {
  log (stmt[Count].subject + " " + stmt[Count].predicate + " " + stmt[Count].object + ".");
  Count = Count + 1;
}
```

### RDFStatement

You can use the RDFStatement function to create and add statements to an RDF model.

You specify the following parameters in the function:

- Model object
- Subject string or resource
- Predicate string or property
• Object string or RDF node

If the Object input parameter is a string, you must specify a flag to determine whether the object input parameter is RDF literal or an RDF resource type.

To create an anonymous resource in the statement, define the value for the Subject as null. When this parameter value is set to null, the policy function creates an anonymous resource in the statement.

Syntax

The RDFStatement function has the following syntax:

[Statement =] RDFStatement (Model, Subject, Predicate, Object, isResource)

Parameters

If the Object input parameter is a string, you must specify the isResource parameter. The RDFStatement function has the following parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>Model</td>
<td>Model object that the statement is added to.</td>
</tr>
<tr>
<td>Subject</td>
<td>String, resource, or null</td>
<td>Subject value of statement. If this parameter is set to null, the function creates an anonymous resource in the statement. The policy function returns a statement instead of a model.</td>
</tr>
<tr>
<td>Predicate</td>
<td>String or property</td>
<td>Predicate value of statement.</td>
</tr>
<tr>
<td>Object</td>
<td>String or RDF node</td>
<td>Object value of statement.</td>
</tr>
<tr>
<td>isResource</td>
<td>Boolean</td>
<td>Determines whether an object is a resource or a literal.</td>
</tr>
</tbody>
</table>

The following example shows how to create a basic RDF with a single statement.

1. Use the RDFModel policy function to create a model:

```java
Model = RDFModel();
RDFModelUpdateNS(model,"oslc","http://open-services.net/ns/core#");
subject = "http://ibm.com/ns/netcool-impact/data/SCR_Components#MYCLASS";
property = "http://open-services.net/ns/core#name";
value = "Brian";
isResource = false;
```

2. Use the RDFStatement policy function to create a statement:

```java
RDFStatement(model,subject,property,value,isResource);
```

3. Finally, specify how the RDF model is output:

```java
body = RDFModelToString(model, null);
```

4. Finally, specify how the RDF model is output:

```java
body = RDFModelToString(model, null);
```
The following example shows how to create a model that is based on an existing model and that uses only the subjects that the user is interested in:

1. Use the GetHTTP method to retrieve the RDF from the OSLC provider:
   ```plaintext
   HTTPHost="example.com";
   HTTPPort=9081;
   Protocol="https";
   Path="/NCICLUSTER_NCI_oslc/data/resourceShapes/alerts";
   ChannelKey="tom";
   Method="";
   AuthHandlerActionTreeName="";
   FormParameters=NewObject();
   FilesToSend=NewObject();
   HeadersToSend=NewObject();
   HttpProperties = NewObject();
   HttpProperties.UserId="impactadmin";
   HttpProperties.Password="passw0rd";
   x=GetHTTP(HTTPHost, HTTPPort, Protocol, Path, ChannelKey, Method, AuthHandlerActionTreeName, FormParameters, FilesToSend, HeadersToSend, HttpProperties);
   ```

2. Create the RDF model from the RDF payload:
   ```plaintext
   rdf=RDFParse(x);
   ```

3. Define a subject to filter:
   ```plaintext
   mySubject="http://ibm.com/ns/netcool-impact/data/SCRComponents#ID";
   ```

4. Retrieve all the statements that contain mySubject from the model:
   ```plaintext
   allStatements=RDFSelect(rdf,mySubject,null,null);
   ```

5. Use the RDFModel function to create a new model:
   ```plaintext
   newModel = RDFModel()
   ```

6. Use the RDFModelUpdateNS function to add the required namespaces to the model:
   ```plaintext
   RDFModelUpdateNS(newModel,"oslc","http://open-services.net/ns/core#");
   RDFModelUpdateNS(newModel,"rdfs","http://www.w3.org/2000/01/rdf-schma#");
   RDFModelUpdateNS(newModel,"dcterms","http://purl.org/dc/terms/");
   ```

7. Use the RDFStatement function to add the statements from the old model to the new model
   ```plaintext
   Size=Length(stmt);
   Count=0;
   While (Count < Size) {
     RDFStatement(newModel, stmt[Count].subject, stmt[Count].predicate, stmt[Count].object, IsRDFNodeResource(stmt[Count].object));
     Count = Count + 1;
   }
   ```

8. Output the new model to the log:
   ```plaintext
   log(RDFModelToString(model, null));
   ```

---

**ReceiveJMSMessage**

The ReceiveJMSMessage function retrieves a message from the specified Java Message Service (JMS) destination.

To retrieve the message, you call this function and pass a JMS data source, and a message properties context as input parameters.

**Syntax**

The ReceiveJMSMessage function has the following syntax:

ReceiveJMSMessage(DataSource, MethodCallProperties)
Parameters

The ReceiveJMSMessage function has the following parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataSource</td>
<td>String</td>
<td>Existing, and valid JMS data source.</td>
</tr>
<tr>
<td>MethodCallProperties</td>
<td>Context</td>
<td>Context that contains optional MessageSelector and Timeout.</td>
</tr>
</tbody>
</table>

RemoveHibernation

The RemoveHibernation function deletes a data item from the Hibernation data type and removes it from the hibernation queue.

To remove a hibernation, you call RemoveHibernation and pass the action key for the data item as an input parameter.

Syntax

The RemoveHibernation function has the following syntax:

RemoveHibernation(ActionKey)

Parameters

The RemoveHibernation function has the following parameter.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActionKey</td>
<td>String</td>
<td>String that uniquely identifies the hibernating policy.</td>
</tr>
</tbody>
</table>

Example

The following example shows how to remove a hibernation whose action key is ActionKey0001.

// Call RemoveHibernation and pass the action key for the hibernation as an input parameter

ActionKey = "ActionKey0001";

RemoveHibernation(ActionKey);

Replace

The Replace function uses regular expressions to replace a substring of a specified string.

Note: To replace a backslash character (\) in a string, you must escape the character twice in the expression, resulting in a string with four backslash characters (\\\). For example, to replace the substring first\second in a string, you must specify it as first\\\\second.
Syntax

The Replace function has the following syntax:

\[ String = \text{Replace}(Expression, Pattern, Substitution, \text{MaxNum}) \]

Parameters

The Replace function has the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>String</td>
<td>String that contains the substring to be replaced.</td>
</tr>
<tr>
<td>Pattern</td>
<td>String</td>
<td>Substring pattern to be replace.</td>
</tr>
<tr>
<td>Substitution</td>
<td>String</td>
<td>String to substitute for the substring.</td>
</tr>
<tr>
<td>MaxNum</td>
<td>Integer</td>
<td>Maximum number of replacements to perform.</td>
</tr>
</tbody>
</table>

Return value

The resulting string.

Example

The following example shows how to replace a substring in a string.

\begin{verbatim}
MyString = "New York";
Pattern = "York";
Substitution = "Jersey";
MyReplace = Replace(MyString, Pattern, Substitution, 1);
Log(MyReplace);
\end{verbatim}

This example prints the following message to the policy log:

Parser Log: New Jersey

RESTfulAPIGET

The RESTfulAPIGET function retrieves resources from a RESTful API.

You can use the RESTfulAPIGET function to issue HTTP GET requests to RESTful APIs.

You can connect through a proxy server if the data source has enabled it. To enable SSL connections between Netcool/Impact servers and external servers, refer to the Enabling SSL connections with external servers topic within the Security chapter of the Netcool/Impact Administration Guide.

Syntax

The following is the syntax for RESTfulAPIGET:

\[ \text{RESTfulAPIGET}(\text{DataSource}, \text{Path}, \text{Config}) \]
Parameters

The RESTfulAPIGET function has the following parameters.

**Table 78. RESTfulAPIGET function parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DataSource</strong></td>
<td>String</td>
<td>Name of the data source defined in the data model.</td>
</tr>
<tr>
<td><strong>Path</strong></td>
<td>String</td>
<td>URL path after the hostname to the resource that you want to retrieve.</td>
</tr>
<tr>
<td><strong>Config</strong></td>
<td>String</td>
<td>Object containing any extra request information that you want to add to the request.</td>
</tr>
<tr>
<td></td>
<td>Object</td>
<td></td>
</tr>
</tbody>
</table>

The object must contain name-value pairs. Valid variables are:

- **Params:** Any extra HTTP parameters to be added to, or overridden from, the data model.
- **Headers:** Any extra HTTP headers to be added to, or overridden from, the data model.
- **ConnectionTimeout:** Sets the timeout until a connection is established. The default value is 0, which means that timeout is not used.
- **ResponseTimeout:** Sets the default socket timeout (SO_TIMEOUT) in milliseconds. This is the timeout for waiting for data. If you set the timeout to 0, the function waits indefinitely for a response from the socket.

**Return Values**

When you call the RESTfulAPIGET function, it outputs the following variables:

- **status:** The status code associated with the HTTP call.
- **data:** The HTTP response received from the data source.
- **config:** The configuration object sent to the data source.
- **statusText:** The status text associated with the latest response.
- **headers:** The HTTP response headers.

**Example 1**

A RESTfulAPIGET request to a RESTful data source with an additional parameter:

```javascript
Config = NewObject();
Param1 = NewObject();
Param1['size'] = Int(20);
Config['params'] = Param1;
var hits = RESTfulAPIGET("ElasticDS","logstash-serverlogs/_search", Config);
Log("Hits:"+hits);
```

This is equivalent to the raw request: `<hostname>/logstash-serverlogs/_search?size=20`

**Example 2**

A RESTfulAPIGET request to a RESTful data source with an additional header:

```javascript
Config = NewObject();
Header1 = NewObject();
Header1['Accept'] = "application/json; charset=UTF-8";
```
Config['Headers'] = Header1;
var users = RESTfulAPIGET("CloudantDB","users", Config);
if (users.status == 404) {
    Log("No users found");
}

### Example 3

A RESTfulAPIGET request to a RESTful data source that handles a JSONArray Response:

```javascript
var result = RESTfulAPIGET("JSONArrayDS", "comments");
var commentsArray = result.data;
//CommentArray contains objects in the form { id: "...", message: "...."}
for (var i = 0; i < 20; i++) {
    Log("Comment:" + commentsArray[i].message);
}
```

Combing different restful functions in one policy.

```javascript
var objectserverResponse = RESTfulAPIGET("OMNI_REST");
if (objectserverResponse.status == 200) {
    //Got data from objectserver try move it to our ElasticSearch instance
    var data = objectserverResponse.data.rowset.rows;
    Config = {};
    Config['data'] = { latest_rows: data };

    //Update headers for post request
    var headers = {};
    headers['Content-Type'] = "application/x-www-form-urlencoded";
    Config['headers'] = headers;
    var elasticResponse = RESTfulAPIPOST("ElasticDS","objectserver/alerts/status", Config);
    Log("Elastic Response:" + elasticResponse.status);
} else {
    Log("Unable to fetch objectserver rows");
}
```

### RESTfulAPIPOST

The RESTfulAPIPOST function sends resources to a RESTful API.

You can use the RESTfulAPIPOST function to issue HTTP POST requests to RESTful APIs.

You can connect through a proxy server if the data source has enabled it. To enable SSL connections between Netcool/Impact servers and external servers, refer to the Enabling SSL connections with external servers topic within the Security chapter of the Netcool/Impact Administration Guide.

### Syntax

The following is the syntax for RESTfulAPIPOST:

```
RESTfulAPIPOST(DataSource, Path, Config)
```

### Parameters

The RESTfulAPIPOST function has the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataSource</td>
<td>String</td>
<td>Name of the data source defined in the data model.</td>
</tr>
</tbody>
</table>
Table 79. RESTfulAPIPOST function parameters (continued)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path</td>
<td>String</td>
<td>URL path after the hostname to the resource to which you want to post.</td>
</tr>
<tr>
<td>Config</td>
<td>String</td>
<td>Object containing any extra request information that you want to add to the request.</td>
</tr>
<tr>
<td></td>
<td>Object</td>
<td>The object must contain name-value pairs. Valid variables are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <strong>Params</strong>: Any extra HTTP parameters to be added to, or overridden from, the data model.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <strong>Headers</strong>: Any extra HTTP headers to be added to, or overridden from, the data model.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <strong>ConnectionTimeout</strong>: Sets the timeout until a connection is established. The default value is 0, which means that timeout is not used.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <strong>ResponseTimeout</strong>: Sets the default socket timeout (SO_TIMEOUT) in milliseconds. This is the timeout for waiting for data. If you set the timeout to 0, the function waits indefinitely for a response from the socket.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <strong>Data</strong>: Content that you want to send to the API.</td>
</tr>
</tbody>
</table>

Return Values

When you call the RESTfulAPIPOST function, it outputs the following variables:

• **status**: The status code associated with the HTTP call.
• **data**: The HTTP response received from the data source.
• **config**: The configuration object sent to the data source.
• **statusText**: The status text associated with the latest response.
• **headers**: The HTTP response headers.

Example 1

A RESTfulAPIPOST request to send data through the parameters property:

```javascript
var config = {};  
config['params'] = { channel: '#general', text: 'Hello Slack!' };  
var result = RESTfulAPIPOST("Slack", "chat.postMessage", config);  
Log("Result:" + result);
```

Example 2

A RESTfulAPIPOST request to send an event to the IBM Alert Notification Service:

```javascript
data = {};  
config = {};  
data['What'] = 'Disk Space Full';  
data['Where'] = 'SiteABC';  
data['Severity'] = 'Major';  
config['data'] = data;  
var result = RESTfulAPIPOST("IBMAgentNotification", "", config);  
Log("Result:"+result);
```

Example 3

A RESTfulAPIPOST request to post a new row to the ObjectServer:
var coldescArr = '[[ "type":"string", "name":"Identifier"]]';
var data = '[["Identifier":"ImpactEventInstance0000" ]]';
var osdata = '{"rowset": { "coldesc":' + coldescArr + ' ,"rows": ' + data + ' }}';

Config = {};
Config['data'] = osdata;
Log(osdata);
var osResponse = RESTfulAPIPOST("ObjectServerRest","", Config);
Log("OS Response:" + osResponse);

---

### RESTfulAPIPUT

The RESTfulAPIPUT function sends requests to update or create resources to a RESTful API.

You can use the RESTfulAPIPUT function to issue HTTP PUT requests to RESTful APIs.

You can connect through a proxy server if the data source has enabled it. To enable SSL connections between Netcool/Impact servers and external servers, refer to the *Enabling SSL connections with external servers* topic within the *Security* chapter of the Netcool/Impact Administration Guide.

#### Syntax

The following is the syntax for RESTfulAPIPUT:

RESTfulAPIPUT(DataSource, Path, Config)

#### Parameters

The RESTfulAPIPUT function has the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataSource</td>
<td>String</td>
<td>Name of the data source defined in the data model.</td>
</tr>
<tr>
<td>Path</td>
<td>String</td>
<td>URL path after the hostname to the resource that you want to issue the request against.</td>
</tr>
</tbody>
</table>
Table 80. RESTfulAPIPUT function parameters (continued)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
</table>
| Config    | String | Object | Object containing any extra request information that you want to add to the request.  

The object must contain name-value pairs. Valid variables are:

- **Params**: Any extra HTTP parameters to be added to, or overridden from, the data model.
- **Headers**: Any extra HTTP headers to be added to, or overridden from, the data model.
- **ConnectionTimeout**: Sets the timeout until a connection is established. The default value is 0, which means that timeout is not used.
- **ResponseTimeout**: Sets the default socket timeout (SO_TIMEOUT) in milliseconds. This is the timeout for waiting for data. If you set the timeout to 0, the function waits indefinitely for a response from the socket.
- **Data**: Content that you want to send to the API.
- **File**: File that you want to send to the API.

**Return Values**

When you call the `RESTfulAPIPUT` function, it outputs the following variables:

- **status**: The status code associated with the HTTP call.
- **data**: The HTTP response received from the data source.
- **config**: The configuration object sent to the data source.
- **statusText**: The status text associated with the latest response.
- **headers**: The HTTP response headers.

**Example 1**

A `RESTfulAPIPUT` request to a RESTful data source with an additional headers and parameters:

```javascript
var config = {};  
config['headers'] = { Content-Type: 'application/x-www-form-urlencoded' };  
config['data'] = { title: 'My first blog', text: 'Hello world blog test' };  
var result = RESTfulAPIPUT("ElasticDS", "blog/blog1/123", config);
```

**Example 2**

A `RESTfulAPIPUT` request to a RESTful data source with an additional headers and parameters:

```javascript
var config = {};  
config['headers'] = { Content-Type: 'image/jpeg' };  
config['params'] = { rev: '1test' };  
config['file'] = { picture: 'C:\Impact\tmp\screen1.jpg' };  
var result = RESTfulAPIPUT("CloudantDB", "test/testdoc123", config);
```

**RESTfulAPIDELETE**

The `RESTfulAPIDELETE` function deletes resources from a RESTful API.
You can use the `RESTfulAPIDELETE` function to issue HTTP DELETE requests to RESTful APIs.

You can connect through a proxy server if the data source has enabled it. To enable SSL connections between Netcool/Impact servers and external servers, refer to the `Enabling SSL connections with external servers` topic within the `Security` chapter of the Netcool/Impact Administration Guide.

**Syntax**

The following is the syntax for `RESTfulAPIDELETE`:

```
RESTfulAPIDELETE(DataSource, Path, Config)
```

**Parameters**

The `RESTfulAPIDELETE` function has the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataSource</td>
<td>String</td>
<td>Name of the data source defined in the data model.</td>
</tr>
<tr>
<td>Path</td>
<td>String</td>
<td>URL path after the hostname to the resource that you want to delete.</td>
</tr>
<tr>
<td>Config</td>
<td>String</td>
<td>Object containing any extra request information that you want to add to the request.</td>
</tr>
<tr>
<td></td>
<td>Object</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The object must contain name-value pairs. Valid variables are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <code>Params</code>: Any extra HTTP parameters to be added to, or overridden from, the data model.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <code>Headers</code>: Any extra HTTP headers to be added to, or overridden from, the data model.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <code>ConnectionTimeout</code>: Sets the timeout until a connection is established. The default value is 0, which means that timeout is not used.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <code>ResponseTimeout</code>: Sets the default socket timeout (SO_TIMEOUT) in milliseconds. This is the timeout for waiting for data. If you set the timeout to 0, the function waits indefinitely for a response from the socket.</td>
</tr>
</tbody>
</table>

**Return Values**

When you call the `RESTfulAPIDELETE` function, it outputs the following variables:

- `status`: The status code associated with the HTTP call.
- `data`: The HTTP response received from the data source.
- `config`: The configuration object sent to the data source.
- `statusText`: The status text associated with the latest response.
- `headers`: The HTTP response headers.

**Example 1**

A `RESTfulAPIDELETE` request to a RESTful data source:

```javascript
var result = RESTfulAPIDELETE("ElasticSearch", "blog/test1/1");
```
Example 2

A RESTful DELETE request to a RESTful data source with an additional parameter:

```javascript
var config = {};
config['params'] = { rev: '123' };
var result = RESTfulAPIDELETE("CloudantDB", "blog/test1/1", config);
```

**ReturnEvent**

The `ReturnEvent` function inserts, updates, or deletes an event from an event source.

To insert a new event, you first create a new event container using `NewEvent` and then populate its member variables with the field values for the new event. Then you return the event to the event source using `ReturnEvent`.

Note that the container can contain a variable named `EventReaderName` that specifies the name of an event reader service. If the policy is to be run by another means besides an event reader (for example, using the GUI or `nci_trigger` command), you must assign a value to the `EventReaderName` variable. This value is the name of the event reader you want to use to send the new event.

To update an event, you change the values of the member variables in the event container as needed and then return the event to the event source using `ReturnEvent`.

To delete an event, you set the `DeleteEvent` member variable in the event container to `true` and then return the event to the event source using `ReturnEvent`.

**Syntax**

The `ReturnEvent` function has the following syntax:

```javascript
ReturnEvent(Event)
```

**Parameters**

The `ReturnEvent` function has the following parameter.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event</td>
<td>Event container</td>
<td>Event container that represents the event that you want to insert, update or delete.</td>
</tr>
</tbody>
</table>

**Examples**

The following example shows how to insert a new event using `ReturnEvent`.

```javascript
// Set the EventReaderName variable only if policy is to be triggered using something other than an event reader service
MyEvent.EventReaderName = "OMNIbusEventReader";

// Create a new event container using `NewEvent` and populate its member variables
MyEvent = NewEvent("OMNIbusEventReader");
MyEvent.Node = "ACHILLES";
```
MyEvent.Summary = "Node not responding.";

// Return the new event to the event source
ReturnEvent(MyEvent);

The following example shows how to update an event in an event source.
// Update the values of the member variables
// in the event container
EventContainer.Summary = EventContainer.Summary + ": Updated by Netcool/Impact";
// Return the event to the event source
ReturnEvent(EventContainer);

The following example shows how to delete an event in an event source.
// Set the value of the DeleteEvent member variable
// to true
EventContainer.DeleteEvent = true;
// Return the event to the event source
ReturnEvent(EventContainer);

RExtract

The RExtract function uses regular expressions to extract a substring from a string.

This function supports Perl 5 style regular expressions.

Syntax

The RExtract function has the following syntax:

\[ \text{String} = \text{RExtract(} \text{Expression, Pattern} \text{)} \]

Parameters

The RExtract function has the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>String</td>
<td>String that contains the substring to extract. The data to extract is within () in the Expression.</td>
</tr>
<tr>
<td>Pattern</td>
<td>String</td>
<td>Regular expression pattern that specifies the substring to extract.</td>
</tr>
</tbody>
</table>

Return value

The extracted string.

Example

The following example shows how to use the RExtract function.
Log(RExtract("Not responding to ping on host DB_01", ".*\s(DB_01).*"));
This example prints the following message to the policy log:
Parser Log: DB_01

RExtractAll

The RExtractAll function uses regular expression matching to extract multiple substrings from a string.

The resulting matches are returned as elements in an array. This function supports regular expressions similar to Perl 5.

To extract multiple substrings from a string, you call RExtractAll and pass a string expression and a regular expressions pattern. The pattern specifies the matching characters inside the expression and specifies which characters in the matching substring to extract. You identify the characters you want to extract by enclosing them in parentheses inside the pattern.

The default behavior for the RExtractAll function is to return the last match in a pattern. For example, if you have "<test1>;<test2>;<test3>" and the pattern returns everything between the parentheses ">", by default it returns test3.

You can set the Boolean flag to false to return all the matches in a pattern, then this example "<test1>;<test2>;<test3>" returns all the matches in a pattern test1, test2, test3. If you set the Boolean flag to true the RExtractAll function returns the last match in a pattern, the same as the default behavior. The function assigns the extracted character strings as elements in an array and passes the array back to the policy.

Syntax

The RExtractAll function has the following syntax options and the use of the Boolean flag is optional:

Array = RExtractAll(Expression, Pattern);
Array = RExtractAll(Expression, Pattern, Flag);

Parameters

The RExtractAll function has the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>String</td>
<td>String that contains the substring to extract.</td>
</tr>
<tr>
<td>Pattern</td>
<td>String</td>
<td>Regular expression pattern that specifies the substrings to extract. You specify the string to extract from the match using parentheses characters.</td>
</tr>
<tr>
<td>Boolean</td>
<td>true or false</td>
<td>Optional. Set the Boolean flag to false to return all the matches in a pattern. Set the Boolean flag to true to return the last match in a pattern.</td>
</tr>
</tbody>
</table>

Return value

An array of the resulting substrings.
Examples

The following examples show how to use the RExtractAll function without a Boolean flag.

Example 1
Expression = "Node is DB_02 on rack RK_419";
Pattern = "\s.*(DB_02).*(RK_419).*";
Log(RExtractAll(Expression, Pattern));

This example prints the following message to the policy log:
Parser Log: {DB_02, RK_419}

Example 2 using IPL
expression="<plantId>122</plantId><plantId>204</plantId><plantId>234</plantId>";
test = RExtractAll(expression, "<plantId\b[^>]*>(.*?)</plantId>",
 testNum = Length(test);
Log ("Test Num= " + testNum);
Log ("Plant ID = " + test);

This example prints the following message to the policy log:
Parser Log: Test Num= 1
Parser Log: Plant ID = {234}

Example 2 using JavaScript
expression="<plantId>122</plantId><plantId>204</plantId><plantId>234</plantId>";
test = RExtractAll(expression, "<plantId\b[^>]*>(.*?)</plantId>",
 testNum = Length(test);
Log ("Test Num= " + testNum);
Log ("Plant ID = " + test);

This example prints the following message to the policy log:
Parser Log: Test Num= 1
Parser Log: Plant ID = {234}

The following example shows how to use the RExtractAll function with a Boolean flag set as false.

Example 3 using IPL
expression="<plantId>122</plantId><plantId>204</plantId><plantId>234</plantId>";
test = RExtractAll(expression, "<plantId\b[^>]*>(.*?)</plantId>",false);
testNum = Length(test);
Log ("Test Num= " + testNum);
Log ("Plant ID = " + (test));

Example 3 using JavaScript
expression="<plantId>122</plantId><plantId>204</plantId><plantId>234</plantId>";
test = RExtractAll(expression, "<plantId\b[^>]*>(.*?)</plantId>",false);
testNum = Length(test);
Log ("Test Num= " + String(testNum));
Log ("Plant ID = " + String(test));

This example prints the following message to the policy log:
06 Jul 2009 15:06:42,798: Parser Log: Test Num= 3
06 Jul 2009 15:06:42,799: Parser Log: Plant ID = {122,204,234}
RollbackTransaction

The RollbackTransaction function rolls back any changes that are done by an SQL operation.

The RollbackTransaction function is a local transactions function that is used in SQL operations. The function causes all changes to the database to be undone when an exception occurs between the BeginTransaction() and CommitTransaction() functions. It is recommended to call the RollbackTransaction() function within the localized exception handler.

You must always call the CommitTransaction() function to complete a transaction even if the RollbackTransaction() function is called.

For more information about the local transactions functions, see Chapter 6, “Using local transactions in a policy,” on page 49.

Arguments

The RollbackTransaction() function takes no arguments.

Note: The ObjectServer does not support the use of the RollbackTransaction function.

SendEmail

The SendEmail function sends an email that uses the email sender service. You can send emails from within a policy to send email notification to administrators and users when a certain event or combination of events occur.

Important: Netcool/Impact does not provide a built-in mail server. Before you can send email, you must make sure that an SMTP server is available in your environment. The Netcool/Impact email sender service must also be running before a policy can successfully send email.

To send email, you call the SendEmail function and pass the following information as input parameters:

• The email address of the recipient.
• The subject line text for the email.
• The body content of the email.
• The name of the email sender.

You can send an email by passing a data item whose Email member variable contains a valid email address. This field must be named Email. You cannot use a data item that has another field that contains email addresses.

The SendEmail function uses UTF-8 encoding of the operating system by default. You can customize the encoding by including the following syntax before you run the call SendEmail:

EncodingChar = "<type of charset>";
ex:
EncodingChar = "windows-1251";
Syntax

The SendEmail function has the following syntax:

```plaintext
SendEmail([User], [Address], [Subject], Message, [Sender], ExecuteOnQueue,
[Attachment],[CallProperties])
```

Parameters

The SendEmail function has the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>Data item</td>
<td>Data item of any data type whose Email field contains the email address of the recipient. Optional.</td>
</tr>
<tr>
<td>Address</td>
<td>String or a context object</td>
<td>Email address for the recipient of the email. If this parameter is specified, the User parameter is ignored. You can specify the To, CC, and BCC fields of the email separately. Multiple email addresses must be separated by a comma (,). Optional.</td>
</tr>
<tr>
<td>Subject</td>
<td>String</td>
<td>Subject for the message. Optional.</td>
</tr>
<tr>
<td>Message</td>
<td>String</td>
<td>Message body for the email.</td>
</tr>
<tr>
<td>Sender</td>
<td>String</td>
<td>Email address for the sender for the email. Optional.</td>
</tr>
<tr>
<td>ExecuteOnQueue</td>
<td>Boolean</td>
<td>Specifies whether to place the outgoing message in the queue that is governed by the command execution manager service. If you specify true, the message is placed in the queue and sent asynchronously by this service. If you specify false, the message is sent directly by Netcool/Impact. In this case, the policy engine waits for the message to be sent successfully before it processes any subsequent instructions.</td>
</tr>
<tr>
<td>Attachment</td>
<td>String</td>
<td>Full file path name to send as an attachment to the email. Optional. Note: When using the SendEmail functionality on Windows, use a single forward slash character instead of double back slash characters to specify the position of a file.</td>
</tr>
<tr>
<td>Properties</td>
<td>Object</td>
<td>Call properties to set additional parameters such as enable HTML or authentication.</td>
</tr>
</tbody>
</table>

Example

The following example shows how to send an email from a policy that uses the email address of the recipient.

```plaintext
// Call SendEmail and pass the address, subject and message text
// as input parameters

Address = "srodriguez@example.com";
Subject = "Netcool/Impact Notification";
Message = EventContainer.Node + " has reported the following error condition: " + EventContainer.Summary;
Sender = "impact";
ExecuteOnQueue = false;
Attachment = "/opt/IBM/tivoli/impact/example.pdf";

SendEmail(null, Address, Subject, Message, Sender, Attachment, ExecuteOnQueue);
```
Here is an example of using the SendMail function if the Address parameter is a context object:

```java
Addresses = NewObject();
Addresses.To = "to@example.com";
Addresses.Cc = "cc1@example.com,cc2@example.com";
Addresses.Bcc = "bcc@example.com";
Subject = "Netcool/Impact Notification";
Message = "Some problem was encountered";
Sender = "impact";
ExecuteOnQueue = false;
SendEmail(null, Addresses, Subject, Message, Sender, null, ExecuteOnQueue);
```

You can override the SMTP host and SMTP port values that are defined in the EmailSender service, from inside the SendEmail function.

If you do not specify the SMTP host and SMTP port before you use the SendEmail function call, the system uses the SMTP host and SMTP port values that are defined in the EmailSender service. If you want to override the SMTP host and the SMTP port in the service, add the following information to the SendEmail function:

```java
SmtpHost = "hostname.example.com";
SmtpPort = <port number example>;
SendEmail(...);
```

In this example, the SMTP host is provided by the EmailSender service.

```java
SmtpPort = 587;
SendEmail(...);
```

In this example, the SMTP port is provided by the EmailSender service.

```java
SmtpHost = "hostname.example.com";
SendEmail(...);
```

### Enabling HTML format and authentication

You also can optionally add extra call properties to enable HTML format in the body of an email and to enable authentication.

- If you enable HTML format, the body text of the email can be formatted with bold and other fonts, tables, and, lists and other formats. HTML format does not apply to the subject of the email. Add the following CallProps variable.

```java
CallProps.EmailContentType="text/html";
```

- You can enable authentication so that a user must use a user name and password to connect to the email server. For example:

```java
CallProps.UseAuth=true;
CallProps.AuthUser="some email";
CallProps.AuthPassword="plaintext or encrypted password using Encrypt or nci_crypt";
```

If an encrypted password is used the following syntax must be used:

```java
CallProps.DecryptPassword=true
```

To pass the CallProps to the SendEmail function use the following example:

```java
SendEmail(null, Address, Subject, Message, Sender, ExecuteOnQueue,CallProps);
```

- An example with authentication enabled, and an email attachment in the file.

```java
CallProps.UseAuth=true;
CallProps.AuthUser="some email";
CallProps.AuthPassword="plaintext or encrypted password using Encrypt or nci_crypt";
Attachment = "full path to file";
```

To pass the CallProps to the SendEmail function use the following example:
SendEmail(null, Address, Subject, Message, Sender, ExecuteOnQueue, Attachment, CallProps);

**SendInstantMessage**

The SendInstantMessage function sends an instant message by using the Jabber service.

You must configure the Jabber service as described in the *online help* before you use this function.

**Syntax**

The SendInstantMessage function has the following syntax:

SendInstantMessage(To, Group, Subject, TextMessage, ExecuteOnQueue)

**Parameters**

The SendInstantMessage function has the following parameters.

**Table 86. SendInstantMessage function parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>To</td>
<td>String</td>
<td>Screen name of the message recipient. To send multiple recipients, use a comma-separated list of names.</td>
</tr>
<tr>
<td>Group</td>
<td>String</td>
<td>String that identifies a chatroom (if any) to join. The format of this string is <em>chatroom name@server/nickname</em>, where <em>chatroom name</em> is the name of the chatroom, <em>server</em> is the name of a Jabber server and <em>nickname</em> is the name that you want to use in the chat. Only available for use with Jabber servers. Optional.</td>
</tr>
<tr>
<td>Subject</td>
<td>String</td>
<td>Title for instant message. Only available for use with Jabber servers. Optional.</td>
</tr>
<tr>
<td>TextMessage</td>
<td>String</td>
<td>Content of the instant message.</td>
</tr>
<tr>
<td>ExecuteOnQueue</td>
<td>Boolean</td>
<td>Specifies whether to place the outgoing message in the queue that is governed by the command execution manager service. If you specify true, the message is placed in the queue and sent asynchronously by this service. If you specify false, the message is sent directly by the Netcool/Impact. In this case, the policy engine waits for the message to be sent successfully before processing any subsequent instructions. Optional.</td>
</tr>
</tbody>
</table>

**Recipient ID formats**

When you call SendInstantMessage, you specify the message recipient using the To parameter. The recipient ID is typically a combination of the messaging system user name and service ID. The service ID is defined in the configuration properties for the Jabber service. A set of abbreviations is also provided that you can use instead of the service ID. The format of the message recipient ID varies, depending on the instant messaging system you are using.

The Jabber interface supports the following messaging systems:

- Jabber
- ICQ
You can use the following recipient ID formats for the Jabber messaging service.

**Table 87. Recipient IDs for Jabber messaging service**

<table>
<thead>
<tr>
<th>Format</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jabber ID and fully qualified service ID</td>
<td><a href="mailto:NetcoolAdmin@jabber.example.com">NetcoolAdmin@jabber.example.com</a></td>
</tr>
<tr>
<td>Jabber ID and fully qualified service ID with resource</td>
<td><a href="mailto:NetcoolAdmin@jabber.example.com">NetcoolAdmin@jabber.example.com</a>/ops1</td>
</tr>
<tr>
<td>Jabber ID and service abbreviation</td>
<td>NetcoolAdmin@jabber</td>
</tr>
</tbody>
</table>

You can use the following recipient ID formats for the ICQ messaging service.

**Table 88. Recipient IDs for ICQ messaging service**

<table>
<thead>
<tr>
<th>Format</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICQ ID and fully qualified service ID</td>
<td>137463829@<a href="mailto:icq@example.com">icq@example.com</a></td>
</tr>
<tr>
<td>ICQ ID and service abbreviation</td>
<td>137463829@icq</td>
</tr>
</tbody>
</table>

You can use the following recipient ID formats for the AIM messaging service.

**Table 89. Recipient IDs for AIM messaging service**

<table>
<thead>
<tr>
<th>Format</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIM ID and fully qualified service ID</td>
<td>NetcoolAdmin@<a href="mailto:aim@example.com">aim@example.com</a></td>
</tr>
<tr>
<td>AIM ID and service abbreviation</td>
<td>NetcoolAdmin@aim</td>
</tr>
</tbody>
</table>

You can use the following recipient ID formats for the Yahoo! messaging service.

**Table 90. Recipient IDs for Yahoo! messaging service**

<table>
<thead>
<tr>
<th>Format</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yahoo! ID and fully qualified service ID</td>
<td><a href="mailto:NetcoolAdmin@yahoo.example.com">NetcoolAdmin@yahoo.example.com</a></td>
</tr>
<tr>
<td>Yahoo! ID and service abbreviation</td>
<td>NetcoolAdmin@yahoo</td>
</tr>
</tbody>
</table>

**Examples**

The following example shows how to send an instant message to a user named NetcoolAdmin.

```java
// Call SendInstantMessage and pass the name of the recipient
// and the content of the message as input parameters
To = "NetcoolAdmin@jabber.example.com";
TextMessage = "Node_0456 is not responding to ping.";
SendInstantMessage(To, null, null, TextMessage, false);
```

The following example shows how to send instant messages to multiple recipients.
// Call SendInstantMessage and pass a comma-separated list of recipients
// and the content of the message as input parameters

To = "NetcoolOperator@jabber.example.com, NetoolAdmin@jabber.example.com";
TextMessage = "Node_0123 is not responding to ping."

SendInstantMessage(To, null, null, TextMessage, false);

The following example shows how to send a message to a chatroom hosted on a Jabber server.
// Call SendInstantMessage and pass the chatroom information,
// a subject and the content of the message as input parameters

Group = "netcoolchat@jabber.example.com/NetcoolAdmin";
Subject = "Alert: Node_0123 status changed.
TextMessage = "Node_0123 has been restored.";

SendInstantMessage(null, Group, Subject, TextMessage, false);

**SendJMSMessage**

The SendJMSMessage function sends a message to the specified destination by using the Java Message Service (JMS) DSA.

To send the message, you call the SendJMSMessage function and pass the JMS data source, a message properties context, and the message body as input parameters.

**Syntax**

The SendJMSMessage function has the following syntax:

SendJMSMessage(DataSource, MethodCallProperties, Message)

**Parameters**

The SendJMSMessage function has the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataSource</td>
<td>String</td>
<td>Valid, and existing JMS data source.</td>
</tr>
<tr>
<td>MethodCallProperties</td>
<td>Context</td>
<td>Context that contains message header, and other JMS properties for the message. Custom message properties are supported.</td>
</tr>
<tr>
<td>Message</td>
<td>String</td>
<td>String or context that contains the body of the message.</td>
</tr>
</tbody>
</table>

**SetFieldValue**

Use the SetFieldValue function to set a public field variable in a Java class to some value.

If it is a static field, then you specify the Java class ClassName. If it is a non-static value, then you provide the instance at TargetObject.
Adding Java archive (JAR) files to the shared library directory

Before you can use this policy function, you must make the Java classes available to Netcool/Impact during run time. To make the Java classes available, complete the following steps:

1. Copy the Java classes to the $NCHOME/dsalib directory.
2. Restart the Impact Server to load the JAR files.

You must repeat this procedure for each Impact Server. This procedure repetition is necessary because the Java class files in the $NCHOME/dsalib directory are not replicated between servers.

Syntax

SetFieldValue( ClassName, TargetObject , FieldName, FieldValue );

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClassName</td>
<td>Name of the Java class. When you use a non-static method call, this parameter would be set to null.</td>
</tr>
<tr>
<td>TargetObject</td>
<td>Name of the instantiated Java object. When you use a static method, this parameter would be set to null.</td>
</tr>
<tr>
<td>FieldName</td>
<td>Name of the public field variable in the Java class that you are setting the value for.</td>
</tr>
<tr>
<td>FieldValue</td>
<td>The value that you are setting the field to.</td>
</tr>
</tbody>
</table>

Returns

N/A

Examples

Reversing the example for “GetFieldValue” on page 155. Using IPL, if you want to set the non-static firstname field in a hypothetical Java class DeveloperAccount:

dev_acct = NewJavaObject("com.ibm.DeveloperAccount", {65224});
SetFieldValue(null, dev_acct, "firstname", "Sam");

Reversing the example for “GetFieldValue” on page 155. Using JavaScript, if you want to set the non-static firstname field in a hypothetical Java class DeveloperAccount:

dev_acct = NewJavaObject("com.ibm.DeveloperAccount", [65224]);
SetFieldValue(null, dev_acct, "firstname", "Sam");

Assuming there is a static counter variable disconnects in a hypothetical Java class com.ibm.tivoli.EventStats, which you want to increment through an Impact policy:

counter = GetFieldValue("com.ibm.tivoli.EventStats", "disconnects");
counter += 1;
SetFieldValue("com.ibm.tivoli.EventStats", null, "disconnects", counter);
SetGlobalVar

The SetGlobalVar function creates in a policy a global variable, which can be accessed from any local functions, library functions, and exception handlers in a policy.

The word “global” refers to the thread scope, which means that any policy code will access its own copy of the global variable from its own thread. Different threads that run the same policy will not interfere with one another, that is if the policy value is changed by one such running thread, the change does not affect the value of the global variable in other threads.

Syntax

SetGlobalVar(variablename, variablevalue)

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>variablename</td>
<td>Name of the variable.</td>
</tr>
<tr>
<td>variablevalue</td>
<td>Initial value of the variable.</td>
</tr>
</tbody>
</table>

Example

This piece of code creates a global variable "MyAge" and sets its initial value to 33:

SetGlobalVar("MyAge", 33)

This piece of code clears the entry for the variable1 variable, by passing null value to SetGlobalVar() call:

SetGlobalVar(variable1, null)

SetServerVar

The SetServerVar function creates a server-wide global variable in a policy.

It can be accessed by any functions and exception handlers, like a global variable created by SetGlobalVar(). Unlike in SetGlobalVar() calls, however, all threads running the same policy will share the same copy of the global variable. So if one thread running the same policy changes the variable value, the change is visible to all other threads running the same policy.

Syntax

SetServerVar(variablename, variablevalue)

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>variablename</td>
<td>Name of the variable.</td>
</tr>
<tr>
<td>variablevalue</td>
<td>Initial value of the variable.</td>
</tr>
</tbody>
</table>
Examples

Here are examples of policies that use the SetServerVar and GetServerVar functions:

```javascript
//Policy 1
function SaveServerVarTest(){
    flag = "THIS FLAG DENOTES A SERVERVAR";
    SetServerVar("runTimeFlag", flag);

    Activate(null, 'Policy2');
}

function GetServerVarTest(){
    Log("runTimeFlag = " + GetServerVar("runTimeFlag"));
}

//Policy2
Log("runTimeFlag = " + GetServerVar("runTimeFlag"));
```

SNMPGetAction

The SnmpGetAction function retrieves a set of SNMP variables from the specified agent.

The values are stored in a variable named ValueList. This function operates by sending an SNMP GET command to the specified agent.

When you call SnmpGetAction, you pass an SNMP data type and, for SNMP v3, any authorization parameters that are required. To override the agent and variable information specified in the SNMP data type, you can also optionally pass a host name, a port number, a list of OIDs, and other information needed to retrieve the data.

Syntax

The following is the syntax for SnmpGetAction:

```
SnmpGetAction(TypeName, [HostId], [Port], [VarIdList], [Community], [Timeout], [Version], [UserId], [AuthProtocol], [AuthPassword], [PrivPassword], [ContextId], [ContextName])
```

Parameters

The SNMPGetAction function has the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TypeName</td>
<td>String</td>
<td>Name of the SNMP data type that specifies the host name, port, OIDs, and other information needed to retrieve the SNMP data.</td>
</tr>
<tr>
<td>HostId</td>
<td>String</td>
<td>Optional. Host name or IP address of the system where the SNMP agent is running. Overrides value specified in the SNMP data type.</td>
</tr>
<tr>
<td>Port</td>
<td>Integer</td>
<td>Optional. Port where the SNMP agent is running. Overrides value specified in the SNMP data type.</td>
</tr>
<tr>
<td>VarIdList</td>
<td>Array</td>
<td>Optional. Array of strings containing the OIDs of SNMP variables to retrieve from the agent. Overrides values specified in the SNMP data type.</td>
</tr>
</tbody>
</table>
Table 95. SNMPGetAction function parameters (continued)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Community</td>
<td>String</td>
<td>Optional. Name of the SNMP write community string. Default is public.</td>
</tr>
<tr>
<td>Timeout</td>
<td>Integer</td>
<td>Optional. Number of seconds to wait for a response from the SNMP agent.</td>
</tr>
<tr>
<td>Version</td>
<td>Integer</td>
<td>Optional. SNMP version number. Possible values are 1, 2, and 3. Default is 1.</td>
</tr>
<tr>
<td>UserId</td>
<td>String</td>
<td>Required for SNMP v3 authentication. If using SNMP v1 or v2, or using v3 without authentication, pass a null value for this parameter.</td>
</tr>
<tr>
<td>AuthProtocol</td>
<td>String</td>
<td>Optional. For use with SNMP v3 authentication only. Possible values are. MD5, MD5_AUTH, NO_AUTH, SHA, SHA_AUTH. NO_AUTH is the default.</td>
</tr>
<tr>
<td>AuthPassword</td>
<td>String</td>
<td>Optional. For use with SNMP v3 authentication only. Authentication password associated with the specified SNMP User ID.</td>
</tr>
<tr>
<td>PrivProtocol</td>
<td>String</td>
<td>Optional. Privacy policy to be used with this function. Possible values are. DES, AES, None. None is the default.</td>
</tr>
<tr>
<td>PrivPassword</td>
<td>String</td>
<td>Optional. For use with SNMP v3 authentication only. Privacy password associated with the specified SNMP User ID.</td>
</tr>
<tr>
<td>ContextId</td>
<td>String</td>
<td>Optional. For use with SNMP v3 authentication only. Authentication context ID.</td>
</tr>
<tr>
<td>ContextName</td>
<td>String</td>
<td>Optional. For use with SNMP v3 authentication only. Authentication context name.</td>
</tr>
</tbody>
</table>

**Return Values**

When you call SNMPGetAction, it sets the following variables in the policy context: ValueList.

The ValueList variable is an array of strings, each of which stores the value of one variable retrieved from the SNMP agent. The strings in the array are assigned in the order that the variable OIDs are specified in the SNMP data type or the VarIdList parameter.

**Error handling**

If the SNMP operation fails, the Impact policy engine throws an SnmpDSAException. You can handle this exception using the Handle() function:

```java
// Handle SNMP OSA Exceptions
Handle com.micromuse.dsa.snmpdsa.SnmpDSAException {
  log("ErrorMessage: " + ErrorMessage);
  javaCall("com.micromuse.dsa.snmpdsa.SnmpDSAException", ExceptionMessage, "getCause", null);
  log("MyException is " + ExceptionMessage);
}
```

**Example 1**

The following example shows how to retrieve a set of SNMP variables by calling SNMPGetAction and passing the name of an SNMP data type as an input parameter.
In this example, the SNMP data type is named SNMP_PACKED. The data type configuration specifies the host name and port where the SNMP agent is running and the OIDs of the variables you want to retrieve.

// Call SNMPGetAction and pass the name of the SNMP data type that contains // configuration information required to perform the SNMP GET

TypeName = "SNMP_PACKED";

SnmpGetAction(TypeName, "192.168.1.1", 161, null, null, null, null, null, null, null, null, null, null);  

// Print the results of the SNMP GET to the policy log

Count = 0;

while (Count < Length(ValueList)) {
    Log(ValueList[Count]);
    Count = Count + 1;
}

Example 2

The following example shows how to retrieve a set of SNMP variables by calling SNMPGetAction and explicitly overriding the default host name, port, and other configuration values set in the SNMP data type.

Example 2 using IPL.

// Call SnmpGetAction and pass the name of the SNMP data type that contains // configuration information required to perform the SNMP GET

TypeName = "SNMP_PACKED";
HostId = "192.168.1.1";
Port = 161;
VarIdList = {"1.3.6.1.2.1.1.5.0", "1.3.6.1.2.1.1.6.0"};
Community = "private";
Timeout = 15;

SnmpGetAction(TypeName, HostId, Port, VarIdList, Community, Timeout, null, null, null, null, null, null,null);

// Print the results of the SNMP GET to the policy log

Count = 0;

while (Count < Length(ValueList)) {
    Log(ValueList[Count]);
    Count = Count + 1;
}

Example 2 using JavaScript.

// Call SnmpGetAction and pass the name of the SNMP data type that contains // configuration information required to perform the SNMP GET

TypeName = "SNMP_PACKED";
HostId = "192.168.1.1";
Port = 161;
VarIdList = ["1.3.6.1.2.1.1.5.0", "1.3.6.1.2.1.1.6.0"];  
Community = "private";
Timeout = 15;
SnmpGetAction(TypeName, HostId, Port, VarIdList, Community, Timeout, null, null, null, null, null, null,null);

// Print the results of the SNMP GET to the policy log

Count = 0;
while (Count < Length(ValueList)) {
    Log(ValueList[Count]);
    Count = Count + 1;
}

Example 3

The following example shows how to retrieve a set of SNMP variables using SNMP v3 authentication.

Example 3 using IPL.

// Call SnmpGetAction and pass the name of the SNMP data type that contains
// configuration information required to perform the SNMP GET

TypeName = "SNMP_PACKED";
HostId = "192.168.1.1";
Port = 161;
VarIdList = ["1.3.6.1.2.1.1.5.0", "1.3.6.1.2.1.1.6.0"]; Community = "private";
Timeout = 15;
Version = 3;
UserId = "snmpusr";
AuthProtocol = "MD5_AUTH";
AuthPassword = "snmppwd";
ContextId = "ctx";

SnmpGetAction(TypeName, HostId, Port, VarIdList, Community, 
        Timeout, Version, UserId, AuthProtocol, AuthPassword, null, ContextId, null);

// Print the results of the SNMP GET to the policy log

Count = 0;
while (Count < Length(ValueList)) {
    Log(ValueList[Count]);
    Count = Count + 1;
}

Example 3 using JavaScript.

// Call SnmpGetAction and pass the name of the SNMP data type that contains
// configuration information required to perform the SNMP GET

TypeName = "SNMP_PACKED";
HostId = "192.168.1.1";
Port = 161;
VarIdList = ["1.3.6.1.2.1.1.5.0", "1.3.6.1.2.1.1.6.0"]; Community = "private";
Timeout = 15;
Version = 3;
UserId = "snmpusr";
AuthProtocol = "MD5_AUTH";
AuthPassword = "snmppwd";
ContextId = "ctx";

SnmpGetAction(TypeName, HostId, Port, VarIdList, Community, 
        Timeout, Version, UserId, AuthProtocol, AuthPassword, null, ContextId, null);

// Print the results of the SNMP GET to the policy log

Count = 0;
while (Count < Length(ValueList)) {
    Log(ValueList[Count]);
    Count = Count + 1;
}
**SNMPGetNextAction**

The SnmpGetNextAction function retrieves the next SNMP variables in the variable tree from the specified agent.

It stores the resulting OIDs in a variable named VarIdList, the resulting values in a variable named ValueList. The function sends a series of SNMP GETNEXT commands to the specified agent where each command specifies a single OID for which the next variable in the tree is to be retrieved.

When you call SnmpGetNextAction, you pass an SNMP data type and, for SNMP v3, any authorization parameters that are required. To override the agent and variable information specified in the SNMP data type, you can also optionally pass a host name, a port number, a list of OIDs, and other information needed to retrieve the data.

**Syntax**

The following is the syntax for SnmpGetNextAction:

```c
SnmpGetNextAction(TypeName, [HostId], [Port], [VarIdList], [Community],
                    [Timeout], [Version], [UserId], [AuthProtocol], [AuthPassword],
                    [PrivPassword], [ContextId], [ContextName])
```

**Parameters**

The SnmpGetNextAction function has the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TypeName</td>
<td>String</td>
<td>Name of the SNMP data type that specifies the host name, port, OIDs, and other information needed to retrieve the SNMP data.</td>
</tr>
<tr>
<td>HostId</td>
<td>String</td>
<td>Optional. Host name or IP address of the system where the SNMP agent is running. Overrides value specified in the SNMP data type.</td>
</tr>
<tr>
<td>Port</td>
<td>Integer</td>
<td>Optional. Port where the SNMP agent is running. Overrides value specified in the SNMP data type.</td>
</tr>
<tr>
<td>VarIdList</td>
<td>Array</td>
<td>Optional. Array of strings containing the OIDs of SNMP variables to retrieve from the agent. Overrides values specified in the SNMP data type.</td>
</tr>
<tr>
<td>Community</td>
<td>String</td>
<td>Optional. Name of the SNMP write community string. Default is public.</td>
</tr>
<tr>
<td>Timeout</td>
<td>Integer</td>
<td>Optional. Number of seconds to wait for a response from the SNMP agent before timing out.</td>
</tr>
<tr>
<td>Version</td>
<td>Integer</td>
<td>Optional. SNMP version number. Possible values are 1, 2 and 3. Default is 1.</td>
</tr>
<tr>
<td>UserId</td>
<td>String</td>
<td>Required for SNMP v3 authentication. If using SNMP v1 or v2, or v3 without authentication, pass a null value for this parameter.</td>
</tr>
<tr>
<td>AuthProtocol</td>
<td>String</td>
<td>Optional. For use with SNMP v3 authentication only. Possible values are. MD5, MD5_AUTH, NO_AUTH, SHA, SHA_AUTH. NO_AUTH is the default.</td>
</tr>
</tbody>
</table>
Table 96. SnmpGetNextAction function parameters (continued)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AuthPassword</td>
<td>String</td>
<td>Optional. For use with SNMP v3 authentication only. Authentication password associated with the specified SNMP User ID.</td>
</tr>
<tr>
<td>PrivProtocol</td>
<td>String</td>
<td>Optional. Privacy policy to be used with this function. Possible values are: DES, AES, None. None is the default. Note: This parameter does not form a part of the function call. It must be defined before the call to the function.</td>
</tr>
<tr>
<td>PrivPassword</td>
<td>String</td>
<td>Optional. For use with SNMP v3 authentication only. Privacy password associated with the specified SNMP User ID.</td>
</tr>
<tr>
<td>ContextId</td>
<td>String</td>
<td>Optional. For use with SNMP v3 authentication only. Authentication context ID.</td>
</tr>
<tr>
<td>ContextName</td>
<td>String</td>
<td>Optional. For use with SNMP v3 authentication only. Authentication context name.</td>
</tr>
</tbody>
</table>

**Error handling**

If the SNMP operation fails, the Impact policy engine throws an SnmpDSAException. You can handle this exception using the Handle() function:

```java
// Handle SNMP DSA Exceptions
Handle com.micromuse.dsa.snmpdsa.SnmpDSAException {
    log("ErrorMessage: " + ErrorMessage);
    javaCall("com.micromuse.dsa.snmpdsa.SnmpDSAException", ExceptionMessage, "getCause", null);
    log("MyException is " + ExceptionMessage);
}
```

**Example 1**

The following example shows how to retrieve SNMP variables in the variable tree by calling SnmpGetNextAction and passing the name of an SNMP data type as an input parameter. In this example, the SNMP data type is named SNMP_PACKED. The data type configuration specifies the host name and port where the SNMP agent is running and the OIDs of the variables whose subsequent values in the tree you want to retrieve.

```java
// Call SnmpGetNextAction and pass the name of the SNMP
data type that contains configuration information required
to perform the SNMP GETNEXT

TypeName = "SNMP_PACKED";

SnmpGetNextAction(TypeName, "192.168.1.1", 161, null, null, null, null, null, null, null, null, null);

// Print the results of the SNMP GETNEXT to the policy log

Count = 0;

while (Count < Length(ValueList)) {
    Log(VarIdList[Count] + ": " + ValueList[Count]);
    Count = Count + 1;
}
```
Example 2

The following example shows how to retrieve SNMP variables in the variable tree by calling SNMPGetNextAction and explicitly overriding the default host name, port, and other configuration values set in the SNMP data type.

Example 2 using IPL.

// Call SnmpGetNextAction and pass the name of the // SNMP data type that contains configuration information // required to perform the SNMP GETNEXT

TypeName = "SNMP_PACKED";
HostId = "192.168.1.1";
Port = 161;
VarIdList = {".1.3.6.1.2.1.1.5.0", ".1.3.6.1.2.1.1.6.0"};
Community = "private";
Timeout = 15;

SnmpGetNextAction(TypeName, HostId, Port, VarIdList, Community, 
Timeout, null, null, null, null, null, null);

// Print the results of the SNMP GETNEXT to the policy log

Count = 0;
while (Count < Length(ValueList)) {
    Log(VarIdList[Count] + ": " + ValueList[Count]);
    Count = Count + 1;
}

Example 2 using JavaScript.

// Call SnmpGetNextAction and pass the name of the // SNMP data type that contains configuration information // required to perform the SNMP GETNEXT

TypeName = "ipRouteTable";
HostId = "localhost";
Port = 161;
VarIdList = [".1.3.6.1.2.1.1.5.0", ".1.3.6.1.2.1.1.6.0"];
Community = "public";
Timeout = 15;
SnmpGetNextAction(TypeName, HostId, Port, VarIdList, Community, Timeout, null, null, null, null, null, null);

// Print the results of the SNMP GETNEXT to the policy log

Count = 0;
while (Count < Length(ValueList)) {
    Log(VarIdList[Count] + ": " + ValueList[Count]);
    Count = Count + 1;
}

Example 3

The following example shows how to retrieve subsequent SNMP variables in the variable tree using SNMP v3 authentication.

Example 3 using IPL.

// Call SnmpGetNextAction and pass the name of the // SNMP data type that contains configuration information // required to perform the SNMP GETNEXT

TypeName = "SNMP_PACKED";
HostId = "192.168.1.1";
Port = 161;
VarIdList = {".1.3.6.1.2.1.1.5.0", ".1.3.6.1.2.1.1.6.0"};
```javascript
Community = "private";
Timeout = 15;
Version = 3;
UserId = "snmpusr";
AuthProtocol = "MD5_AUTH";
AuthPassword = "snmppwd";
ContextId = "ctx";

SnmpGetNextAction(TypeName, HostId, Port, VarIdList, Community,
   Timeout, Version, UserId, AuthProtocol, AuthPassword, null,
   ContextId, null);

// Print the results of the SNMP GET to the policy log
Count = 0;
while (Count < Length(ValueList)) {
   Log(VarIdList[Count] + "=" + ValueList[Count]);
   Count = Count + 1;
}
```

Example 3 using JavaScript.

```javascript
// Call SnmpGetNextAction and pass the name of the
// SNMP data type that contains configuration information
// required to perform the SNMP GETNEXT
TypeName = "ipRouteTable";
HostId = "localhost";
Port = 161;
VarIdList = [".1.3.6.1.2.1.1.5.0", ".1.3.6.1.2.1.1.6.0"];  
Community = "public";
Timeout = 15;
Version = 3;
UserId = "snmpuser";
AuthProtocol = "MD5";
AuthPassword = "snmppwd";
PrivPassword = "privpwd";

SnmpGetNextAction(TypeName, HostId, Port, VarIdList, Community, Timeout, Version, UserId, AuthProtocol, AuthPassword, PrivPassword, null, null);

// Print the results of the SNMP GET to the policy log
Count = 0;
while (Count < Length(ValueList)) {
   Log(VarIdList[Count] + "=" + ValueList[Count]);
   Count = Count + 1;
}
```

---

**SNMPSetAction**

The SNMPSetAction function sets variable values on the specified SNMP agent.

This function operates by sending an SNMP SET command to the specified agent.

When you call SNMPSetAction, you pass an SNMP data type, the host name, and port of the agent, an array of OIDs, and the array of values that you want to set. If you are using SNMP v3, you can also include information required to authenticate as an SNMP user.

**Syntax**

The following is the syntax for SNMPSetAction:

```javascript
SnmpSetAction(TypeName, [HostId], [Port], [VarIdList],
ValueList, [Community], [Timeout], [Version], [UserId], [AuthProtocol],
[AuthPassword], [PrivPassword], [ContextId], [ContextName])
```
Parameters

The SNMPSetAction function has the following parameters.

Table 97. SNMPSetAction function parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TypeName</td>
<td>String</td>
<td>Name of the SNMP data type that specifies the host name, port, OIDs, and other information needed to set the SNMP data.</td>
</tr>
<tr>
<td>HostId</td>
<td>String</td>
<td>Optional. Host name or IP address of the system where the SNMP agent is running. Overrides value specified in the SNMP data type.</td>
</tr>
<tr>
<td>Port</td>
<td>Integer</td>
<td>Optional. Port where the SNMP agent is running. Overrides value specified in the SNMP data type.</td>
</tr>
<tr>
<td>VarIdList</td>
<td>Array</td>
<td>Array of strings containing the OIDs of SNMP variables to set on the agent. Overrides values specified in the SNMP data type.</td>
</tr>
<tr>
<td>ValueList</td>
<td>Array</td>
<td>Array of strings containing the values you want to set. You must specify these values in the same order that the OIDs appear either in the SNMP data type or in the VarIdList variable.</td>
</tr>
<tr>
<td>Community</td>
<td>String</td>
<td>Optional. Name of the SNMP write community string. Default is public.</td>
</tr>
<tr>
<td>Timeout</td>
<td>Integer</td>
<td>Optional. Number of seconds to wait for a response from the SNMP agent before timing out.</td>
</tr>
<tr>
<td>Version</td>
<td>Integer</td>
<td>Optional. SNMP version number. Possible values are 1, 2 and 3. Default is 1.</td>
</tr>
<tr>
<td>UserId</td>
<td>String</td>
<td>Required for SNMP v3 authentication. If using SNMP v1 or v2, or using v3 without authentication, pass a null value for this parameter.</td>
</tr>
<tr>
<td>AuthProtocol</td>
<td>String</td>
<td>Optional. For use with SNMP v3 authentication only. Possible values are. MD5, MD5_AUTH, NO_AUTH, SHA, SHA_AUTH. NO_AUTH is the default.</td>
</tr>
<tr>
<td>AuthPassword</td>
<td>String</td>
<td>Optional. For use with SNMP v3 authentication only. Authentication password associated with the specified SNMP User ID.</td>
</tr>
<tr>
<td>PrivProtocol</td>
<td>String</td>
<td>Optional. Privacy policy to be used with this function. Possible values are. DES, AES, None. None is the default. Note: This parameter does not form a part of the function call. It must be defined before the call to the function.</td>
</tr>
<tr>
<td>PrivPassword</td>
<td>String</td>
<td>Optional. For use with SNMP v3 authentication only. Privacy password associated with the specified SNMP User ID.</td>
</tr>
<tr>
<td>ContextId</td>
<td>String</td>
<td>Optional. For use with SNMP v3 authentication only. Authentication context ID.</td>
</tr>
<tr>
<td>ContextName</td>
<td>String</td>
<td>Optional. For use with SNMP v3 authentication only. Authentication context name.</td>
</tr>
</tbody>
</table>
Error handling

If the SNMP operation fails, the Impact policy engine throws an SnmpDSAException. You can handle this exception using the Handle() function:

```java
// Handle SNMP DSA Exceptions
Handle com.micromuse.dsa.snmpdsa.SnmpDSAException {
    log("ErrorMessage: " + ErrorMessage);
    javaCall("com.micromuse.dsa.snmpdsa.SnmpDSAException", ExceptionMessage, "getCause", null);
    log("MyException is " + ExceptionMessage);
}
```

Example 1

The following example shows how to set SNMP variables by calling SNMPSetAction and passing the name of an SNMP data type, an array of OIDs, and an array of values as input parameters. In this example, the SNMP data type is named SNMP_PACKED.

Example 1 using IPL.

```java
// Call SnmpSetAction and pass the name of the SNMP data type that contains configuration information required to perform the SNMP SET
TypeName = "SNMP_PACKED";
HostId = "192.168.1.1";
Port = 161;
VarIdList = {"1.3.6.1.2.1.1.4.0", "1.3.6.1.2.1.1.5.0"};
ValueList = {"Value_01", "Value_02"};
SnmpSetAction(TypeName, HostId, Port, VarIdList, ValueList, null, null, null, null, null, null, null, null, null, null);
```

Example 1 using JavaScript.

```java
// Call SnmpSetAction and pass the name of the SNMP data type that contains configuration information required to perform the SNMP SET
TypeName = "SNMP_PACKED";
HostId = "192.168.1.1";
Port = 161;
VarIdList = ["1.3.6.1.2.1.1.4.0", "1.3.6.1.2.1.1.5.0"];
ValueList = ["Value_01", "Value_02"];
SnmpSetAction(TypeName, HostId, Port, VarIdList, ValueList, null, null, null, null, null, null, null, null, null, null);
```

Example 2

The following example shows how to set SNMP variables using SNMP v3 authentication.

Example 2 using IPL.

```java
// Call SnmpSetAction and pass the name of the SNMP data type that contains configuration information required to perform the SNMP SET
TypeName = "SNMP_PACKED";
HostId = "192.168.1.1";
Port = 161;
VarIdList = {"1.3.6.1.2.1.1.4.0", "1.3.6.1.2.1.1.5.0"};
ValueList = {"Value_01", "Value_02"};
Community = "private";
Timeout = 15;
Version = 3;
```
userId = "snmpusr";
authProtocol = "MD5_AUTH";
authPassword = "snmppwd";
contextId = "ctx";

snmpSetAction(typeName, hostId, port, varIdList, valueList, community, timeout, version, userId, authProtocol, authPassword, null, contextId, null);

Example 2 using JavaScript.

// Call SnmpSetAction and pass the name of the
// SNMP data type that contains configuration information
// required to perform the SNMP SET
typeName = "SNMP_PACKED";
hostId = "192.168.1.1";
port = 161;
varIdList = ["1.3.6.1.2.1.1.4.0", "1.3.6.1.2.1.1.5.0"];
valueList = ["Value_01", "Value_02"];    
community = "private";
timeout = 15;
version = 3;
userId = "snmpusr";
authProtocol = "MD5_AUTH";
authPassword = "snmppwd";
contextId = "ctx";

SnmpSetAction(typeName, hostId, port, varIdList, valueList, community, timeout, version, userId, authProtocol, authPassword, null, contextId, null);

SnmpTrapAction

The SnmpTrapAction function sends a trap (for SNMP v1) or a notification (for SNMP v2) to an SNMP manager. Sending traps or notifications is not supported for SNMP v3.

Syntax

The following is the syntax for SnmpTrapAction:

SnmpTrapAction(hostId, port, [varIdList], [valueList],
    [community], [timeout], [version], [sysUpTime], [enterprise],
    [genericTrap], [specificTrap], [snmpTrapOid])

Parameters

The SnmpTrapAction function has the following parameters.

Table 98. SnmpTrapAction function parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hostId</td>
<td>String</td>
<td>Host name or IP address of the system where the SNMP manager is running.</td>
</tr>
<tr>
<td>port</td>
<td>Integer</td>
<td>Port where the SNMP manager is running.</td>
</tr>
<tr>
<td>varIdList</td>
<td>Array</td>
<td>Optional. Array of strings containing the OIDs of SNMP variables to send to the manager.</td>
</tr>
<tr>
<td>valueList</td>
<td>Array</td>
<td>Optional. Array of strings containing the values you want to send to the manager. You must specify these values in the same order that the OIDs appear in the varIdList variable.</td>
</tr>
</tbody>
</table>
Table 98. SnmpTrapAction function parameters (continued)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Community</td>
<td>String</td>
<td>Optional. Name of the SNMP write community string. Default is public.</td>
</tr>
<tr>
<td>Timeout</td>
<td>Integer</td>
<td>Optional. Number of seconds to wait for a response from the SNMP agent before timing out.</td>
</tr>
<tr>
<td>Version</td>
<td>Integer</td>
<td>Optional. SNMP version number. Possible values are 1 and 2. Default is 1.</td>
</tr>
<tr>
<td>SysUpTime</td>
<td>Integer</td>
<td>Optional. Number of milliseconds since the system started. Default is the current system time in milliseconds.</td>
</tr>
<tr>
<td>Enterprise</td>
<td>String</td>
<td>Required for SNMP v1 only. Enterprise ID.</td>
</tr>
<tr>
<td>GenericTrap</td>
<td>String</td>
<td>Required for SNMP v1 only. Generic trap ID.</td>
</tr>
<tr>
<td>SpecificTrap</td>
<td>String</td>
<td>Required for SNMP v1 only. Specific trap ID.</td>
</tr>
<tr>
<td>SnmpTrapOid</td>
<td>String</td>
<td>Optional for SNMP v1. Required for SNMP v2. SNMP trap OID.</td>
</tr>
</tbody>
</table>

Example 1

The following example shows how to send an SNMP v1 trap to a manager using SnmpTrapAction.

```c
// Call SnmpTrapAction

HostId = "localhost";
Port = 162;
Version = 1;
Community = "public";
SysUpTime = 1001;
Enterprise = ".1.3.6.1.2.1.11";
GenericTrap = 3;
SpecificTrap = 0;
VarIdList = {"1.3.6.1.2.1.2.2.1.1.0", "sysDescr"};
ValueList = {"2", "My system"};
SnmpTrapAction(HostId, Port, VarIdList, ValueList,
              Community, 15, Version, SysUpTime, Enterprise, GenericTrap,
              SpecificTrap, NULL);
```

Example 2

The following example shows how to send an SNMP v2 notification to a manager using SnmpTrapAction. SNMP v2 requires that you specify an SNMP trap OID when you call this function.

```c
// Call SnmpTrapAction

HostId = "localhost";
Port = 162;
Version = 1;
Community = "public";
SysUpTime = 1001;
Enterprise = ".1.3.6.1.2.1.11";
GenericTrap = 3;
SpecificTrap = 0;
VarIdList = {"1.3.6.1.2.1.2.2.1.1.0", "sysDescr"};
ValueList = {"2", "My system"};
SnmpTrapOid = ".1.3.6.1.2.4.1.11";
```
SnmpTrapAction(HostId, Port, VarIdList, ValueList, Community, 15, Version, SysUpTime, Enterprise, GenericTrap, SpecificTrap, SnmpTrapOid);

**Split**

The Split function returns an array of substrings from a string by using the specified delimiters.

**Syntax**

The Split function has the following syntax:

\[
Array = \text{Split}(Expression, \text{Delimiters})
\]

**Parameters**

The Split function has the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>String</td>
<td>String to split into substrings</td>
</tr>
<tr>
<td>Delimiters</td>
<td>String</td>
<td>String that contains the delimiter characters.</td>
</tr>
</tbody>
</table>

**Return value**

Array of substrings.

**Example**

The following example shows how to split a string into an array of substrings.

```plaintext
MyString = "This is a test."
Delimiters = " ";
MyArray = Split(MyString, Delimiters);
Log(MyArray);
```

This example prints the following message to the policy log:

Parser Log: {This, is, a, test.}

**String**

The String function converts an integer, float, or boolean expression to a string.

**Syntax**

The String function has the following syntax:

\[
String = \text{String}(Expression)
\]
Parameters

The String function has the following parameters.

Table 100. String function parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>Integer</td>
<td>Float</td>
</tr>
</tbody>
</table>

Return value

Converted string value.

Example

The following example shows how to convert integers, floats, and boolean expressions to a string.

MyString = String(123);
Log(MyString);

MyString = String(123.54);
Log(MyString);

MyString = String(true);
Log(MyString);

This example prints the following message to the policy log for IPL:

Parser Log: 123
Parser Log: 123.54
Parser Log: true

This example prints the following message to the policy log for JavaScript.

JavaScript treats numbers as doubles, as a result, numbers display using decimals:

Parser Log: 123.0
Parser Log: 123.54
Parser Log: true

Strip

The Strip function strips all instances of the specified substring from a string.

The order in which you supply the characters is not significant.

Syntax

The Strip function has the following syntax:

\[ String = Strip(\text{Expression}, \text{Characters}) \]

Parameters

The Strip function has the following parameters.

Table 101. Strip function parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>String</td>
<td>String to strip.</td>
</tr>
</tbody>
</table>
Table 101. Strip function parameters (continued)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Characters</td>
<td>String</td>
<td>String that contains characters to strip from the string.</td>
</tr>
</tbody>
</table>

**Return value**

The string with the specified characters stripped out.

**Example**

The following example shows how to strip a characters from a string.

```plaintext
MyString = "abccababc."
MyCharacters = "ab"
MyStrip = Strip(MyString, MyCharacters)
Log(MyStrip);
```

This example prints the following message to the policy log:

Parser Log: ccc.

**Substring**

The Substring function returns a substring from a specified string by using index positions.

Index positions start at 0.

**Syntax**

The Substring function has the following syntax:

```plaintext
String = Substring(Expression, Start, End)
```

**Parameters**

The Substring function has the following parameters.

Table 102. Substring function parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>String</td>
<td>String to search for the substring.</td>
</tr>
<tr>
<td>Start</td>
<td>Integer</td>
<td>Starting character position of the substring.</td>
</tr>
<tr>
<td>End</td>
<td>Integer</td>
<td>Ending character position of the substring, plus one.</td>
</tr>
</tbody>
</table>

**Return value**

The substring returned by index positions.

**Example**

The following example shows how to return a substring using index positions.

```plaintext
MyString = "This is a test."
MySubstring = Substring(MyString, 0, 4)
Log(MySubstring);
```
This example prints the following message to the policy log:
Parser Log: This

**Synchronized**

Use the Synchronized function to write thread-safe policies for use with a multi-threaded event processor by using IPL or JavaScript.

**Synchronized for IPL**

The syntax and parameters for the Synchronized function differ for IPL and JavaScript. Refer to the information in the following sections for the details.

The Synchronized function has the following syntax for IPL:

```plaintext
synchronized(identifier) { statements }
```

**Parameters**

The Synchronized function has the following parameters for IPL.

*Table 103. Synchronized function parameters for IPL*

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>identifier</td>
<td>String</td>
<td>The unique name for the statement block.</td>
</tr>
<tr>
<td>statements</td>
<td>String</td>
<td>Programming statements.</td>
</tr>
</tbody>
</table>

**Example**

The following example shows how to insert an item into a custom data type using IPL.

```plaintext
insert_table = NewObject();
synchronized (insert_table) {
  MyContext = NewObject();
  MyContext.Cust_ID = 5;
  MyContext.Cust_Loc = "New York";
  AddDataItem("Cust", MyContext);
}
```

**Synchronized for JavaScript**

The Synchronized function has the following syntax for JavaScript:

```plaintext
synchronized(<function name>, <identifier> );
```

**Parameters**

The Synchronized function has the following parameters for JavaScript.

*Table 104. Synchronized function parameters for JavaScript*

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;function name&gt;</td>
<td>String</td>
<td>The name for the function.</td>
</tr>
<tr>
<td>&lt;identifier&gt;</td>
<td>String</td>
<td>The unique name for the statement block.</td>
</tr>
</tbody>
</table>
Example

The following example shows how to insert an item into a custom data type using JavaScript.

```javascript
insert_table = NewObject();

function mySyncFunc() {
    MyContext = NewObject();
    MyContext.Cust_ID = 5;
    MyContext.Cust_Loc = "New York";
    AddDataItem("Cust", MyContext);
}

Synchronized(mySyncFunc, insert_table);
```

ToLower

The ToLower function converts a string to lowercase characters.

Syntax

The ToLower function has the following syntax:

```javascript
String = ToLower(Expression)
```

Parameters

The ToLower function has the following parameter.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>String</td>
<td>String to convert to lower case.</td>
</tr>
</tbody>
</table>

Return value

The lower case string.

Example

The following example shows how to convert a string to lower case.

```javascript
MyString = "tHiS iS a TeSt.";
MyLower = ToLower(MyString);
Log(MyLower);
```

This example prints the following message to the policy log:

Parser Log: this is a test.

ToUpper

The ToUpper function converts a string to uppercase characters.

Syntax

The ToUpper function has the following syntax:

```javascript
String = ToUpper(Expression)
```
Parameters

The ToUpper function has the following parameter.

*Table 106. ToUpper function parameters*

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>String</td>
<td>String to convert to upper case.</td>
</tr>
</tbody>
</table>

Return value

The upper case string.

Example

The following example shows how to convert a string to upper case.

```java
MyString = "tHiS iS a TeSt.";
MyUpper = ToUpper(MyString);
Log(MyUpper);
```

This example prints the following message to the policy log:
Parser Log: THIS IS A TEST.

Trim

The Trim function trims leading and trailing white space from a string.

Syntax

The Trim function has the following syntax:

```
String = Trim(Expression)
```

Parameters

The Trim function has the following parameter.

*Table 107. Trim function parameters*

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>String</td>
<td>String that you want to trim.</td>
</tr>
</tbody>
</table>

Return value

The string with white space trimmed.

Example

The following example shows how to trim white space from a string.

```java
MyString = " This is a test. ";
MyTrim = Trim(MyString);
Log(MyTrim);
```

This example prints the following message to the policy log:
Parser Log: This is a test.
TBSM functions

An overview of functions that are used specifically with TBSM.

Netcool/Impact has the three Tivoli Business Service Manager-specific policy functions.

- **PassToTBSM**: Used to send events from Netcool/Impact to Tivoli Business Service Manager. In an Netcool/Impact policy, you can add the PassToTBSM function to the policy. When you activate the policy using an Netcool/Impact service, the event information is sent to TBSM.

- **RemoteTBSMShell**: Used to send RadShell commands from a policy using a remote Impact Server. The Impact Server and the Tivoli Business Service Manager server must be configured for Name Server clustering. The clustered Name Server must contain clusters of the Impact Server and the Tivoli Business Service Manager server.

- **TBSMShell**: Used to add RadShell commands to a policy. This function is Tivoli Business Service Manager-specific and is available only in a Tivoli Business Service Manager installation.

**PassToTBSM**

Use the PassToTBSM function to send event information from Netcool/Impact to TBSM. In a Netcool/Impact policy, you can invoke the PassToTBSM function which gets read as an event by Tivoli Business Service Manager.

PassToTBSM takes one argument, EventContainer. The EventContainer has a set of member variables that correspond to fields in the incoming event.

**Examples**

An example of a JMSSend policy that sends messages to a JMS data source using the SendJMSMessage function.

```
// Set JMSDataSource to a valid and existing JMSDataSource in Impact.
// The destination where the message is sent is obtained from the JMSDataSource.

Log("\nSetting up props before calling SendJMSMessage action function");
JMSDataSource = 'myJMS_ds';

// Create a message properties object and populate its
// member variables with message header properties and custom properties

MsgProps = NewObject();
MsgProps.TimeToLive = 0;
MsgProps.Priority = 9;
MsgProps.Expiration = 2000;
MsgProps.DeliveryMode = "PERSISTENT";
MsgProps.ReplyTo="queue2";

// Create a map message content and populate its member
// variables where each variable and value represent a name/
// value pair for the resulting map

MsgMapBody = NewObject();
MsgMapBody.branch = "ATM1";
MsgMapBody.status = "Marginal";

// Call SendJMSMessage and pass the JNDI properties
// context, the message properties context, the message
// map context and other parameters
```
Calling SendJMSMessage action function now; SendJMSMessage(JMDSDataSource, MsgProps, MsgMapBody);

log("Received a Message : " + currentContext());
log("Message : " + EventContainer);

An example of a JMSReceive policy run by the JMSMessageListener service that assigns fields to the received message and passes the information to Tivoli Business Service Manager:

```java
// branch = @JMSMessage.branch;
// status = @JMSMessage.status;

e=NewEvent();
e.branch = @JMSMessage.branch;
e.status = @JMSMessage.status;

Log(" Branch : " + e.branch + " Status : " + e.status);
PassToTBSM(e);
```

**RemoteTBSMShell**

A stand-alone implementation of Netcool/Impact can run RADShell commands from a policy in Tivoli Business Service Manager.

The stand-alone Impact Server and the Tivoli Business Service Manager server must be configured to use Name Server clustering. The clustered Name Server must contain the clusters of the Impact Server and the Tivoli Business Service Manager server. The RemoteTBSMShell command can be run from any policy using the following syntax:

**Syntax**

```java
Result = RemoteTBSMShell('<command>');
```

Where `command` is the radshell command to run and `Result` is the string returned by the radshell command.

This example creates a template called DBFarm.

```java
Result = RemoteTBSMShell ('createTemplate ("DBFarm","man_svg.gif");
```

**Known issue**

The Policy editor **Insert Function** does not have TBSM functions in Impact for PassToTBSM and RemoteTBSMShell.

**UpdateDB**

The UpdateDB function updates any supported SQL database by using a data type or data source.

**Syntax**

You can use this function to update a database directly with any existing objects. You no longer have to pass the objects to DirectSQL or BatchUpdate, and loop through the objects and form the SQL statement that uses `where` and set the key value pairs.

The UpdateDB function has the following syntax.
UpdateDB(DataType |DataSource, WHERE Clause, Fields To Set);

Parameters

The UpdateDB function has the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>String</td>
<td>Data type name or data source</td>
</tr>
<tr>
<td>Impact Object</td>
<td>Key-Value pairs</td>
<td>The pairs are used to form the WHERE clause statement. WHERE key1=value1 AND key2=value2 AND ... keyn=valuen</td>
</tr>
<tr>
<td>Impact Object</td>
<td>Field-Value pairs</td>
<td>The fields to be updated in the SET statement. SET field1=value, field2=value2, ... fieldn=valuen</td>
</tr>
</tbody>
</table>

Examples

Remember:

- If data type is used, the function uses the table schema to process the update statement and get the fields data types and properly format the values.
- To use the data source, Object.UseDataSource = true; must be present in Key-Value pairs object.
- If an Object Server data source is used the default table is alerts.status. To use any other database or to use a different table for Object Server, the following values must be present in key-value pairs as first element. KeysValues.DataSource_TableName="tableName";
- To debug, turn on the trace and SQL statement in Policy Logger to see the query.

A policy example that uses a data type.

```java
//Using Data Type
Type = "CHANGE_TO_DATATYPE_NAME";
KeysValues = NewObject();
KeysValues.Identifier='hostname3';
FieldsValues = NewObject();
FieldsValues.Node='Updated by New Function';

Result = UpdateDB(Type, KeysValues, FieldsValues);
Log("Result : " + Result);
```

A policy example that uses a data source.

```java
Log("Using data source");
KeysValues = NewObject();
KeysValues.Identifier="hostname4";
KeysValues.UseDataSource=true;
FieldsValues=NewObject();
FieldsValues.Node="Updated by New Function Using data source";
Log("CurrentContext: " + CurrentContext());
Result = UpdateDB('defaultobjectserver',KeysValues,FieldsValues);
Log("Result : " + Result);
```
UpdateEventQueue

The UpdateEventQueue function updates or deletes events in the event reader event queue.

Use UpdateEventQueue for situations in which a policy modifies an incoming event that is expected to have other related events in the event queue at the same time.

**Note:** The UpdateEventQueue does not access events in the EventProcessor queue. It works only with the events in the EventReader queue. If you update or delete events in the EventReader queue, it modifies only the event containers within Netcool/Impact and does not affect the events in Netcool/OMNIbus. To modify events in Netcool/OMNIbus, you need to use the ReturnEvent function.

To update events, you call UpdateEventQueue and pass the name of the event reader, a filter string, and an update expression as input values. The filter string specifies which events to update. It uses the SQL filter syntax, which is similar to the syntax of the WHERE clause in an SQL SELECT statement. The update expression is a comma-separated list of field assignments similar to the contents of the SET clause in an SQL UPDATE statement. For more information about SQL filters, see "SQL filters" on page 79.

To delete events, you call UpdateEventQueue and pass the name of the event reader, a filter string, and a boolean value that indicates that you want to perform a delete operation. As with the update operation, the filter string uses the SQL filter syntax and specifies which events you want to delete.

**Syntax**

The UpdateEventQueue function has the following syntax:

```
[Integer = ] UpdateEventQueue(EventReaderName, Filter, UpdateExpression, IsDelete)
```

**Parameters**

The UpdateEventQueue function has the following parameters.

**Table 109. UpdateEventQueue function parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EventReaderName</td>
<td>String</td>
<td>Name of the event reader whose queue you want to update or delete.</td>
</tr>
<tr>
<td>Filter</td>
<td>String</td>
<td>SQL filter expression that specifies which events in the queue to update or delete.</td>
</tr>
<tr>
<td>UpdateExpression</td>
<td>String</td>
<td>Update expression that specifies which fields and corresponding values to update. If you want to delete events, pass a null value for this parameter.</td>
</tr>
<tr>
<td>IsDelete</td>
<td>Boolean</td>
<td>Boolean value that indicates whether to delete the specified events. Possible values are true and false.</td>
</tr>
</tbody>
</table>

**Return value**

Number of events updated or deleted. Optional.
Examples

The following example shows how to update events in the event queue:

```java
EventReaderName = "OMNIbusEventReader";
Filter = "Node = 'Node Name'";
UpdateExpression = "Node = 'New Node Name'";
IsDelete = false;

NumUpdatedEvents = UpdateEventQueue(EventReaderName, Filter, UpdateExpression, IsDelete);
Log("Number of updated events: " + NumUpdatedEvents);
```

The following example shows how to delete events in the event queue:

```java
EventReaderName = "OMNIbusEventReader";
Filter = "Node = 'ORA_01'";
IsDelete = true;

NumDeletedEvents = UpdateEventQueue(EventReaderName, Filter, null, IsDelete);
```

### URLDecode

The URLDecode function returns a URL encoded string to its original representation.

This function parallels the Java function `java.net.URLDecoder.decode()`.

#### Syntax

The URLDecode function has the following syntax:

```java
String = URLDecode(Expression,[Encoding])
```

#### Parameters

The URLDecode function has the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>String</td>
<td>The string that you want to decode.</td>
</tr>
<tr>
<td>Encoding</td>
<td>String</td>
<td>The encoding scheme you want to use. This is optional. The recommended and default encoding is UTF-8.</td>
</tr>
</tbody>
</table>

#### Return value

The decoded string.

#### Example

The following example shows how to decode a URL encoded string back to its original representation.

```java
ReceivedString = "System.out.println%28%22Hello+World%21%22%29";
OriginalString = URLDecode(ReceivedString, "UTF-8");
Log(OriginalString);
```

This example prints the following message to the policy log:
Parser Log:
System.out.println("Hello world!");

**URLDecode**

The URLEncode function converts a string to a URL encoded format.

This function parallels the Java function java.net.URLEncoder.encode().

**Syntax**

The URLEncode function has the following syntax:

```java
String = URLEncode(Expression, [Encoding])
```

**Parameters**

The URLEncode function has the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>String</td>
<td>String that you want to encode.</td>
</tr>
<tr>
<td>Encoding</td>
<td>String</td>
<td>The encoding scheme you want to use. This is optional. The recommended and default encoding is UTF-8.</td>
</tr>
</tbody>
</table>

**Return value**

The URL encoded string.

**Example**

The following example shows how to encode the query string of a URL and form a valid URL.

```java
BaseURL = "http://hostname:port/query";
QName1 = "filter";
QVal1 = URLEncode("key='42ITA'");
QName2 = "comment";
QVal2 = URLEncode("#$&%@^%$!!","UTF-8");
Querystring = "?" + QName1 + "=" + QVal1 + "&" + QName2 + "=" + QVal2;
FullURL = BaseURL + Querystring;
Log(FullURL);
```

This example prints the following message to the policy log:

Parser Log:
http://hostname:port/query?filter=key%3D%2742ITA%27&comment=%23%24%26%40%5E%25%24%21%21

**WSInvokeDL**

The WSInvokeDL function makes web services calls when a Web Services Description Language (WSDL) file is compiled with nci_compilewsdl, or when a policy is configured using the Web Services wizard.
**Syntax**

This function has the following syntax:

\[
\text{[Return]} = \text{WSInvokeDL}(\text{WSService}, \text{WSEndPoint}, \text{WSMethod}, \text{WSParams}, [\text{callProps}])
\]

This function returns the value of your target web services call.

**Parameters**

The `WSInvokeDL` function has the following parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>WSService</code></td>
<td>String</td>
<td>This web service name is defined in the <code>/definitions/service</code> element of the WSDL file.</td>
</tr>
<tr>
<td><code>WSEndPoint</code></td>
<td>String</td>
<td>The web service endpoint URL of the target web service.</td>
</tr>
<tr>
<td><code>WSMethod</code></td>
<td>String</td>
<td>The web service method defines which method you would like to call in <code>WSInvokeDL()</code>.</td>
</tr>
<tr>
<td><code>WSParams</code></td>
<td>Array</td>
<td>The web services operation parameters are defined by <code>/definitions/message/part</code> elements in the WSDL file. It comprises an array that contains all of the parameters that are required by the specified web service operation.</td>
</tr>
<tr>
<td><code>callProps</code></td>
<td>String, Boolean, integer</td>
<td>The optional container in which you can set any of the properties, which are listed in the <code>callProps</code> properties section.</td>
</tr>
</tbody>
</table>

**callProps properties**

**Remember:** Any options that are set in `callProps` must precede the actual call to `WSInvokeDL`.

- **Chunked** divides the packets into small chunks if the server supports the feature. The default property is true.

  **Tip:** If you receive the following error message when running the `WSInvokeDL` function then set this property to false.

  Transport error: 11 Error: Length Required in policy

- **MTOM** enables or disables the Message Optimization for the SOAP message.

- **CharSet** sets the encoding other than UTF-8.

- **HTTP** the default HTTP version is 1.1. You can use this property to set the protocol version to 1.0.

- **ReuseHttpClient** enables the underlying infrastructure to reuse the HTTP client if one is available. The `ReuseHttpClient` is useful if the client is using HTTPS to communicate with the server. The SSL handshake is not repeated for each request. The parameter must be set to true or false.

- **EnableWSS** enables web Service Security. If you specify `EnableWSS`, you must also specify the following properties:
  - **WSSRepository**, which specifies the path location of WSS Repository.
  - **WSSConfigFile**, which specifies configuration file for `EnableWSS`.

- **Username** specifies the user name for basic authentication.

- **Password** specifies the password for basic authentication. **PreemptiveAuth** enables Preemptive Authentication.
• **Timeout** this property is used in a blocking scenario. The client system times out after the specified amount of time.

You can optionally set a global web Service DSA call timeout property called `impact.server.dsainvoke.timeout`. The property must be added to the Netcool/Impact server property file, `<servername>_server.props`. It is best to use the timeout property on a per policy basis as specified in `callProps.Timeout`.

The value is set in milliseconds, for example, `impact.server.dsainvoke.timeout=30000` (30 seconds).

When you set the properties in any of the `.props` files, restart the Netcool/Impact server to implement the changes.

If the `impact.server.dsainvoke.timeout` property is set, all WSIvokeDL calls use the same timeout setting.

• **MaintainSession** sets the session management to enabled status. When session management is enabled, the system maintains the session-related objects across the different requests. The parameter must be set to true or false.

• **CacheStub** caches generated stubs. This value must be set to true if either or both of the following properties are enabled, `ReuseHttpClient`, `MaintainSession`.

Examples of usage:

```java
callProps.CacheStub=true;
callProps.ReuseHttpClient = true;
```

• **CustomHeaders** adds custom header values other than the headings that are already supported in the documentation.

• **DecryptPassword** enables the decryption of an encrypted password in a policy.

• **HandleFault** is used to manage faults. Fault messages are returned from the web services server to indicate that there is a problem.

• **LogSoapMessages**: You can enable logging of the full outgoing and incoming soap messages, by adding the LogSoapMessages property to callProps and setting it to true. This property should be used for debugging purposes and not left on permanently in a production environment.

```java
CallProps = newObject();
CallProps.LogSoapMessages = "true":
WSInvokeDLResult = WSIvokeDL(WSService, WSEndPoint, WSMethos, WSParams, CallProps);
```

The logs are written to `$IMPACT_HOME/wlp/usr/servers/<server>/logs/messages.log`.

### Examples

**Remember**: Any options that are set in `callProps` must precede the actual call to `WSInvokeDL`.

Apart from its primary usage, the `callProps` container can be used to enable security. For example, if the basic authentication is enabled through the wizard, the sample policy contains the following lines:

```java
callProps.Username="username";
callProps.Password="password";
```

The following example shows how to use the `WSInvokeDL` function to send a message to the target web service.

**Example using IPL**:

```java
ServiceName = "StockQuote";
EndPointURL = "http://www.webservicex.net/stockquote.asmx";
MethodName = "GetQuote";
```
ParameterArray = { "IBM" }; 
Results = WSInvokeDL(WSService, WSEndPoint, WSMethod, WSParams, {callProps});

Example using JavaScript:
ServiceName = "StockQuote";
EndPointURL = "http://www.webservicex.net/stockquote.asmx";
MethodName = "GetQuote";
ParameterArray = [ "IBM" ];
Results = WSInvokeDL(WSService, WSEndPoint, WSMethod, WSParams, [callProps])

Use the DecryptPassword policy parameter to enable the decryption of an encrypted password in a policy that is used with the CallProps function:

CallProps=NewObject();
CallProps.Password="<Web Service encrypted using nci_crypt>";
CallProps.DecryptPassword=true;
//default is false and must be plain text password.

The password is decrypted at policy runtime and is used in plain text internally to Netcool/Impact.

You can also use the CustomHeaders parameter to add custom http header values other than the headings that are already supported in the documentation.

Headers = NewObject();
Headers.HeaderName1='HeaderValue1';
Headers.HeaderName2='HeaderValue2';
CallProps.CustomHeaders=Headers;

Use the HandleFault parameter to handle fault messages.
CallProps=NewObject();
CallProps.HandleFault=true;

When the default value is false, the policy throws an exception. When the value is true, the policy returns only the fault string message. No fault code is returned.

If the value is true, the return is similar to the following example.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Fault>
  <faultstring> some message here </faultstring>
</Fault>
```

To turn off the formatting and return the message in plain text add the following parameter anywhere in <serverName>_server.props:

`impact.wsinvoke.formatfaultmessage=false`

The changes are implemented dynamically without restarting the server.

**WSNewArray**

The WSNewArray function creates an array of complex data type objects or primitive values, as defined in the WSDL file for the web service.

You use this function when you are required to pass an array of complex objects or primitives to a web service as message parameters.
Syntax

This function has the following syntax:

```
Array = WSNewArray(ElementType, ArrayLength)
```

Parameters

The WSNewArray function has the following parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ElementType</td>
<td>String</td>
<td>Name of the complex object or primitive data type that is defined in the WSDL file. The name format is [Package.]TypeName, where Package is the name of the package you created when you compiled the WSDL file, without the .jar suffix. The package name is required only if you did not previously call the WSSetDefaultPKGName function in the policy.</td>
</tr>
<tr>
<td>ArrayLength</td>
<td>Integer</td>
<td>Number of elements in the new array.</td>
</tr>
</tbody>
</table>

Return Value

The WSNewArray returns the new array that is created by the function.

Examples

The following example shows how to use WSNewArray to creates a web services array, where you previously called WSSetDefaultPKGName in the policy. This example creates an array of the data type String as defined in the mompkg.jar file that is compiled from a WSDL file.

```
// Call WSSetDefaultPKGName
WSSetDefaultPKGName("mompkg");

// Call WSNewArray
MyArray = WSNewArray("String", 4);
```

The following example shows how to use WSNewArray to create a web services array, where you did not previously call WSSetDefaultPKGName in the policy.

```
// Call WSNewArray
MyArray = WSNewArray("mompkg.String", 4);
```

The following example invokes a web service method called runPolicy and passes in a string array as a parameter to the method. The policy creates a WSNewArray object and populates the object with 3 elements. Note that WSNewArray object is used only for arrays of primitives. For arrays of complex types, you need to create a WSNewSubObject for each array element. Also, in general it is easier to use the web services wizard to generate the web services policy from a WSDL, rather than manually creating the web services policy.

```
_RunPolicy=WSNewSubObject(RunPolicyDocument,"RunPolicy");

_Arg0=WSNewArray("java.lang.String",3);
_Arg0[0] = 'aaa';
```
_Arg0[1] = 'bbb';
_Arg0[2] = 'ccc';
_RunPolicy.Arg0Array=_Arg0;

WSPrams = {RunPolicyDocument};

WSService = 'MyWebService';
WSEndPoint = 'http://localhost:8888/MyWebService';
WSMethod = 'runPolicy';

WSInvokeDLResult = WSInvokeDL(WSService, WSEndPoint, WSMethod, WSPrams);

**WSNewEnum**

The WSNewEnum function returns an enumeration value to a target web service.

**Syntax**

This function has the following syntax:

```plaintext
[Return] = WSNewEnum(ENUMType, EnumValue);
```

**Parameters**

The WSNewEnum function has the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EnumType</td>
<td>String</td>
<td>The enumeration class name that exists in the package that is created by nci_compilewsdl</td>
</tr>
<tr>
<td>EnumValue</td>
<td>String</td>
<td>The enumeration value to return</td>
</tr>
</tbody>
</table>

**Return Value**

A new enumeration type and value.

**Example**

The following example shows how to use the WSNewEnum function to send a message to the target web service.

```plaintext
```

**WSNewObject**

The WSNewObject function creates an object of a complex data type as defined in the WSDL file for the web service.

You use this function when you are required to pass data of a complex type to a web service as a message parameter.

**Syntax**

This function has the following syntax:

```plaintext
Object = WSNewObject(ElementType)
```
Parameters

This WSNewObject function has the following parameter.

Table 115. WSNewObject function parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ElementType</td>
<td>String</td>
<td>Name of the complex data type that is defined in the WSDL file. The name format is [Package.]TypeName, where Package is the name of the package you created when you compiled the WSDL file, without the .jar suffix.</td>
</tr>
</tbody>
</table>

Return Value

A new web services object.

Examples

The following example shows how to use WSNewObject to create a web services object, what you previously called WSSetDefaultPKGName in the policy. This example creates an object of the data type ForwardeeInfo as defined in the mompkg.jar file compiled from the corresponding WSDL.

// Call WSSetDefaultPKGName
WSSetDefaultPKGName("mompkg");

// Call WSNewObject
MyObject = WSNewObject("ForwardeeInfo");

The following example shows how to use WSNewObject to create a web services object, where you did not previously call WSSetDefaultPKGName in the policy.

// Call WSNewObject
MyObject = WSNewObject("mompkg.ForwardeeInfo");

WSNewSubObject

The WSNewSubObject function creates a child object that is part of its parent object and has a field or attribute name of ChildName.

Syntax

This function has the following syntax:

Object = WSNewSubObject(ParentObject, ChildName)

Parameters

This WSNewSubObject function has the following parameters.

Table 116. WSNewSubObject function parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParentObject</td>
<td>String</td>
<td>Name of the parent object</td>
</tr>
<tr>
<td>ChildName</td>
<td>String</td>
<td>Name of the new child object</td>
</tr>
</tbody>
</table>
Return Value

A new web services child object.

Examples

The following example shows how to use WSNWewSubObject to create a web services child object:

```java
// Call WSNWewSubObject
ticketId=WSNewSubobject(incident, "TICKETID");
```

WSSetDefaultPKGName

The WSSetDefaultPKGName function sets the default package that is used by WSNWewObject and WSNWewArray.

The package name is the name that you supplied to the nci_compilewsdl script when you compiled the WSDL file for the web service. It is also the name of the JAR file that is created by this script, without the .jar suffix.

Syntax

This function has the following syntax:

```java
WSSetDefaultPKGName(PackageName)
```

Parameters

The WSSetDefaultPKGName function has the following parameter.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PackageName</td>
<td>String</td>
<td>Name of the default WSDL package used by WSNWewObject and WSNWewArray.</td>
</tr>
</tbody>
</table>

Example

The following example sets the default package that is used by subsequent calls to WSNWewObject and WSNWewArray to google.

```java
WSSetDefaultPKGName("google");
```
Appendix. Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785 U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement might not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.
IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
2Z4A/101
11400 Burnet Road
Austin, TX  78758  U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM’s future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM’s suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to
IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

---

**Trademarks**

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at www.ibm.com/legal/copytrade.shtml.

Adobe, Acrobat, PostScript and all Adobe-based trademarks are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other product and service names might be trademarks of IBM or other companies.
Index

Special characters

- 20
!= 20
/ 20
* 20
@ notation 15
% 20
+ 20, 21
= 19, 20
== 20

A
Access Service-related information 3
accessibility viii
Activate 111
ActivateHibernation 112
activating 89
AddDataItem 113
AOL Instant Messenger 3
array 11
arrays 94
finding distinct values 95
finding the length 94
assignment operator 19

B
backslash
See line continuation character
BatchDelete 114
BatchUpdate 116
BeginTransaction 117
Bitwise operators 19
books
see publications vii
boolean operators 20

C
CallDBFunction 117
calling procedures 59
returning database rows 70
returning single value 68
CallStoredProcedure 118
CallStoredProcedure variable 55, 57, 60, 64, 69, 72, 75
chained policy 45
ClassOf 119
clear cache syntax 36
code commenting 46
command line utility
nci_trigger 2
CommandResponse 47, 120
commenting 46
CommitChanges 127
CommitTransaction 128
comparison operators 20
concatenation
strings 21
correct 10
control structures 22
conventions
typeface xi
ConvertObjectsToJSON 128
ConvertXMLToImpactObjects 129
creating 2
CurrentContext 131
custom code encapsulating 31
customer support ix
Customize data output 9

data handling 2
data item 14
data types
complex 10
policy-level 10
simple 10
DatatItem (built-in variable) 17
DatatItems (built-in variable) 16
date
format 10
date patterns 37
DB2 SQL Array 76
DB2 SQL automatic schema
discovery 73
DB2 SQL IN parameters 75
DB2 SQL INOUT parameters 75
DB2 SQL OUT parameters 75
DB2 SQL parameters 73
DB2 SQL result set 73, 75
DB2 SQL scalar values 73
DB2 SQL Stored Procedure 76
DB2 SQL stored procedure examples 75
DB2 SQL stored procedures 75
DB2 SQL stored procedures overview 73
Decrypt 132
DeleteDataItem 132
DeleteEvent 15
Deploy 133
directory names
notation xii
DirectSQL 135
DiscoverProcedureSchema
setting 64
Distinct 137

education
See Tivoli technical training
Encrypt 138
encrypted policy 45
environment variables
notation xii
Escape function 139
eval 139
evalArray 140
event container 14
event handling 2
event state variables 15
EventContainer (built-in variable) 15
example 38
exceptions 35
handlers 35
raising 35
Exit 141
Extract 143

F
filters 79
LDAP filters 80
Mediator filters 82
SQL filters 79
fixes
obtaining viii
Float 143
FormatDuration 144
function 163
Activate 111
ActivateHibernation 112
AddDataItem 113
BatchDelete 114
BatchUpdate 116
BeginTransaction 117
CallDBFunction 117
CallStoredProcedure 118
ClassOf 119
CommandResponse 120
CommitChanges 127
CommitTransaction 128
CurrentContext 131
Decrypt 132
DeleteDataItem 132
Deploy 133
DirectSQL 135
Distinct 137
Encrypt 138
Eval 139
EvalArray 140
Exit 141
Extract 143
Float 143
FormatDuration 144
GetByFilter 145
GetByKey 147
GetByLinks 148
GetByXPath 150
GetClusterName 154
GetDate 154
GetFieldValue 155
GetGlobalVar 156
GetHibernatePolicies 162
GetHTTP 157
GetScheduleMember 163
GetServerName 165

© Copyright IBM Corp. 2006, 2016

251
GetByLinks 148
GetByPath 150
GetClusterName 154
GetDate 154
GetFieldValue 155
GetGlobalVar 156
GetHibernatePolicies 162
GetHostAddress 163
GetHTTP 157
GetScheduleMember 163
GetServerName 165
GetServerVar 165
Hibernate 166
hibernations 87
removing 90
retrieving 88
waking 89

H
Hibernate 166
hibernations 87
removing 90
retrieving 88
waking 89

I
ICQ 3
If statements 22
Illegal function 167
Impact policy language 1
input parameters 41
Int 167
IPL
See Impact policy language
IPL to XML function
adding new sub element 100
adding the content to XML element
object 103
adding XML attributes element
object 101
adding XML attributes to element
objects 102, 103
adding XML comments element
object 104
adding XML element objects to each
other (nesting) 105
appending content to XML element
object 104
creating unassociated element 101
creating XML document object 100
generating XML string from document
object 105
IPL to XML functions
default XML entities 106
element ordering 106
elements 106
examples 106
overview 99
XML entities 106

J
Jabber messaging service 3
Java Policy functions
GetFieldValue 155
JavaCall 168
NewJavaObject 177
overview 28
SetFieldValue 213
JavaCall 168
JournalEntry 15
JRExec server 170
running commands 47
JRExecAction 170

K
Keys 171

L
LDAP filters 80
Length 172
LIKE 20
line continuation character 46
local transactions 30, 49, 117, 127, 128,
208
best practices 51
Local transactions template 49
LocalTime 173
Log 174
Log function 9

M
manuals
See publications vii
mathematic operators 20
Mediator filters 82
Merge 175
Microsoft Messenger 3

N
nci_policy script 5
nci_trigger 2, 4, 44
nci_trigger script 41
NewEvent 176
NewJavaObject 177
NewObject 179
notation
environment variables xii
path names xii
typeface xii
Num (built-in variable) 17

O
online publications
accessing viii
operators 19
Operators 19
Oracle stored procedure
example 55, 58, 61
ordering publications viii
output parameters 41

P
parameter contexts
creating 63
ParseDate 179
ParseJSON 180
PassToTBSM 234