

# HP Operations Orchestration

## Best Practices for Content Authoring

Document Release Date: September 2015  
Software Release Date: September 2015



## Legal Notices

### Warranty

The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

The information contained herein is subject to change without notice.

### Restricted Rights Legend

Confidential computer software. Valid license from HP required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

### Copyright Notice

© Copyright 2015 Hewlett-Packard Development Company, L.P.

### Trademark Notices

Adobe™ is a trademark of Adobe Systems Incorporated.

Microsoft® and Windows® are U.S. registered trademarks of Microsoft Corporation. UNIX® is a registered trademark of The Open Group.

This product includes an interface of the 'zlib' general purpose compression library, which is Copyright © 1995-2002 Jean-loup Gailly and Mark Adler.

## Documentation Updates

The title page of this document contains the following identifying information:

- Software Version number, which indicates the software version.
- Document Release Date, which changes each time the document is updated.
- Software Release Date, which indicates the release date of this version of the software.

To check for recent updates or to verify that you are using the most recent edition of a document, go to: <http://h20230.www2.hp.com/selfsolve/manuals>

This site requires that you register for an HP Passport and sign in. To register for an HP Passport ID, go to: <http://h20229.www2.hp.com/passport-registration.html>

Or click the **New users - please register** link on the HP Passport login page.

You will also receive updated or new editions if you subscribe to the appropriate product support service. Contact your HP sales representative for details.

## Support

Visit the HP Software Support Online web site at: <https://softwaresupport.hp.com/>

This web site provides contact information and details about the products, services, and support that HP Software offers.

HP Software online support provides customer self-solve capabilities. It provides a fast and efficient way to access interactive technical support tools needed to manage your business. As a valued support customer, you can benefit by using the support web site to:

- Search for knowledge documents of interest
- Submit and track support cases and enhancement requests
- Download software patches
- Manage support contracts
- Look up HP support contacts
- Review information about available services
- Enter into discussions with other software customers
- Research and register for software training

Most of the support areas require that you register as an HP Passport user and sign in. Many also require a support contract. To register for an HP Passport ID, go to:

<http://h20229.www2.hp.com/passport-registration.html>

To find more information about access levels, go to:

[http://h20230.www2.hp.com/new\\_access\\_levels.jsp](http://h20230.www2.hp.com/new_access_levels.jsp)

HP Software Solutions Now accesses the HPSW Solution and Integration Portal Web site. This site enables you to explore HP Product Solutions to meet your business needs, includes a full list of Integrations between HP Products, as well as a listing of ITIL Processes. The URL for this Web site is

<http://h20230.www2.hp.com/sc/solutions/index.jsp>

# Contents

Introduction.....	5
General Guidelines .....	5
Naming.....	5
Casing .....	5
Content Packs/Projects .....	5
Folders.....	5
Flows, Operations, and Steps.....	6
Inputs and Outputs .....	6
Configuration Items.....	7
Acronyms.....	8
Plural versus Singular .....	8
Legal Naming .....	8
Folder Structure .....	9
Content Structuring .....	9
Handling Multiple Integration Product Versions .....	10
Create Operations or Flows?.....	10
Descriptions.....	10
Operations and Flows .....	10
Folders.....	11
Additional Description Recommendations.....	12
Date and Time, Numbers and Units .....	12
Flow Authoring.....	13
Flow Design .....	13
Reusing Existing Content .....	13
Create Modular Flows .....	13
Use Step References .....	14
Keep Flows Small.....	14
Start Step.....	15
Transitions.....	15
Callouts.....	16
Layout.....	16
Icons.....	18
Operations.....	20
Avoid Creating Multiple Operations that Run the Same Command.....	20
Soft Copies Instead of Hard Copies .....	21
JavaScript “Do Nothing” Operation.....	21
Inputs.....	23

Assignment.....	23
Handling Multiple Versions of the Same Product .....	23
Handling Versions .....	23
Timeouts .....	24
Outputs.....	24
Responses .....	24
Error Messages .....	25
Scriptlets.....	25
Coding Style.....	25
Long Scriptlets.....	25
Validate Scriptlets .....	25
JavaScript Best Practices and Scriptlets .....	26
Exception Handling.....	27
Handling Scriptlets in OO .....	27
OO Special Objects and Methods .....	28
Scriptlet Capabilities .....	28
Reusing Scriptlets .....	29
Handling Variables .....	29
Content Pack Versioning .....	30
Deprecation.....	30
Avoid Using Content from <i>Deprecated</i> Folders .....	30
When to Deprecate Content.....	30
How to Deprecate Content .....	31
References .....	31

## Introduction

This document provides guidelines for OO Content Authoring, as well as aligning Flow Authors to a common way to develop, document, package, and deliver OO 10.x Content.

The target audience is the Flow Author, a persona who creates and debugs flows in HP OO Studio.

This document starts with general guidelines, and then continues with best practices for flows, operations, scriptlets, and content pack versioning and deprecation.

## General Guidelines

This section contains the recommendations regarding casing, the naming of the content packs/projects, folders, flows/operations/steps, inputs/outputs, configuration items, acronyms, when to use plural/singular, and legal naming.

## Naming

### Casing

The names of different types of objects must be consistent, as follows:

- Use title case for folders, flows, operations, steps, and configuration items. Using title case means that you capitalize the first letter for all words, except helper words like 'a', 'the', 'and', 'by', 'for', 'to', 'from', and so on. [Examples: Reboot Host, Check the Log Files](#)
- Use camel case for inputs, outputs, results, flow variables. Camel case occurs when the first letter is in lower-case, and subsequent first letters of words contained in the name are upper-case, with no spaces within the name. [Example: serverName](#)
- Use lower case for responses.  
[Examples: success, failure, has more, and no more](#)
- Use lower case with dashes for content pack names or project names.  
[Example: oo-linux-remediation-cp.jar](#)

### Content Packs/Projects

Naming of multiple content packs must be consistent as follows:

**oo-domain | use case | cp-<version>.jar**

[Examples: oo-remediation-cp-1.0.121.jar, oo-linux\\_remediation-cp-4.5.0.jar](#)

*In OO 10.x the content packs are independent of the OO 10.x platform (Central, Studio, RAS). Consider including the version (for example, oo10-remediation-cp) in the artifact name only when it's required by use cases, such as content that is delivered/supported on multiple OO major versions (OO 9.x and OO 10.x). In other cases, it is a best practice *not* to link the OO platform version with CPs version.*

### Folders

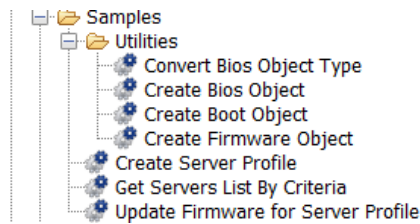
Naming must be consistent between multiple folders. The names of the folders can be a maximum of 128 characters long, and are not case-sensitive. Using the OO common folder names as much as possible. In this way, the folders are aligned to a common naming and as a result, the operations/flows groups will be more visible.

The OO common folder names are:

- "Community" for content developed by and for Community
- "Deprecated" for content that was deprecated according to the deprecation policies
- "Integrations" for content specific per integration product

- “Samples” for flows that demonstrate how operations and flows can realize simple use cases
- “Utilities” for reusable flows and operations.

The following directory structure reflects this nomenclature. Flows in the Utilities folder are also used in other flows, such as “Create Server Profile” and “Update Firmware for Server Profile”. The common Utility flows also create JSON objects.



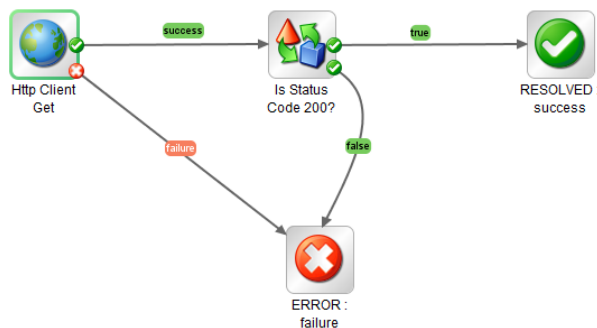
## Flows, Operations, and Steps

Naming must be consistent between multiple content entities. The names of the flows can be a maximum of 128 characters long, and are not case-sensitive.

Use <Verb> <Noun> name format for flow and operation names. Also, where possible, use Create, Read, Update, Delete, or another group as common verbs with Get, and Set for <Verb>. When adding operations or flows to an existing integration, use the existing naming for <Verb>. This will ensure consistency with the out-of-the-box flows and operation names. [Examples: Create Snapshot, Get Image Details](#)

It is recommended to rename steps to match the actual action. When using such content in specific flows, it is a good practice to rename the step with the name of the actual action that is performed. This can enhance the clarity of the flow.

[Example: the following picture shows that a step named “Is Status Code 200?” is more intuitive than “String Equals”.](#)



## Inputs and Outputs








Input and output names must be aligned with common names. For inputs, it is recommended that you use common prefixes/suffixes such as:

- host
- username
- password
- port
- timeout
- protocol

It is preferable to add a prefix with the integration product name or acronym such as:

<product>Host, <product>Username, <product>Password

Examples: `omiHost`, `omiUsername`, `omiPassword`

Input	Required	Type
<code>omiHost</code>	<input checked="" type="checkbox"/>	
<code>omiUsername</code>	<input type="checkbox"/>	
<code>omiPassword</code>	<input type="checkbox"/>	
<code>proxyHost</code>	<input type="checkbox"/>	
<code>proxyPort</code>	<input type="checkbox"/>	
<code>proxyUsername</code>	<input type="checkbox"/>	
<code>proxyPassword</code>	<input type="checkbox"/>	

The common outputs are: `returnResult`, `returnCode`, `exception`, and `FailureMessage`.

The *returnResult* is the primary result of an operation.

The *returnCode* is an internal code: 0 for success, -1 for failure. A *returnCode* is used to determine the response of a step.

The *exception* is the exception stack trace, which is mainly used for debugging the root cause in case of failures.

The *FailureMessage* is the error message that is used to propagate exception messages between steps and flows. This output name starts with uppercase 'F' because OO Studio automatically makes it uppercase by default when a flow is created.

## Configuration Items

The following format is recommended for configuration item names:

**<Configuration item prefix>\_<Project name>\_<Type>**

The *Configuration item prefix* consists of the category initials. For example, SL for Selection Lists, SA for System Accounts, and so on.

The *Project name* is simply the project to which the configuration item belongs.

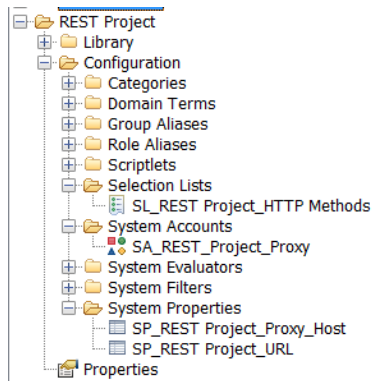
The *Type* is the configuration item type, such as: selection lists, system accounts, system properties and so on.

Using the recommended format enables you to better differentiate the following:

- A flow variable from a configuration item.
- Configuration items by their categories. The prefix helps to easily distinguish between a selection list and system account
- Configuration items by the project name. [Example: a selection list that belongs to REST Project or to another project](#)
- Configuration items by their type. [Example: the system property that contains the URL from that containing the proxy host](#)

Starting with OO 10.20, configuration items can be structured in folders. This way, working with too long names will be avoided.

In the following tree structure, you can see examples of the recommended format for configuration items names:



## Acronyms

The following recommendations apply for well-known abbreviations such as: ID, URL, HTML, and HTTP:

- Use uppercase acronyms rather than Camel case for *folder*, *flow*, or *operation* names.  
Example: [HTTP Client](#)
- Use Camel case for inputs and outputs, such as: [httpUrl](#)

## Plural versus Singular

Use plurals for *folder names*.

Examples: [Servers](#), [Databases](#), [Operating Systems](#), [Images](#)

Use singular for *flows* and *operations* that interact with a single resource, such as getting details on a device.

Examples: [Get Server Details](#), [Run Script on Server](#)

## Legal Naming

Following are the valid characters for naming folders, operations, flows, and configuration items:

- Alphanumeric characters
- Spaces
- Curly, round and square brackets and full width brackets
- Underscore, full width underscore
- Hyphen, full width hyphen
- Dot, full width katakana dot, full stop, ideographic half width stop, half-width katakana middle dot, ideographic full stop, katakana middle dot, middle dot

The following strings are reserved names and should not be included in flow names, configuration item names, and folder names:

- ., .attic, .section, .section.xml, .xml, .properties
- CON, PRN, AUX, CLOCK, NUL, COM1, ..., COM9, LPT1, ..., COM9

Additional naming conventions:

- Flow names, configuration item names, and folder names cannot contain the sequence “..”
- Flow names, configuration item names, and folder names cannot end with “.”



## Folder Structure

This section describes how to organize content within folders to create a consistent folder structure.

### Content Structuring

Make sure that the folder structure is well-defined and consistent between projects, so that other authors will be able to locate your flows, operations, and utilities.

It is a best practice to organize the content according to functionality/use case/product:

**/Library/<Company | Publisher>/<Domain>/ [Area]/<Use Case | Project>/ [version]**

*Company | Publisher* - Having a common top-level folder is advantageous because it clearly separates your own developed content from that of OO and also makes it easier to set permissions.

*Domain* - The domain is typically tied to larger groups within the organization, as well as the logical separation by technologies.

Examples: Remediation, Incident Management, Cloud, Virtualization

*Area* (optional) – Depending on the number of flows and size of your organization, this can be beneficial in that it creates an extra layer based upon either a technical or functional area.

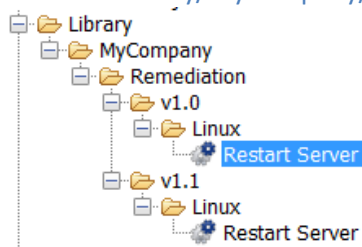
Example: Linux

*Use Case | Project* - This level offers a variety of uses, much of which depends on the type and pace of development of OO flows.

Example: Restart Server

*Version* (optional) - Useful for when you must run several versions of the same flow at the same time.

Examples: Library/MyCompany/Remediation/v1.0/Linux/Restart Server  
Library/MyCompany/Remediation/v1.1/Linux/Restart Server



Note: It is recommended that you use the subversioning capabilities of Studio for multiple versions of the same flow. For more details, refer to “Working with Source Control in HP OO Studio” in the *Studio Guide*.

## Handling Multiple Integration Product Versions

The folder structure of the integrations must be kept small. When you create integrations with many products for the same vendor, consider organizing the flows and operations in a folder structure containing the official vendor and official product names (or broadly used acronyms such as ALM, QC):

**../Integrations/Vendor/Product/ [ProductVersion]**

Example:

[.../Integrations/Hewlett-Packard/OperationManager i/1.0](#)

## Create Operations or Flows?

For basic operations, such as CRUD, it is recommended that you create operations rather than flows. Also, developing operations is a good choice when you need to support multiple product versions, or hide complex API in atomic operations. Also, programming features such as refactoring provided by different IDEs (such as Eclipse and IntelliJ) helps you to speed up the development and maintenance time. For more details regarding the development of operations, please read *The OO 10 SDK - Getting Started* tutorial.

In the following example, the “Wait for task” flow has a large number of steps; in such situations, you don’t want to have busy wait loops in flows as it makes triaging results very difficult. Writing a new “Wait” operation allows for smaller granularity checking, without creating hundreds of tracked results in the OO run logs.

Step	Result
Wait for task	success
Get Task Details	success
Still Running?	true
Sleep	success
Over Max Time?	continue
Get Task Details	success
Still Running?	true
Sleep	success
Over Max Time?	continue
Get Task Details	success
Still Running?	true
Sleep	success
Over Max Time?	continue
Get Task Details	success
Still Running?	true
Sleep	success
Over Max Time?	continue
Get Task Details	success
Still Running?	true
Sleep	success

On the other hand, creating new operations involves development and test time, while flows can use out-of-the-box operations that have been already tested. To avoid these additional resource requirements, it is a best practice to create common and modular code that can be reused.

## Descriptions

### Operations and Flows

The operations/flows must have clear and consistent descriptions. These descriptions should include the following sections:

*Summary* is a short description of the operation/flow functionality. It should contain enough information to understand the use case.

For *Inputs*, the following items are required for each input whenever the information is applicable and available:

- Description

- Format
- Default Values
- Valid Values
- Examples

For *Results*, the following items are required for each:

- Description
- Format
- Examples

The *Responses* section contains the description of each response.

*Notes (optional)* is a custom note section that contain prerequisites, some limitations, or other information deemed useful for the user.

*Keywords (optional)* is a section to help the flow/operation to be found in searches.

It is recommended that all inputs and outputs are documented in the description, and that they are listed in the same order in which they appear in the Inputs tab.

The following items must be enclosed in double quotation marks when used in a sentence:

- Inputs
- Responses or results
- Examples and values

Example: You must specify a value for the “proxyHost” input.

Following is an example of a *long description* spread across multiple lines:

authType - The type of authentication used by this operation when trying to execute the request on the target server. The authentication is not preemptive: a plain request not including authentication info will be made and only when the server responds with a 'WWW-Authenticate' header the client will send required headers. If the server needs no authentication but you specify one in this input the request will work nevertheless. The client cannot choose the authentication method and there is no fallback so you have to know which one you need. If the web application and proxy use different authentication types, these must be specified like in the Example model.

Default: basic

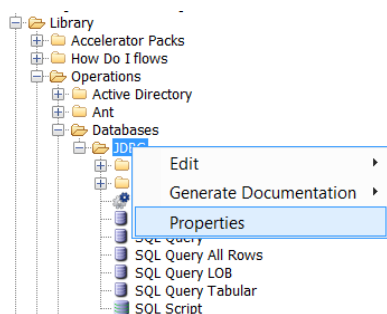
Valid: basic, digest, ntlm, kerberos, any, anonymous, "" or a list of valid values separated by comma.

The accepted and recommended HTML tags inside the descriptions are:

*a*, *b*, *blockquote*, *br*, *cite*, *code*, *dd*, *dl*, *dt*, *em*, *i*, *li*, *ol*, *p*, *pre*, *q*, *small*, *span*, *strike*, *strong*, *sub*, *sup*, *u*, *ul* and *appropriate attributes*.

## Folders

Folder descriptions can be added by right-clicking on a folder and then selecting Properties, as shown in the following image:



Add *general information about the integration* such as versions supported, API, deployment requirements and third party libraries at folder level.

Example:

The content in this folder integrates with VMware vSphere vCenter and ESX(i).

Supported versions: VMware vSphere 5.1, 5.5

Deployment requirements: Windows or Linux RAS

Interfaces / APIs: SOAP

Third party libraries: VI Java API 5.1

## **Additional Description Recommendations**

In Descriptions, use:

- Formulations with second person singular instead of third person neutral.

Example:

If you do not specify a value for this input, the operation uses the default value.

Instead of:

If a value for this input is not specified, the operation uses the default value.

- The same syntax for section headings everywhere in the description.

Example: use "Examples" not "e.g." or "ex".

- The values from "Default Values", "Valid Values", "Format ", "Examples" that contain only a punctuation mark should be between apostrophes.

Example: Default Value: ‘,’

- Often the summary contains all relevant keywords. But if it is not the case, add a line in the end of the description with keywords. This is particularly helpful for shared and reusable content.

Example: incident management

## **Date and Time, Numbers and Units**

The representation of Date and Time, Numbers and Units must be consistent.

When integrating with a certain product consider using the *date and time specific to the integration product*. If you report the details of the events from a system, then the expectation is to have the date in the same format. In the event that there is a need for conversion, use the Date and Time content.

It is preferable that you use *the most common and specific units*. Asking the user to provide a timeout in milliseconds or a disk size in kilobytes simply creates extra calculations.

Examples: Prefer seconds over milliseconds, base 10 (MegaBytes) over base 2 (Mebibyte).

## Flow Authoring

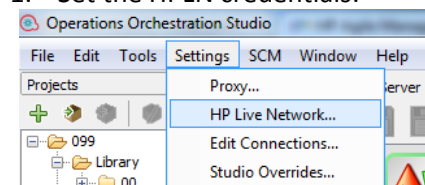
### Flow Design

#### Reusing Existing Content

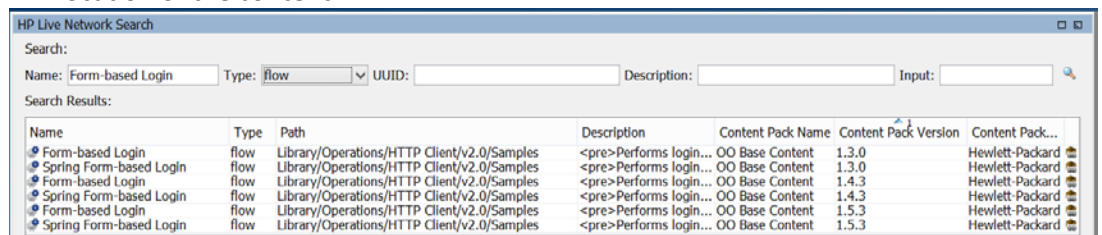
Because there are many flows and operations already created, check first if a flow or operation already exists using OO Studio Search, OO Studio HPLN search capability (introduced in OO10.10) and product documentation.

Note: The following pictures shows how to use OO Studio *HP Live Network Search*:

1. Set the HPLN credentials:



2. The HPLN Search pane is located at the bottom of the Studio Workspace. Right-click on a row to view the full description of the search results, or, to connect directly to the location of the content in HPLN.



You can find more details on this feature in “Searching Content on HP Live Network from Studio” in the *Studio Guide*.

#### Create Modular Flows

It is a best practice to design modular flows rather than a single flow that performs all tasks, and it would be even better to have many subflows with very specific roles. Each flow should have *a single, well-focused purpose*, executing only single aspect of the desired functionality. Also, the name of flow must be self-explanatory.

The benefit of creating such modular flows is that they are *easier to create, test, maintain, enhance, and reuse*.

Why it is important to separate more responsibilities into separate flows? Because each responsibility is an axis to change. When the requirements change, that change will be manifest through a change in responsibility among the flows. If a flow has more than one responsibility, then the responsibilities become coupled. This kind of coupling leads to fragile designs that break in unexpected ways when changed.

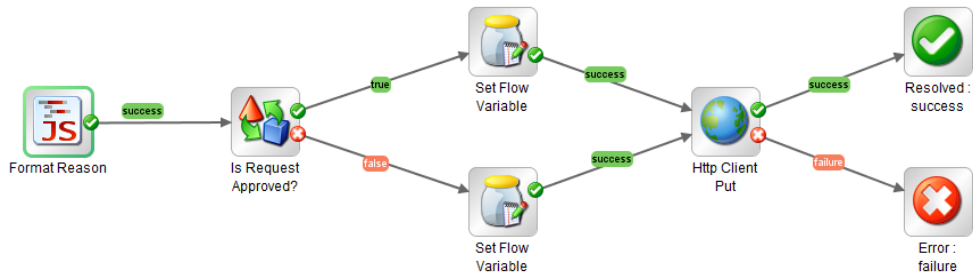
Modular flows are *easier to test*. Breaking a large flow into smaller independent flows means that parts of a complex flow can be tested independently and likely with fewer tests.

Modular flows have *greater cohesion*. Smaller flows adapt more readily to new requirements and are more easily stabilized.

Modular flows can be *reused*. This “economy” eliminates the duplication of knowledge or logic that has already been implemented.

Modular flows are *easy to understand*. Imagine that you have been assigned to fix a bug in the flow shown in Figure A (this page). Now take a look at the flow shown in Figure B (page 15). Which flow would you rather work on?

The following example shows a well-focused flow with a clear purpose.



### Use Step References

A step reference is flow or operation placed in a wrapper and used in other complex flows. Step referencing is a common practice when reusing flows and operations. Step references are advantageous because a wrapper flow can be tailored to specific process and user requirements, thereby reducing the overall amount of flow authoring work.

### Keep Flows Small

A flow should fit on the canvas with Studio maximized, with a 1:1 view magnification. It is recommended that you employ modular flows. In fact, when you have flows with more than 10 steps, you should consider splitting them into subflows. While large flows with 10 steps or more are allowed, it is a best practice to carefully examine such flows to identify where sequences of steps can be broken down into subflows.

Small flows are recommended because they are *easy to test, easy to debug, easy to maintain, easy to understand for others*.

For example, the following flow epitomizes poor design practices (it has more than 40 steps, spaghetti transitions and the start step is obscure placed).

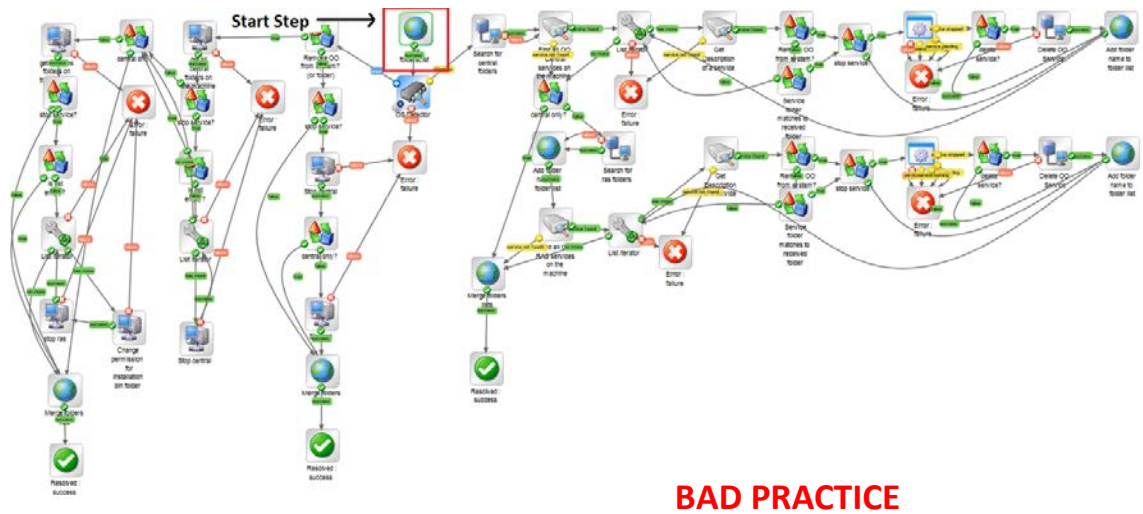


Figure A

In Figure B, the steps have been grouped into subflows; the main flow is much improved:

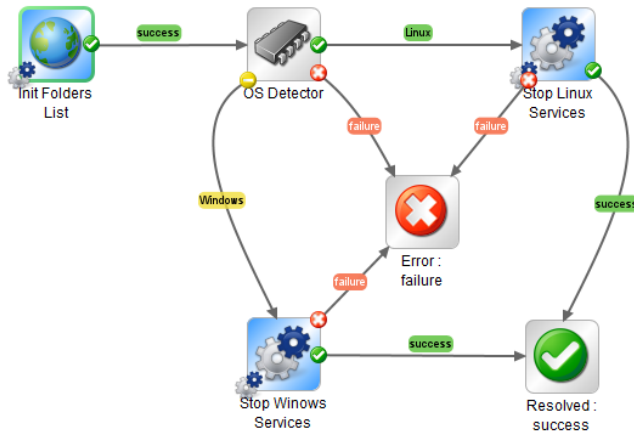


Figure B

### Start Step

The start step should be located in the upper-left corner of the flow, with the following exceptions:

- The Start step has many responses, each of which leads to another step
- Placing the start step in the upper-left corner would cause excessive visual complexity, such as crossing of transitions

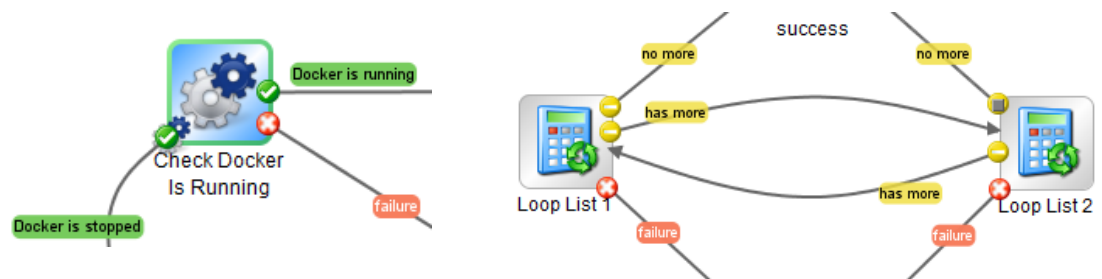
### Transitions

#### Keep Transition Neat and Tidy

As much as possible, transition lines should not cross over each other.

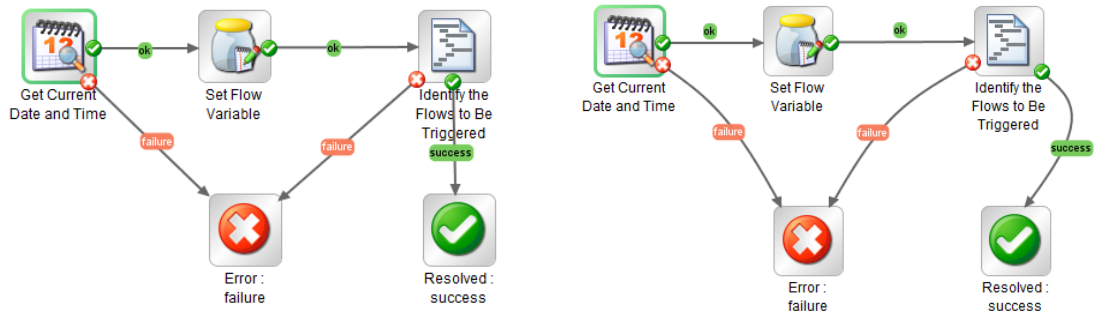
Collapse multiple transitions from one step to another (in same direction) so that a single line represents all of the transitions.

Place transition labels close to the source/origin step:



#### Keep Transition Lines Out of Step Names

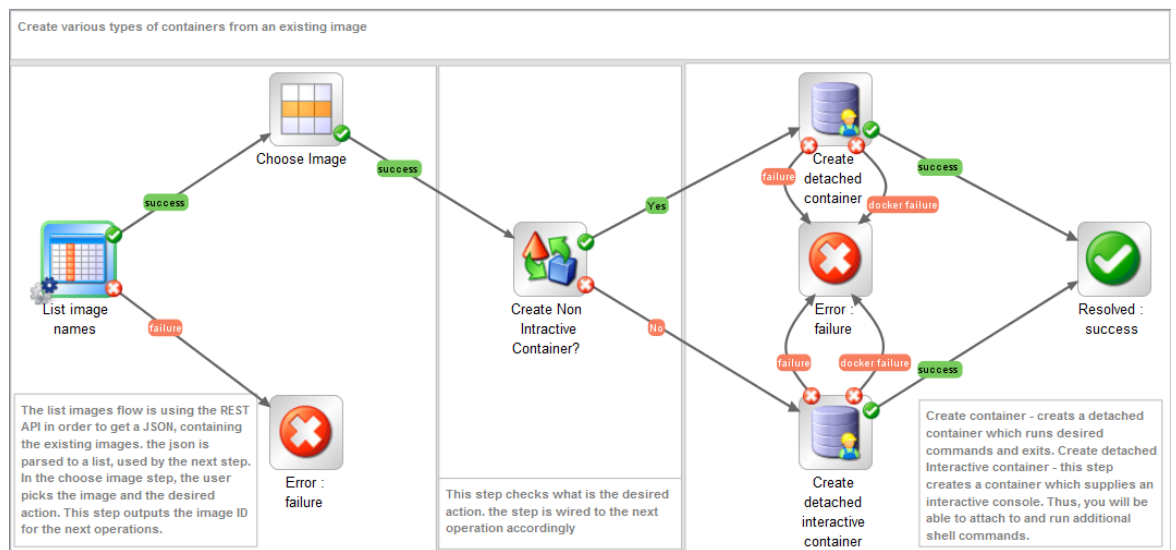
It is helpful for users if you use a well-chosen, meaningful step name. For example, in the following pictures, the diagram on the left is quicker to code, but the example on the right is easier to read and debug.



## Callouts

It is helpful to see a well-chosen step name, but sometimes you want to have more information displayed in the flow panel. The OO *callouts* represents a good way to accomplish this.

The following example shows how you can use callouts in a flow:



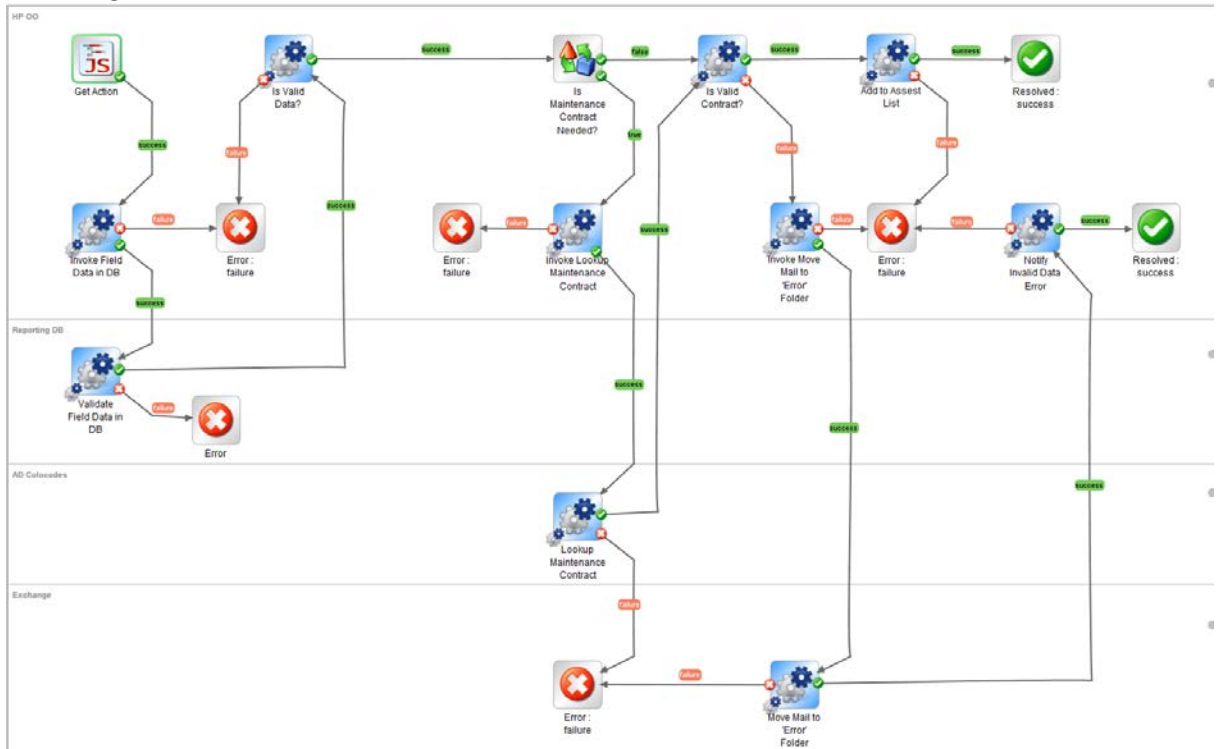
## Layout

When designing a flow layout, it is a best practice to consider well-known process automation layouts.

*Swimlane* (cross-functional) diagrams are process flowcharts that can be expanded to show times (such as when tasks are complete and how long they take) as well as information about who is responsible for what. Swimlane diagrams enable you to identify time traps - which processes take the most time - as well as capacity constraints, and which resources are getting bogged down with work.

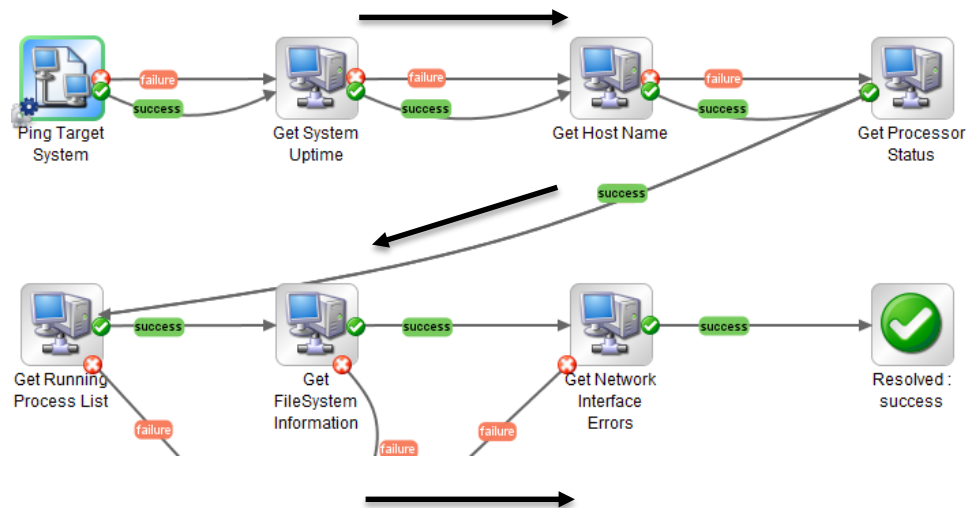


Following is an OO Swimlane flow:

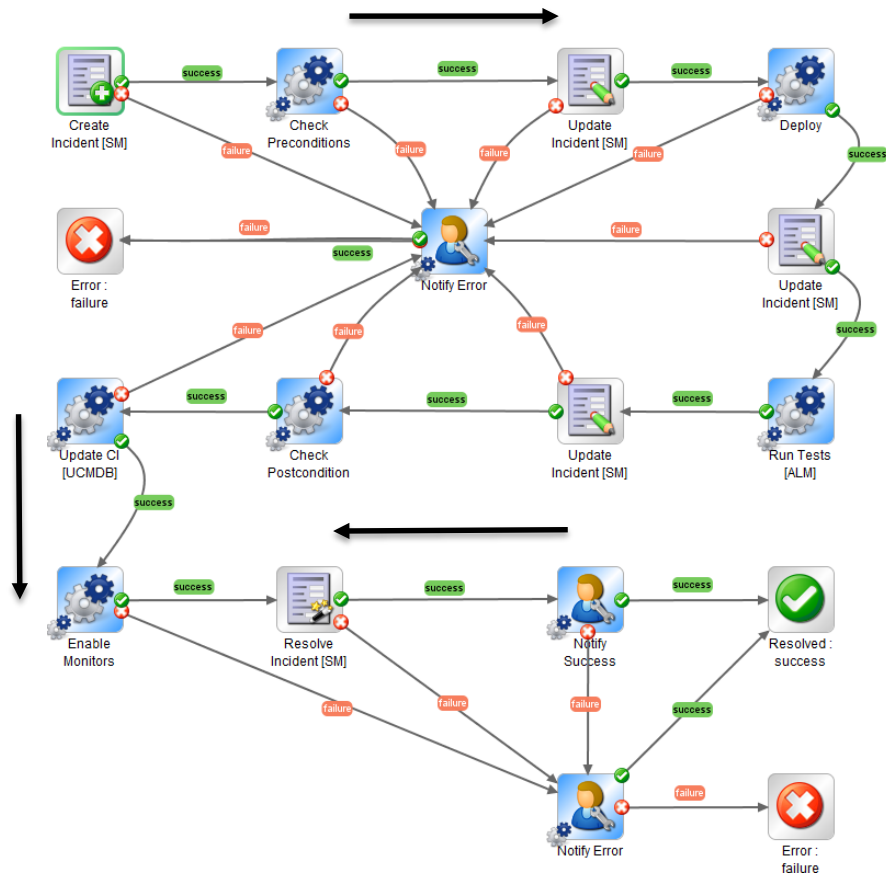


Note: For more details regarding the customization of the transitions, refer to *Customizing Flow Transitions*.

“Z” Layout diagrams progress from left to right and from top to bottom, and are best-suited when transitions (success, failure) to return steps do not cross transitions between regular steps.



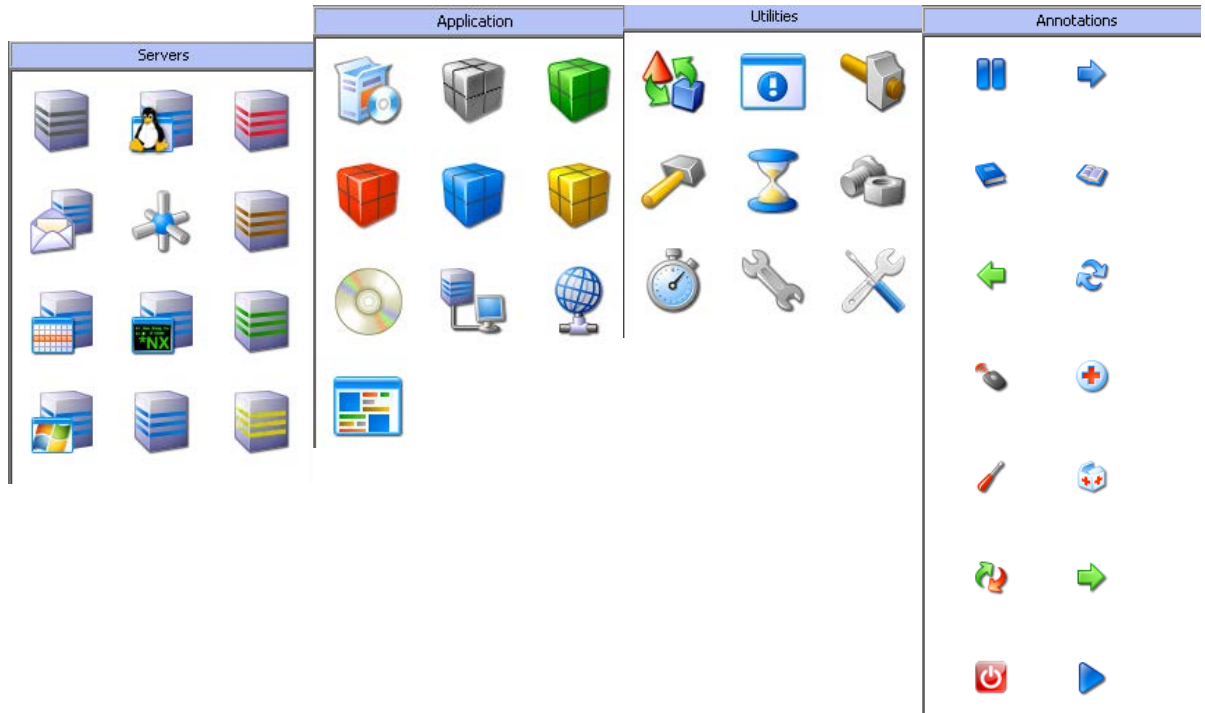
“2” Layout diagrams progress from left to right and from top to bottom, and are best-suited when there are “central” steps - such as notification or exception handler - in the flow, or to avoid crossing transitions.



## Icons

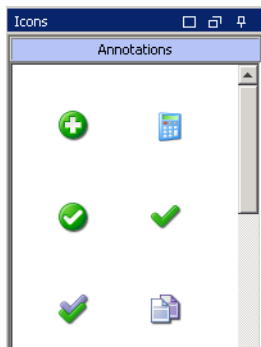
The saying, “a picture is worth a thousand words” applies to the use of icons, which can quickly communicate content and structure, as well as improve readability.

To achieve quick and easy communication, select relevant icons. For example, use specific, well-known *icons to represent resources* (such as Servers or Databases). When appropriate and helpful, add *annotations* to represent different actions (such as Start, Stop, and Check).

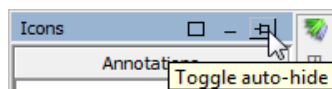


To add annotations:

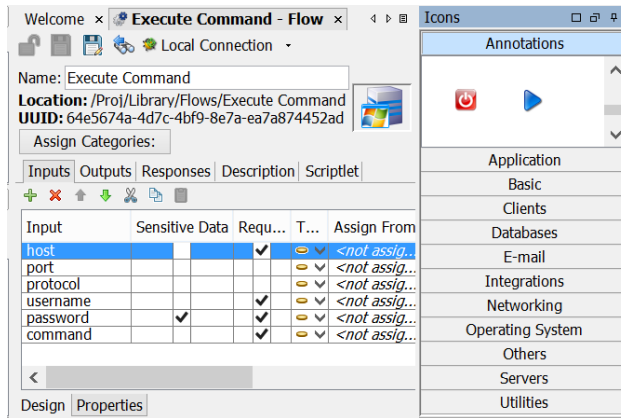
1. Expand the Icons panel in Studio:



2. Fix it on the screen by clicking on the small pin:



3. Open the Properties of a flow or operation:



4. Drag the Annotation icon over the original icon:



5. Final icon is ready:



## Operations

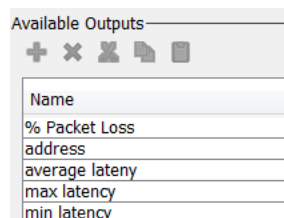
This section contains recommendations for:

- Designing an operation
- Reusing out-of-the-box operations
- Executing against multiple versions of a integration product
- Using standard outputs across OO Content
- Using proper responses and providing meaningful error messages

### Avoid Creating Multiple Operations that Run the Same Command

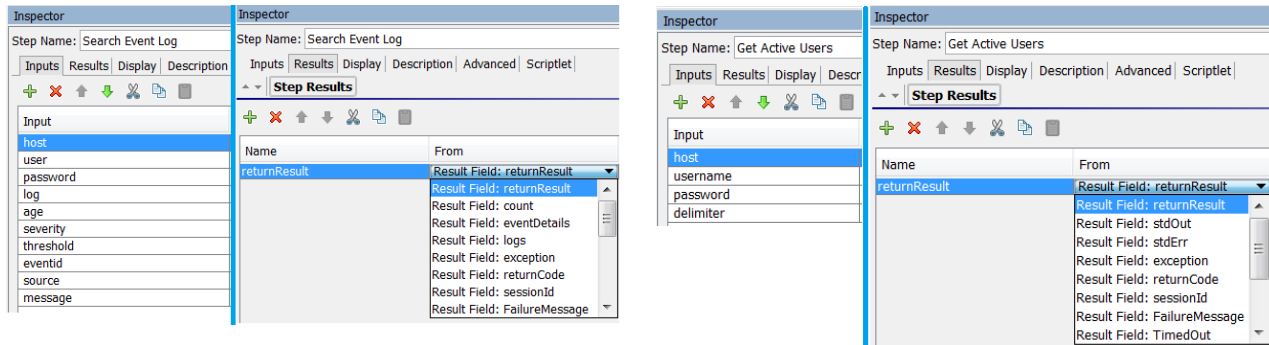
If at all possible, avoid creating multiple operations that run the same command. For example, you can get both packet loss and maximum latency from a Ping operation. It is a best practice to capture both pieces of information in one operation by using multiple outputs of one Ping operation, rather than create multiple operations that use the ping command.

Following are the available outputs for a Ping operation:



Exceptions to this principle are operations that are *extremely generic*, such as an operation that runs a WMI command. It is better to create WMI command operations that are specific to particular functions, instead of a single operation that has a very generic input for the WMI command and very generic outputs.

Example: The “Search Event Log” and “Get Active Users” operations run a WMI command:



## Soft Copies Instead of Hard Copies

Using soft copies ensures *future updates and fixes* in the out-of-the-box content and will be reflected in your flows. For more details on soft copies vs hard copies, refer to “Complete operation fields” in *Operations: Soft VS Hard Copy*.

Each operation belongs to an action plugin jar that is part of a content pack archive. An action plugin is identified by GAV – GroupId, ArtifactId, and Version.

An operation is uniquely identified by the GAV of the plugin and by the *Action Name* (the @Action name), so these fields are mandatory (disabled for editing). For an operation without GAV and Action Name, the OO engine won’t know which code to execute.

The *Group Alias* lets you separate between assigning an operation to a RAS during authoring time and the runtime environment.

When providing an *Override Group*, the flow author can set the RAS by entering a different group if needed to override the current one. More details, can be found in “Configuring Group Aliases” in the *Studio Guide*.

The following image shows an example with all Operation fields completed:

Operation fields	
Group Id:	com.hp.oo
Artifact Id:	oo-http-client-plugin
Version:	1.5.0-SNAPSHOT
Action Name:	Http Client
Group Alias:	/Base [1.5.0-SNAPSHOT]/Configuration/Group Aliases/RAS_Operator_Path
Override Group:	\${overrideJRAS}

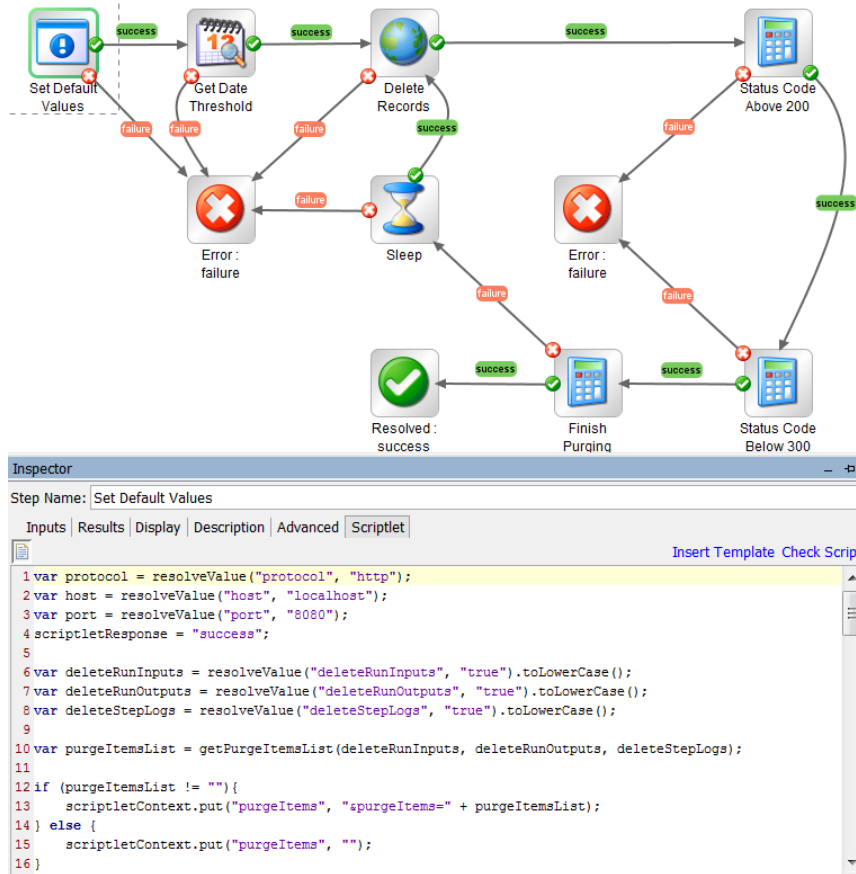
## JavaScript “Do Nothing” Operation

The “Do Nothing” operation is recommended when JavaScript is needed *to create content* or *to extend the existent functionality*. JavaScript provides a powerful engine for manipulating flow variables and to hide complex logic in one operation.

This operation can be used in following cases:

- To manipulate a flow variable. For example, if you want to filter a flow variable, define the input in a “Do Nothing” operation and filter it in Results tab.
- To create new light operations, such as: operations for strings, and date/time manipulation.
- As a step in a flow for input validation.

In the following example, the first step, “Set Default Values” is really a “Do Nothing with Two Responses”, used for validating flow inputs and setting default values:

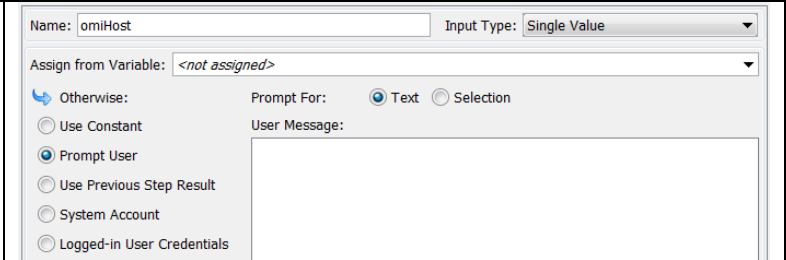
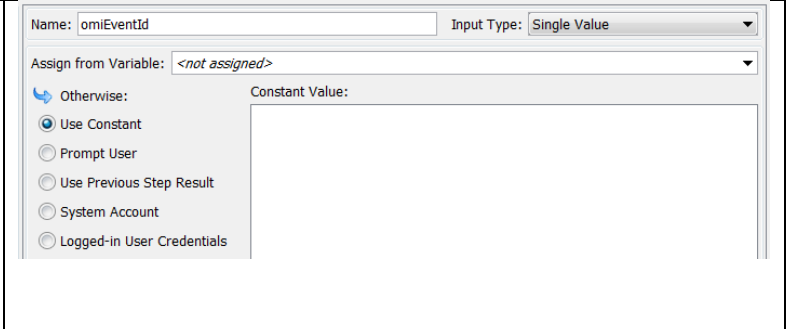


## Inputs

### Assignment

The assignment controls how values are exchanged between steps and is different for required and non-required inputs as follows:

#### Required Inputs:

<p>Assign from Variable: &lt;not assign&gt;          Otherwise: Prompt User          Assign to Variable: &lt;not assign&gt;</p>	
<b>Non-Required Inputs</b>	
<p>Assign from Variable: &lt;not assign&gt;          Otherwise: Use Constant          Assign to Variable: &lt;not assign&gt;          Constant Value: &lt;default value&gt;</p> <p>Note: In this picture, the <i>omiEventId</i> variable will be initialized with "" (empty) because the Constant Value is "" (empty).</p>	

When you create a new input for an operation, flow, or step, the default value for *Assign From/Assign To* is automatically set to <not-assigned> when installing Studio10.x.

This functionality can be changed using the properties:

*dharmastudio.ui.inputinspector.assignfrom.selected* or  
*dharmastudio.ui.inputinspector.assignto.selected*

in the <OO Install Path>/Studio/conf/studio.properties file with the following possible values:

*assigned* - default when upgrading from an earlier Studio version

*not assigned* - default when installing Studio 10.x

These properties allow you to set the initial option in the "Assign from Variable"/"Assign to Variable" combo-box, in the Input Inspector.

### Handling Multiple Versions of the Same Product

Sometimes, you must execute operations against multiple versions of a target product.

There are products such as Operations Orchestration or Microsoft Hyper-V that provide APIs for server version detection. This means that for every execution, the operation should determine the version of the integration product. Because this may create performance issues you should consider *adding a version input*. If specified, its value takes precedence over dynamic version detection.

### Handling Versions

Off-premise applications (such as those from different Cloud providers like Amazon and OpenStack) usually expose the REST API version, while on premise applications expose *product version*. The advantage of using an API version is that it may change less often than the actual product version.

Example: 1.0, 2.0 - for OpenStack, Windows Server 2012 - for Microsoft Windows product

If *the target product supports only one version* and the user provides a value greater than the supported one, the code should try to use the single supported version.

Example: If version 6.0 is latest supported for RHEL and the user enters “6.2” for the “version” input, the code will use the 6.0 version.

In case *the target product supports multiple versions*, the default value for the “version” input should be the lowest one.

Example: If a product supported both 5.5 and 6.0 versions, the “5.5” should be the default version.

## Timeouts

There are at least 3 types of timeout inputs: *connectTimeout*, *socketTimeout*, and *executionTimeout*. It is recommended that you to expose these inputs because it is important in high load scenarios.

The *connectTimeout* represents the time to wait for a connection to be established, in seconds. A timeout value of '0' represents an infinite timeout.

The *socketTimeout* represents the timeout for waiting for data (a maximum period of inactivity between two consecutive data packets), in seconds. A *socketTimeout* value of '0' represents an infinite timeout.

The *executionTimeout* represents the time to wait for an operation to complete.

## Outputs

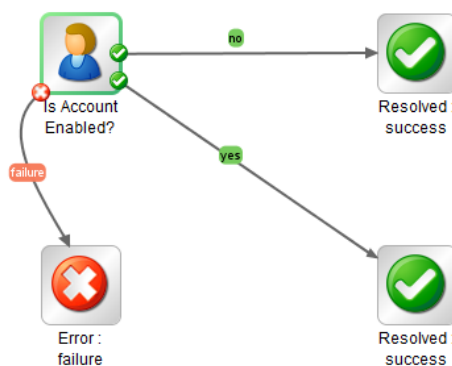
When developing new operations, you should use consistent and common naming for outputs.

Examples: *returnResult*, *returnCode*, *exception*

## Responses

You should use *common responses* such as “success”, “failure”, or “has next” to ensure a common look-and-feel.

The *decision operations* should have one success response for True, one success response for False and one failure response. This is because there should be no difference between and “false” and a “true” response.



Example: “Is Account Enabled?” should have one success response for True, one success response for False and one failure response.



The default response for an operation should be “*failure*” - use “*failure*” response as *Default* and *On-Fail*. This way, an incomplete operation shows as a failure during flow debugging and points the author to the problem before the flow goes into production.

Response	Default	On-Fail	Type	Rules
success	<input type="checkbox"/>	<input type="checkbox"/>	✓ + - ✗	1 Rule [Source: returnCode, No Filters, Compare = 0]
failure	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	✓ + - ✗	1 Rule [Source: returnCode, No Filters, Compare = -1]

## Error Messages

For troubleshooting, it is important to provide meaningful error messages.

Ensure that all error messages contain:

- The invalid value provided
- The input name
- Valid values

Example: “Invalid value “10.10.10.257” for input “targetHost”. Valid values: IPv4, IPv6 numeric format or fully qualified domain name (FQDN) format.”

Also, for troubleshooting, it is important to provide *the exception* and *the code returned by the server* and not hide the exceptions that are provided by the integration product.

Example: WMI based operations should not hide the Microsoft standard HRESULT Error Constants such as 2147749891 (0x80041003) - WBEM\_E\_ACCESS\_DENIED which means that current user does not have permission to perform the action.

## Scriptlets

An OO Scriptlet is a JavaScript (Rhino) script used for manipulating flow data.

### Coding Style

#### Long Scriptlets

If possible, *avoid long scriptlets*. When trying to save a flow that contains scriptlets exceeding 64kb, an exception will occur. Usually, scriptlets should not contain complex logic. Use scriptlets for parsing strings as well (the following image shows an example).

```
function addParamToCommand(commandString, paramName, paramValueFlag) {
    if ((paramValueFlag != null) && (paramValueFlag.toLowerCase() == "true")) {
        commandString += " -" + paramName;
    }
    return commandString;
}
```

#### Validate Scriptlets

When saving a flow that has a scriptlet, OO Studio automatically performs a scriptlet validation, checking the script for syntax errors. This validation can also be triggered manually by clicking the “Check Script” button from the Scriptlet tab.

## JavaScript Best Practices and Scriptlets

It is recommended that you apply JavaScript best practices to OO scriptlets, as follows:

- In JavaScript, function or variable names must begin with a letter, an underscore (`_`), or a dollar sign (`$`). Subsequent characters can be letters, digits, underscores, or dollar signs.

Note: If the inputs of flows or steps are referenced in scriptlets, they should also follow these guidelines.

Example: If an input name is “input.1”, it can’t be used in a scriptlet because it’s an invalid JavaScript variable name, even though it is a valid name for an OO input.

- Always use “var” when declaring scriptlet variables – Variables in JavaScript can have *global* or *function* scope, so prefixing the variable declaration with “var” is especially important in the function scope.
- Always use curly brackets when writing an *if/for/while* clauses even if the statement’s scope is composed of only one operation.
- Always use semicolons (;) at the end of each statement. Consider the following example that could lead to errors:

```
x
++
y
```

This example is parsed as `x; ++y;` not as `x++; y.`

- Use brackets to ensure that operations are executed in the desired order.

```
Examples: var a = null;
          var b = "a==null is: " + a == null;    // false

          var a = null;
          var b = "a==null is: " + (a == null);  // a == null is: true
```

- Use the unary “+” operator for addition, concatenation and converting a String to Number when used as a prefix. For more complex conversions, like conversions with radix, use `parseInt()` or `parseFloat()` functions.

```
Examples: 1 + 2 + "4" + 1 + 2;    // 3412
          1 + 2 + (+ "4") + 1 + 2; // 10
          1 + 2 + "4" + (1 + 2);  // 343
```

The functions `parseInt()` or `parseFloat()` can also be used to convert a *String* to *Number*, but these functions can sometimes “guess” the result. For `parseInt()` the “radix” parameter should always be specified to avoid confusion and to obtain a number in the desired numeric base.

```
Examples: +"";                // 0
          parseInt("",10);     // NaN
          +"45abc";           // NaN
          +parseInt("45abc");  // 45
          +"2.3";             // 2.3
          parseInt("2.3");    // 2
          +"2e3";             // 2000
          parseInt("2e3",10); // 2
          parseFloat("2e3");   // 2000
```

- Avoid using “eval” – The `eval()` function provides a way to run arbitrary code at run time so the compiler can't optimize it and might eventually introduce a potential security risk.
- Use `{}` instead of `new Object()` and `[]` of `new Array()`.
- Avoid using the “with” statement because it has been deprecated for security reasons.

## Exception Handling

The exception handling is important *whenever you aren't in direct control of the inputs to the scriptlet*. For example, if you are filtering the output of a REST call (that might fail) in a scriptlet you should protect the initial parsing of such inputs in a try/catch block as the REST call might fail and yield an odd or unexpected input.

Another important exception usage is for providing *meaningful error messages*:

“Invalid format for input ”jsonText”. The valid format is JSON.”

The previous example is more intuitive for a non-programmer rather a Java-specific exception.

How to handle exceptions:

- The `try/catch/finally` statement is the exception handling mechanism of the scriptlet:
  - The try statement allows you to test a block of code for errors
  - The catch statement allows you to handle the error
  - The finally statement enables you to execute code, after try and catch, regardless of the result. This statement is optional.

Example:

```

try {
    Block of code to try
}
catch(err) {
    Block of code to handle errors
}
finally {
    Block of code to be executed regardless of the try / catch result
}

```

- The `throw` statement allows you to create custom errors. This statement shouldn't be used to throw errors from a scriptlet and all the errors that are encountered in a scriptlet should be handled by the scriptlet itself.
- Always use the `try/catch` statement when writing code that is advanced or that uses Java classes.
- In cases where an error is encountered in the scriptlet, the error message should also be placed in the `scriptletResult`. At the step level, this could be defined as a result variable (for example, `errorMsg`) that uses this result. The reason is that the Primary Result is easy to see in logs and in debugging mode, compared to setting a variable in the `scriptletContext` (e.g. `scriptletContext.put("errorMsg", "Some error text here")`).

## Handling Scriptlets in OO

Scriptlets can be used in following cases:

- On the *Scriptlet* tab of the operation, flow or step. In this case, the scriptlet runs after the execution of the operation, flow or step. Its main goal is to filter or run some logic on the operation/flow/step results. In case the scriptlet is set on an operation and on a step based on that operation, both scriptlets are executed in the following order: first the scriptlet declared on the operation and then the scriptlet declared at the flow step level.

- The scriptlet filter can be applied on *flow results* and *step results*. In this case, the scriptlet is used for filtering or manipulating results. The scriptlet filters should be used only if the task can't be accomplished with a simpler (predefined) filter or set of filters.

## OO Special Objects and Methods

The following table includes the most typical objects and methods used in OO scriptlets:

OO scriptlets objects and methods	Description
<code>scriptletContext.get("myContextKey")</code>	Gets the value associated with the context key <code>myContextKey</code>
<code>scriptletContext.put("LocalVariable", "LocalValue")</code>	Puts data into the local context, so it will be available in the current flow, but not to its parent flow or other subflows
<code>scriptletContext.putGlobal("GlobalVariable", "GlobalValue");</code>	Has the same function as <code>scriptletContext.put()</code> , but for the global context
<code>scriptletRawResult['resultName']</code>	Used for accessing the output from the operation
<code>scriptletResponse = "success"</code>	Used for setting the response of the operation (must match one of the responses from the Responses tab)
<code>scriptletResult = "Your Result Here"</code>	Used for setting the result of the operation

Note: More advanced information about this topic can be found in the scriptlet template in OO. You can add the scriptlet template in a Scriptlet tab by clicking the "Insert Template" button.

## Scriptlet Capabilities

Scriptlets have direct access to inputs of the operation/flow/flow step through their name and a variable that is created for each input. Scriptlets can't change input values; they can only read them.

Scriptlets allow for the creation of local/global variables and can store values that can be used in other steps in the flow (through the *scriptletContext*):

- Local data is available only to the current flow (not subflows or a parent flow).
- Globally add or change system properties (through *scriptletContext.putGlobal*). The data is merged to the global parameters only after subflow execution is complete and when two subflows change the global parameters, the last flow to complete will override the value of the parameters.
- A scriptlet can be extended with classes from JDK, meaning that Java code can be used inside a scriptlet. For creating a Java object the fully qualified class name must be used.

Example: `var javaStringObject = new java.lang.String();`

## Reusing Scriptlets

Scriptlets that are designed to perform a simple and modular task can be reused.

In out-of-the-box content, reusable scriptlets are saved in Configuration/Scriptlets. To use a scriptlet from Configuration/Scriptlets you can drag and drop it in a step's scriptlet tab.

More details about scriptlets can be found in *Studio Guide*, in "Managing Configuration items/Configuring Scriptlets" section.

## Handling Variables

The inputs of the operation/flow/flow step are accessed through their names.

The *scriptletInput* variable is defined only for a scriptlet filter and represents the data to be filtered. The *scriptletInput* and the *scriptletResult* variables are JavaScript strings, so the method or properties on these variables should be called accordingly.

Example: `input1.length` // JavaScript String length property

When using inputs or flow variables they should be accessed through the *scriptletContext*. These are Java specific Strings (of type *Java.lang.String*) so the methods on these variables should be called accordingly.

Example: `var input1 = scriptletContext.get("input1");  
input1.length();`

It is recommended that all the variables that come from outside of the script should be first validated. For example, if a non-existing variable is accessed by *scriptletContext.get()*, then the call will return a *null* value and if it's accessed by its name directly, the scriptlet will fail at runtime.

Example: `var input1 = scriptletContext.get("input1");  
if (input1 != null && input1.length() > 0){...  
}`

Note: `null == undefined; null !== undefined;  
typeof(null) => Object  
typeof(undefined) => undefined`

To access these variables in an easy and reusable way, a simple function should be defined:

Example: `function getValue(inputName) {  
 var inputValue = scriptletContext.get(inputName);  
 if (inputValue != null && inputValue.length() > 0) {  
 return inputValue;  
 }  
 return null;  
}`

Then the function could easily be called for every input:

Example: `var a = getValue("input1");`

When assigning a value to a flow variable through the *scriptletContext* or setting the *scriptletResult*, the value will be converted to JavaScript string.

It is recommended to always set the *scriptletResponse* and *scriptletResult* so that the primary result always has a defined value.

## Content Pack Versioning

The version format used for Content Packs and Projects must be consistent, as follows:

**Versioning: X.Y.Z[-SNAPSHOT ] - Major.Minor.Micro[-SNAPSHOT]**

### Major

- Significant improvements with considerable business implication
- Major changes that break backward compatibility
- Move content, remove content, major redesign

### Minor

New flows or operations on existing content

### Micro

Solely bug fixing to existing content

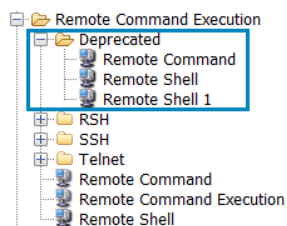
### SNAPSHOT

A content pack with SNAPSHOT version means that it is still under development. A version without SNAPSHOT means that the content pack was released, has been tested and is stable. Uses released versions in a production environment.

## Deprecation

### Avoid Using Content from *Deprecated* Folders

Content from deprecated folders is considered outdated and could be removed in subsequent versions.



### When to Deprecate Content

The operation or flow must be deprecated *when changes to it can break the backward compatibility*, such as:

- Changing non-required inputs to required inputs
- Renaming or removing inputs
- Changing results
- Changing responses

## How to Deprecate Content

To deprecate content, the original version of an operation or flow must be moved to a “Deprecated” folder in the Library tree and then create a new version, usually with the same name, in the original location (because creating a flow or operation with the same name but different UUID is not possible).

## References

- *HP Live Network*: <https://hpln.hp.com>
- *Studio Guide*: <https://hpln.hp.com/node/22004/attachment>
- *Concepts Guide*: <https://hpln.hp.com/node/25861/attachment>
- *Best practices for OO content structuring*: <https://hpln.hp.com/node/20772/attachment>
- *Operations: Soft VS Hard Copy*: <https://hpln.hp.com/node/15963/attachment>
- *The OO 10 SDK - Getting Started*: <https://hpln.hp.com/node/15705/attachment>
- *Customizing Flow Transitions*: <https://hpln.hp.com/node/26352/attachment>

If you have any questions or feedback then please post these on the OO community forum:  
<http://www.hp.com/go/OOPractitionerForum>