vRealize Orchestrator Coding Design Guide

Development Guide

TECHNICAL WHITE PAPER

MARCH 2016

VERSION 1.0

Table of Contents

Revision History

DATE	VERSION	DESCRIPTION
March 2016	1.0	Initial version.

Introduction

This document describes the best practices for using vRealize Orchestrator and its design, development, and configuration functionality. The depth of coding design presented in this document is from a macro level, which ensures that the design fits well within the greater architecture of products used, down to the micro level that includes JavaScript coding guidelines to be used in vRealize Orchestrator. A full life cycle of vRealize Orchestrator development involves a multi-phased approach that includes design, development, testing, and release best practices.

A clean code development approach was applied to vRealize Orchestrator development. This approach can be applied to any programming language or product to help create well-written code. Many of the base concepts that are discussed in the book *Clean Code* by Robert C. Martin are used in this document. It is recommended that you familiarize yourself with the content in that book. For more book recommendations, see Recommended Reading.

Design

You should understand the different components and their interactions to design an overall solution. This section provides guidance with different design process approaches and discusses several methods that you can consider for the overall design as it transitions from the solution, to the subcomponents, to the individual workflow.

Solution Design

You should design the overall solution by using vRealize Orchestrator workflows as drop-in placeholders for the necessary tasks and to use as stubs in the design process. You can use this approach to identify related tasks that can be pulled out into separate, self-contained workflows or actions that can help you identify how much time to budget for each high-level task. During the solution design, you should test the aggregate workflows for completion to achieve the desired use cases from the solution. Each use case must be supported by one master workflow, which relies on subsequent workflows to achieve the desired outcomes. Similarly, each component's workflow, action, or resource, contributes a specific well-defined function that supports one or more master workflows to deliver desired use cases. During the design phase, a clear distinction is made between automation and orchestration workflows.

- **Automation** is achieved by an individual workflow, action, or resource to accomplish a discrete task.
- **Orchestration** is achieved by a Master Workflow that integrates and coordinates execution of such automation components to deliver results required for a specific use case.

Master workflow design must encompass and deliver on the full lifecycle considerations, including use cases for provisioning, reconfiguring, and retiring resources in the VMware ecosystem. Additionally, the master workflow should be designed for availability with validation performed between each step to enable pause, rollback, or failure functionality.

This document uses many development details targeted at delivering a better solution, but a key guiding principle to the overall design is that it should be done purposefully and with clear and expected outcomes between the different components. This methodology is referred to as design by contract. For more information, see the Wikipedia definition of design by contract. Although the solution for a single use case is often designed and developed by a single person, if you considered that the various components of your solution could be developed by others, then an agreement would be necessary between all of the developers with respect to what the expected inputs and outputs are to be, and what type of error handling should be covered in each component. For example, if one of your actions is to provide an array of objects to calling workflows, validating the code before sending it back can prevent the callers from validating it themselves, such as not-null or not-empty checks, which results in precarious exceptions in the system elsewhere if the callers did not remember to explicitly perform those checks. Additionally, you should use the **Description** fields of the workflows or actions to document the behaviors of these edge cases or error conditions.

Bottom Up Approach

To achieve the optimal design and clean code, consider how the components should be designed to work with each other if the development is done from the bottom up, which is the recommended development approach. The design of the solution will become clearer by constructing the fundamental building blocks for your solution, which are typically the actions, which convert to the mid-level workflows, and finally are the main workflows that are the entry points to your overall solution.

For example, if the goal is to integrate vCloud Director with a third-party SOAP system for a given VM life cycle event, you should use the following guidelines:

- Potentially use a separate SOAP tool to achieve some communication with the third-party system.
- Create a workflow in vRealize Orchestrator that does a hard-coded SOAP call to the third-party system to validate the communication and protocol compatibility.
- Pick out elements of the hard-coded call and make them variables and inputs into the workflow, such as the SOAP operation and the XML body.
- Create a matching action which takes the same inputs and have your workflow call this action. The workflow itself now became the basis for your Test suite, and should not be called on its own anymore now that the action is available.
- Create the mid-level workflows or actions that provide the business logic to achieve the simple inputs requested by your SOAP action. For example, a mid-level workflow or action can be tailored to a specific SOAP call in the third-party system, and it can take inputs and create the appropriate XML body to paste to the generic SOAP action already created.
- Create a workflow that takes a set of key-value pairs (as a Properties object) and calls your mid-level workflows. This workflow should still not mention vCloud Director.
- Create a workflow which is specific to vCloud Director that takes on vCloud Director objects and converts them to simpler objects that can call the main system-agnostic workflow that you created.

Workflow Separation

Separate workflows into sets of smaller workflows and actions, preferably so that each one can be run standalone. This prevents global variables, attributes or inputs, of a workflow from being intertwined with most workflow elements, which makes it harder to read, debug issues, and perform testing.

Separating workflows also provides readers the ability to read your code more efficiently and be able to quickly locate a particular use case, knowing that no other aspects are having an impact since it relies on a sub workflow or action.

Additionally, readers can take passes at your code so that someone who understands 40 percent of it can still understand the entire functionality, while someone who understands 90 percent of it understands it in depth. If the workflows are not separated into sub workflows or actions, then someone who has read 40 percent of your code sequentially will only understand 40 percent of the functionality, and would need to read nearly 100 percent of the code before understanding most of the functionality.

The structure of vRealize Orchestrator workflows or actions, and even code itself, is analogous to the structure of a reference book. A book has a table of contents, where you can very quickly see what the book contains, and it has chapters, headings, sub-headings, and finally the words. Each of these components provides a particular level of detail when looked at holistically.

Actions can return only one item, while workflows can return multiple items, which might also influence your design. However, avoid returning more than just a couple of items for workflows and consider breaking out the workflow into separate workflows or actions, or combining the result into a logical object held in a Properties or JavaScript (Any) object if multiple items are to be returned.

Sever the connection between products or interfaces as soon as possible, and only pass around vRealize Orchestrator simple objects where possible. For example, workflows initiated by vRealize Automation tend to interact heavily with

vRealize Automation custom properties. Pull in these key values at the start of the workflow and pass around the attributes into other workflow elements, workflows, or actions, so that any of those can be used separately for another parent orchestrating product, such as vCloud Director or another external system calling into vRealize Orchestrator's REST interface. This also simplifies unit testing, because the inputs to workflows can be easily mocked up.

Individual Workflows

For a single workflow, you should answer the questions presented in this section. Your answers will provide you with an understanding of what the workflow looks like and what attributes you need. Determining what steps are necessary, what information is needed for these steps, and where the information comes from is usually iterative, so you should start with a high-level design that uses the required steps, then determine what information is not available directly, and add another step before that.

What is the Purpose of the Workflow

Specify the tasks the workflow automates, the expected changes the workflow makes in the environment, and the expected output. Also specify the expected circumstances. Your answer to this question becomes a candidate for the Description field of the workflow.

Who Executes the Workflow

Workflows can be executed by human users such as in the vRealize Orchestrator client, vSphere Web Client, vRealize Automation Advanced Service Designer. They can also be executed by a custom portal and the workflow scheduler, policies, external systems by using the API such as vRealize Automation IaaS callouts, or other workflows such as Library workflows.

Depending on your answer to this question, the workflow would have different input parameters and presentation. For example, a workflow designed for human consumption should be as easy to use as possible, so fewer input parameters, and a lot of presentation logic to support the user is desired. To allow flexible reuse in different situations, a library workflow that is being called by other workflows has a lot of input and output parameters and no presentation logic because this is not applied when calling the workflow from other workflows.

What Steps are Necessary

Specify the different steps needed to achieve what the workflow should do. Steps that might be helpful in another context are good candidates to be modularized.

What Information is Required to Complete the Steps

The answer to this question provides an understanding about what attributes are needed for the workflow.

Where is the Information

Information can be hard-coded in workflow attributes or a script (not recommended), read from configuration elements, passed to the workflow as an input parameter, or calculated dynamically by an additional logic step earlier in the workflow.

Third-party Integrations

When designing integrations for third-party systems, it is important to understand the proper integration direction, and the limitations of the external systems will dictate your development as well. For example, scalability limitations of the system vRealize Orchestrator integrating with running many concurrent workflows within vRealize Orchestrator is not a problem but does the third-party system support concurrent requests?

Northbound Integration

An external system calls vRealize Automation or vRealize Orchestrator, and likely controls the outcome of the workflow. Example use cases:

- External portal, like Service Now, or an existing customer portal calls vRealize Automation
- Based on certain tickets in a helpdesk system, vRealize Orchestrator workflows should be executed
- vCloud Director blocks tasks executing vRealize Orchestrator workflows
- Monitoring system or vRealize Operations Manager calls vRealize Orchestrator workflows for automatic remediation.
- To resume vRealize Orchestrator workflow after calling out to the external system, southbound integration asynchronously.

To implement such integrations, different strategies are possible:

- Using the northbound REST API of vRealize Orchestrator or vRealize Automation, the external system has to support making REST calls
- Using cloud client, the external system has to support running CLI commands
- Using vRealize Orchestrator plug-ins that provide trigger in combination with policies, SNMP, AMQP, and vCenter plug-ins provide triggers
- "Passive" integration: Implement a polling strategy into the external system to pick up work using vRealize Orchestrator workflows, for instance using the workflow scheduler.

Southbound Integration

In this case vRealize Orchestrator workflows call out the external systems to automate tasks within those systems. Example use cases:

- As part of vRealize Automation lifecycle: Get an IP address from IPAM system, make some changes in Active Directory, setup load balancer and firewall rules, create backup jobs, CRUD (Create, Read, Update, Delete) entries in a CMDB (Change Management Database), include system in a monitoring solution, and so on.
- Automation of hardware components like storage, network, for non-VM lifecycle related tasks. Very often in combination with exposing these workflows to vRealize Automation portal by using Advanced Service Designer
- Automation of day to day tasks in vSphere administration: Reporting, Snapshot handling, migrations
- Using vRealize Orchestrator to “abstract” legacy systems and provide automation capabilities for them. One real world example included a very old CMDB created in-house that did not provide proper “modern” web service API via REST or SOAP, but only a limited command line interface in terms of scalability and reliability. Building a set of vRealize Orchestrator workflows for common CMDB automation tasks allowed to use input validation, load limitation and error handling within the workflows, so that other external systems can now use the vRealize Orchestrator REST API to call this specific set of CMDB workflows.

Development

You should use the guidelines in this section as you develop your code.

Clean Up Your Code

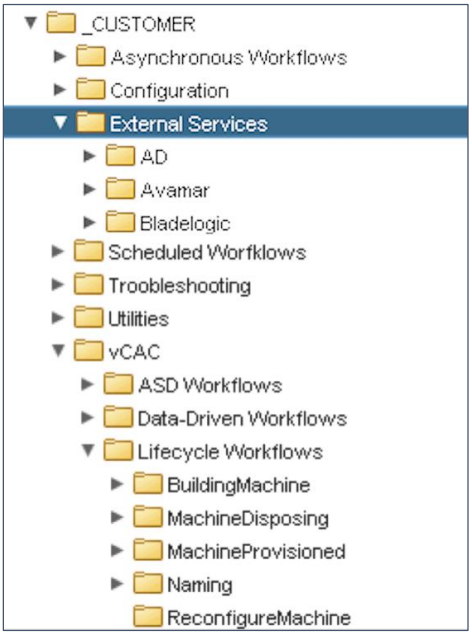
You should strive to clean up your code as you develop, rather than waiting until the end, for the following reasons:

- Waiting until the end to clean up your code is dangerous to the overall functionality that you worked hard to achieve.
- Five days of developing clean code and workflows produces much cleaner results than 4.5 days of coding plus 0.5 days of cleanup.
- You often will not have time for cleanup if that is the only item remaining. If you spent 4.5 days developing clean code, you still have 0.5 days of development time that could be used towards an actual portion of the use case so that it cannot be ignored.
- You cannot provide reusable unit tests since your code is unlikely to be cleaned up to a modular format.
- You cannot valuably engage someone in a code review at the very end of your development as there is too much risk to the overall functionality if any larger changes are proposed, and you might be hesitant or defensive about making those changes because of how daunting the task is when it occurs at the end.

Similar to the way that nothing compares to coding because it can be challenging and it is fun to overcome the challenges, developing clean code provides another opportunity to choose the paths that are more ideal than others. The results are innumerable, yet the journey is very gratifying.

Workflow Folders

Organize workflows into a customer-specific folder for all work done for the customer, and a common folder for work that can be reused without any changes. Create wrappers around library workflows or actions, if you want them to fit your needs better and avoid copying them.



FOLDER	DESCRIPTION
_CUSTOMER	Contains all development related to the customer.
vCAC	Contains all workflows related to vCAC / vRA (and is actually called by vCAC / vRA).
ASD Workflows	Contains all workflows launched in vCAC / vRA by using ASD.
Lifecycle Workflows	Contains one folder per state. Each workflow that is launched at one state is store in its state folder. If a workflow is launched at multiple states, it is stored in a StateMultiple folder.
Scheduled Workflows	Contains all workflows that are scheduled.
External Services	Contains all workflows related to external systems that are used by workflows in Lifecycle Workflows, Scheduled Workflows, ASD Workflows, and so on.
Helpers	Place sub-workflows into a Helpers folder. These workflows are not to be executed by users. These workflows might not even be executable on their own, for example, if they have a Properties input. Potentially separate sub-workflows into additional folders.

Modules or Action Folders

You should keep your actions organized so that they are easy to spot. Create different sets of modules that are partitioned by the common functions, rather than by a customer or a project phase. Common functions can include all sets of communication with an external system.

Naming Guidelines

You should create and follow naming guidelines in order to understand the purpose and use of the workflows and actions by just looking at the name.

Workflows or Actions

- Avoid odd characters, use Latin.
- Names should clearly indicate the use of the workflow, for example DoubleRam.
- Spaces can be used to increase readability for workflows.
- Avoid camelCase for Workflow names.
- Do not name two workflows the same name, even if they are in different folders. They can be differentiated by their unique ID and method signature (parameters), although searching for them and differentiating them simply by the name might become complicated.

Scripting

Use JavaScript conventions for naming. To view a quick set of naming and other JS conventions, see http://www.w3schools.com/js/js_conventions.asp.

- camelCase variables and functions.
- Do not use underscores.

- Do not prefix the name to indicate the type, for example, do not use "iCount" to indicate it is an integer.
- Avoid numbers on variables and clearly name them, for example, do not use vm1 and vm2, but use something similar to vmIn and vmOut.

Configuration Workflow

Provide a configuration workflow to minimize the manual work necessary by the user. For example:

- For vRealize Automation integrations, create Build Profiles programmatically.
- For REST integrations, create a REST host object with all the REST operations already attached. Do the same for SOAP hosts and operations.

Using configuration workflows to initialize or read the vRealize Orchestrator attributes for external systems, including URL for database, Service Accounts, and passwords stored in secure strings. Occasionally, orchestration workflow failures originate from the difference in Service Accounts used by vRealize Orchestrator for external systems through configurations, and when the password resets in the external system on a scheduled basis, for example, 45 days on AD. When this occurs, you need to update the configuration attribute in vRealize Orchestrator to restore the orchestration to operational status.

Packaging

Provide packages based on use cases rather than delivery timeline, such as customer.ipam.package and customer.cmdb.package, instead of customer.phase1.package. This results in multiple packages, but it also makes it easier to update the packages in future phases, with less of a chance of changes to previously released use cases. Be careful, however, with any shared items among the packages and consider having a common package.

Update versions in changed workflows or actions so that reimports on another system happen automatically.

Make use of the vRealize Orchestrator plug-in workflows or actions and do not copy out any of its code. With this method, if a vRealize Orchestrator plug-in changes then your code can reap the benefits such as bug fixes, changes to the design, and so on, and you can open a service request or bug if there is an issue with any of the code used. For example, with the vSphere plug-in there is an action that checks if a device is a network card (com.vmware.library.vc.vm.network.isSupportedNic), with newer versions of vSphere, newer device types have been introduced.

Do not export workflows or actions that are included in the built-in vRealize Orchestrator library or a prerequisite to the package solution, such as a plug-in being installed as it is unnecessary bulk. Check and clean up the package before you export it, as additions of a workflow or workflow folder can cause vRealize Orchestrator to add all used items to the package, even if the source of the item is delivered by vRealize Orchestrator.

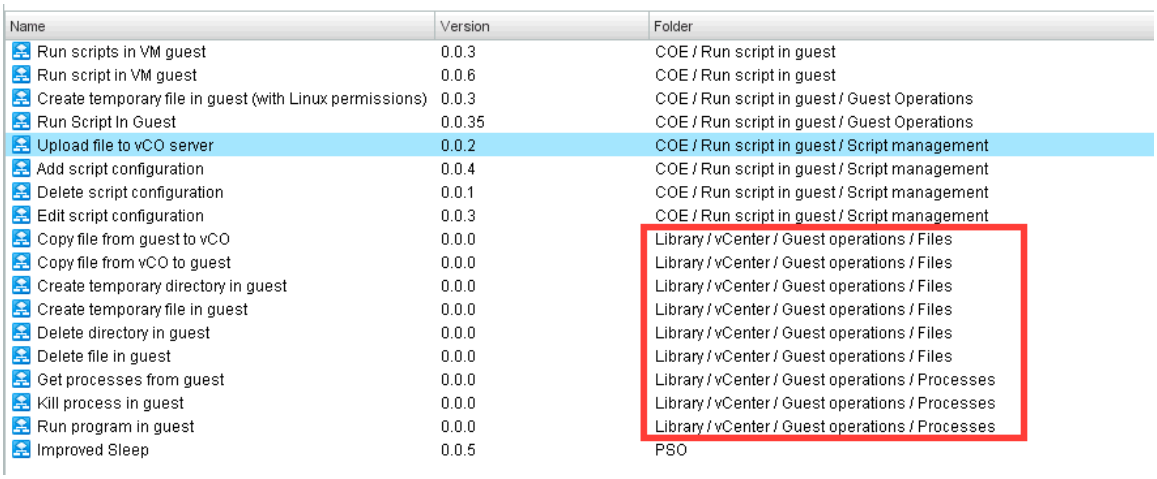
Actions are usually named Server.getModule("action.package").action(param1). In Orchestrator versions which are not the latest, the action might not be automatically discovered when adding a wrapper workflow or action to a package, so these actions need to be added manually to a package. When actions are called differently than the method shown above, they are not automatically discovered and should be added manually to the package.

Include two sets of packages, an editable one and one that is read-only to be used by the client in production.

Provide a separate package for your unit tests. For more information, see the Unit Tests section.

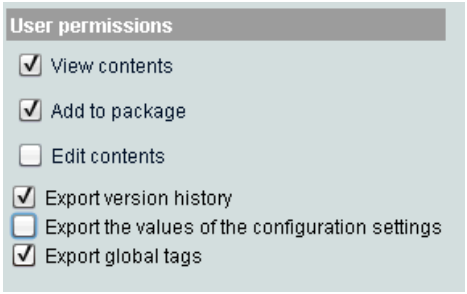
How to Clean Up the Package

1. Order by Folder name.
2. Remove all workflows under **Library**.



Name	Version	Folder
Run scripts in VM guest	0.0.3	COE / Run script in guest
Run script in VM guest	0.0.6	COE / Run script in guest
Create temporary file in guest (with Linux permissions)	0.0.3	COE / Run script in guest / Guest Operations
Run Script In Guest	0.0.35	COE / Run script in guest / Guest Operations
Upload file to vCO server	0.0.2	COE / Run script in guest / Script management
Add script configuration	0.0.4	COE / Run script in guest / Script management
Delete script configuration	0.0.1	COE / Run script in guest / Script management
Edit script configuration	0.0.3	COE / Run script in guest / Script management
Copy file from guest to vCO	0.0.0	Library / vCenter / Guest operations / Files
Copy file from vCO to guest	0.0.0	Library / vCenter / Guest operations / Files
Create temporary directory in guest	0.0.0	Library / vCenter / Guest operations / Files
Create temporary file in guest	0.0.0	Library / vCenter / Guest operations / Files
Delete directory in guest	0.0.0	Library / vCenter / Guest operations / Files
Delete file in guest	0.0.0	Library / vCenter / Guest operations / Files
Get processes from guest	0.0.0	Library / vCenter / Guest operations / Processes
Kill process in guest	0.0.0	Library / vCenter / Guest operations / Processes
Run program in guest	0.0.0	Library / vCenter / Guest operations / Processes
Improved Sleep	0.0.5	PSO

How to Export the Read-Only Package



Workflow Internals

Reduce the complexity of your workflows by considering the following set of guidelines.

Readability

Use the following guidelines to provide clear regions, sections, and steps in your workflows and try to keep these very small and have them call separate workflows or actions.

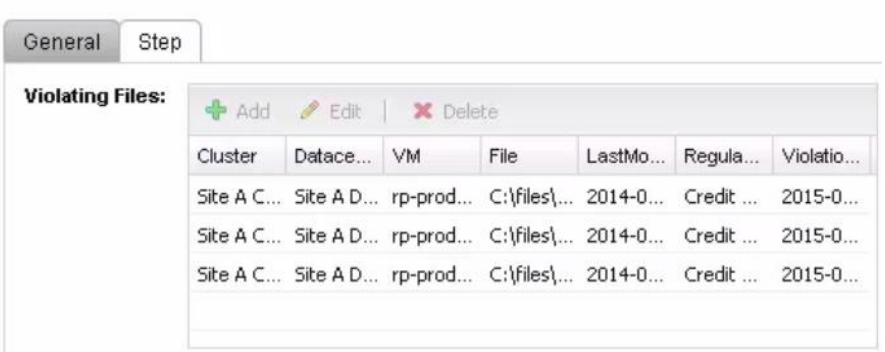
- Use Workflow Notes to enclose regions.
  This helps differentiate the workflow regions by providing different shading background for each region, and make reading the workflow easier.
- Keep workflows to a max of 10-15 workflow elements.
  Overcrowding of the schema of a workflow is a clear sign for moving some of the functionality into sub-workflows or actions such that the overall orchestration of the workflow is clean and clear to understand.

- Try to achieve modularity with the workflow separation, but consider the complexity of the overall solution. Sometimes you may need to sacrifice some modularity to improve readability. Do not try too many tricks just to ensure modularity, as it is difficult for the average vRealize Orchestrator developer or support staff to understand.
- Rename the workflow elements. If you cannot figure out a name that clearly captures everything very succinctly, you likely have too many things happening in a single element. Try to use the workflow element for an action so that it is clear when an action is used rather than having it in a Scriptable Task.
- Limit having multiple actions in a Scriptable Task as it is not obvious, and vRealize Orchestrator cannot resume the Scriptable Task in a very clean checkpoint. For more information, see Scriptable Tasks and Actions.
- Loop through objects using workflow elements and counters, so that vRealize Orchestrator can resume from the correct point if there is a failure. For example, if you have a list of 10 VMs and you processed 4 but vRealize Orchestrator shut down during the processing of the 5th, then it restarts at the 5th instead of at the beginning.

Attributes

Use the following guidelines to ensure that you make the best use of attributes in a workflow.

- Lock attributes that are being used as constants (checkmark in the Lock column).
- Attributes should never need to be updated or changed by a user. Either make them an Input, or link them to a configuration element that is a more natural item for a user to update as part of the installation of the workflows.
- If you have a lot of attributes and inputs in a workflow, you might have an issue with your code and workflow structure, which might be caused due to the following reasons:
    - You are doing too many things in one workflow. The use case of the workflow should not be "do it all" and it should be split into sub workflows to handle the different steps, each with clearer inputs and outputs.
    - You are not working with objects where you should be. Create Properties for attributes where you do not need inputs from the user or CompositeType objects for properties objects used for user input. This is especially important if you have arrays of attributes which are very much linked together, for example, systemHostname[], systemUsername[], systemPassword[]… and you end up matching up the various indexes. You can also use JavaScript objects and pass them around as Any objects, but the structure is more expected when it is a properties object rather than a standalone Javascript object.
        - Example of a change to objects: blueCatHostname, blueCatPort, blueCatUsername, blueCatPassword, and so on -> blueCat.Hostname, blueCat.Port, blueCat.Username, blueCat.Password, and so on. This reduces attributes from 4 to 1 and makes everything a lot clearer.
    - CompositeTypes can be consumed by vRealize Automation 6.1 with Advanced Service Designer in a grid view which is convenient.



Scriptable Tasks and Actions

Most scriptable tasks and actions are called from workflows that use them for accomplishing discreet functionality. You can think of scriptable tasks and actions as functions that are called from an Object Oriented perspective. Due to their discreet nature, separate scriptable tasks or actions so that they have a clear single responsibility rather than overloaded with multiple objectives. For more information about single responsibility, see the Wikipedia definition for single responsibility principle.

Create actions when the scriptable task would or could be reused. Try to encapsulate all communication with an external system through one base action that can provide the basic error handling, such as the error message retrieval, and other actions can build off this to provide the particular system functionality. For example:

1. a base action can be used to handle communication with a REST system, taking as input the operation, inputs, and body. This action would make the calls and retrieve the result set, parse the results into JSON format, retrieve the error messages from a JSON format, and it could throw a well formatted exception
2. Another action calls the base one with the parameters specific to its use case, dealing with any exceptions thrown, if necessary.

Keep every block of code between 5-10 lines, though the overall script / action would have more. In order to achieve this think about layers of detail in the development and separate out each layer into a different set of JavaScript functions. With this method, the top 5-10 lines would be all that is necessary for someone to read in a 100+ line script/action and would provide the reader a sort of Table-of-Contents to the rest of the code. If the reader cared for any detail they can jump to the first-level function, like jumping to a chapter in a book, where there may be very little code other than the call of lower-level functions. This format is analogous to skipping around in the book based on the chapter's headings and sub-headings, until you finally reach the block of text which has the details. With this method, you do not force developers to have to read all the detail in order to understand the overall functionality. One way to clearly identify that you are providing very readable code to others is by providing an initial set of lines before a commented-out divider – readers can then understand that reading below the line simply provides more details to the overall functionality shown in those first few lines.

<table>
<tr><td>SPAGHETTI CODE</td><td>Single block of code which constantly shifts focus from high-level objective to low-level detail. Particular issues in the following block:
• inconsistent and/or duplicate checks for value-set conditions (ex: apiVersion, vmName, DNSSuffix)
• difficult to tell which variables are important and which are transient (ex: apiVersion only used temporarily)</td></tr>
<tr><td></td><td><pre>
var apiVersion = "";
apiVersion = "VirtualMachinePropertiesIn.get("External.IPAM.Version");
if (apiVersion != "" && apiVersion != null){
   infobloxAPIVersion = apiVersion
} else {
   infobloxAPIVersion = "v1.0";
}
System.log("Infoblox REST API Version: " + infobloxAPIVersion);
var vm = JSPN.parse(VirtualMachine, null);
vmName = vm.virtualMachineName;
if (vmName == "" || vmName == null) {
   throw ("Virtual Machine name cannot be empty");
}
System.log("Virtual machine name: " + vmName);
var DNSSuffix =
VirtualMachinePropertiesIn.get("VirtualMachine.Network0.DNSSuffix");
if (DNSSuffix == "" || DNSSuffix == null) {
   throw "property VirtualMachine.Network0.DNSSuffix is not defined";
}
if(DNSSuffix.indexOf(".") != 0) {
   DNSSuffix = "." + DNSSuffix;
}
</pre></td></tr>
</table>

CLEAN CODE	• high level functions called at top; details below • short functions 5-10 lines • function names help document without comments

```
vmName = retrieveVirualMachineName(virtualMachineJSON);
networkDetails = retrieveNetworkDetails(vmProperties);


////////////////// Functions - further detail ////////////////
function retrieveVirtualMachineName(virtualMachineJSON) {
   var vm = JSON.parse(virtualMachineJSON, null);
   vmName = vm.virtualMachineName;
   if (!valueSet(vmName)) {
      throw "Virtual Machine name cannot be empty";
   }
   System.log("Virtual machine name: " + vmName);
   return vmName;
}


function retrieveNetworkDetails(vmProperties) {
   var portGroupName = getPortGroupName();
   var networkToUse = findNetworkInMappings(portGroupName);
   return getNetworkDetails(networkToUse);
}


function getPortGroupName() {
   var portGroupName =
vmProperties.get("VirtualMachine.Network0.Name");
   ...
   return portGroupName;
}
...
```

JavaScript Style

vRealize Orchestrator workflows are scripted using JavaScript. Developing appropriate skill in JavaScript and knowing the conventions for the language will greatly aid in workflow development. For more information on JavaScript coding conventions, see http://javascript.crockford.com/code.html.

If Conditions

Avoid big **If** blocks and many connected **If** conditions. The following are potential ways to avoid big **If** blocks:

- Check related items in a particular If or Switch block. Do not mix with other logical groupings.
- Potentially use switch statements for 3+ conditions.

- Do not have an Else condition for the last portion of a function.

NEEDLESS ELSE	`function getValue(input, expectedValue) {`
	`  if (input != null && input != "") {`
	`  return input;`
	`  } else {`
	`  return "N/A";`
	`  }`
	`}`

PROPER	`function getValue(input, expectedValue) {`
	`  if (input != null && input != "") {`
	`  return input;`
	`  }`
	`  return "N/A";`
	`}`

Use functions when a variable is to be set depending on several conditions. The function block has a name, the function name, so it is clear what is being extracted / set:

POOR EXTRACTION OF INPUT / SETTING OF VARIABLE	`var infobloxAPIVersion;`
	`var apiVersion = vmProperties.get("External.IPAM.Version");`
	`if (apiVersion != "" && apiVersion != null){`
	`  infobloxAPIVersion = apiVersion;`
	`} else {`
	`  infobloxAPIVersion = "v1.0";`
	`}`

FUNCTION HIDES 'IF' DETAILS	`var infobloxAPIVersion = getInfobloxApiVersion(vmProperties);`
	`...`
	`////////// Functions ///////////`
	`function getInfobloxApiVersion(vmProperties) {`
	`  var apiVersion = vmProperties.get("External.IPAM.Version");`
	`  if (apiVersion != "" && apiVersion != null) {`
	`  return apiVersion;`
	`  }`
	`  return "v1.0";`
	`}`

Functions can also be used to improve readability of the **If** clauses. Extract the condition into a function and name the function properly, so that the meaning of the **If** clause is understandable without requiring research into the condition logic.

```
failedWorkflowToken = new Array();
var allWorkflowToken = Server.findAllForType("WorkflowToken");
now = new Date(System.getCurrentTime());
for each (var currentToken in allWorkflowToken) {
   if (isToProcess(currentToken) && isFailedToken(currentToken)) {
      failedWorkflowToken.push(currentToken);
   }
}


function isFailedToken(token) {
   return (token.state == "failed") || (token.state == "canceled");
}


function isToProcess(token) {
   var endDate = token.endDateAsDate;
   if (endDate == null) return false;
   return (endDate.getTime() > onlyCheckTokenNewerAs.getTime());
}
```

Loops

- Avoid big loop blocks
- Indicate the exit condition at the top if possible. A boolean of 'keepRunning' as a while loop's condition is unclear. Make your loop code as clean as possible for the reader to understand.
- If the exit condition is not exceptionally clear, encapsulate it in a function named appropriately.
- If searching through a loop to find an element, break to exit the loop after it is found.
- Use arr.indexOf to find simple objects such as strings and numbers instead of looping through them.
- Use foreach to loop through objects rather than for (var index in objs).
- Consider the scale of the iteration for the loop designed, especially if it will impact overall orchestration if the volume of iterations increases in magnitude. For example, it is often times much more efficient to loop through 2 large arrays sequentially than it is to loop through a smaller array and for each object loop through another array, thereby repeatedly looping through the other array.

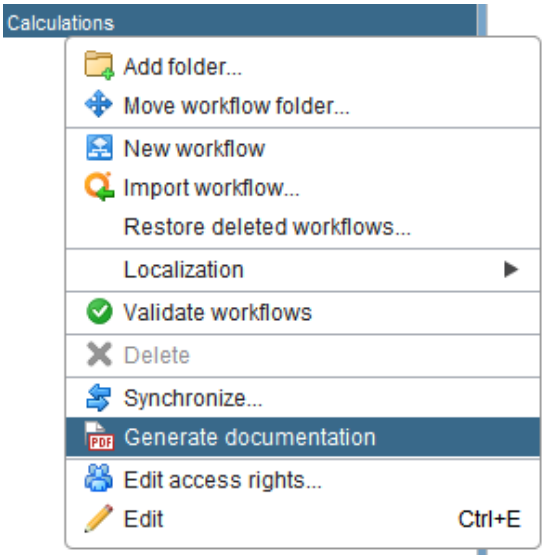
Documentation

Provide a separate document for all of this content, such as a technical guide. At the very least, a detailed functional design specification can help.

Workflow and Action Documentation

Document the workflows and the workflow elements, actions, inputs, outputs, and so on. Use the Description field for most of those elements to provide a clear set of expectations. For inputs, outputs, and attributes which contain a very open ended type like a String, provide examples of acceptable values. For Scriptable tasks, also use the Description field to indicate the task's purpose instead of placing a comment block as the header for the script.

Use the Generate Documentation workflow helper by right-clicking on the workflow or any folder, including the top level folder, to generate a full PDF with documentation.



Comments

Provide comments to indicate some design decision or odd external system behavior and how you needed to compensate for it. Comment WHY you are doing something, rather than WHAT you are doing. The code should clearly explain the WHAT.

Do not comment to make up for confusing code. These types of comments go out of sync with actual code, and cause confusion themselves. Your code can be very clear through the use of multiple, small 5-10 line functions, each of which is well named. The use of well named functions is a form of documentation that does not go out of sync as it is still code and would get changed in the future to reflect its small function or utility.

UI Presentations

Use of actions for presentations should be nearly real time, especially when used by vRealize Automation Advanced Service Designer. There are items that can impede the ability to render the presentation to be real time, including loading of large lists of complex objects in the UI. It is better to pass a smaller object such as an array of VM names and IDs rather than a fully loaded VM object or a list of them in Advanced Service Designer.

Be aware that not all clients, such as vSphere Web Client or vRealize Automation Advanced Service Designer, fully support all properties of the very powerful input presentation capabilities of vRealize Orchestrator. So, be careful when you put mandatory business logic into input presentation logic. Sometimes you must use workarounds, such as data-binding to hidden input parameters, to overcome shortcomings of the calling system.

Configuration Elements

- Do not write in Properties objects into configuration elements, dynamically.
    - If a user has to open and save the configuration element, the Properties object is set to null.
    - Possibly use a CompositeType, or if it is really meant for internal use then you can save the Properties object as a Resource Element. Still be careful to Lock if this is something accessed by multiple workflows.
- Separate between configuration elements that are "general settings / default settings", such as timeout values or polling rates of your solution and elements that contain environment specific details, like hostnames and credentials of external systems being called, inventory folders used for placement.

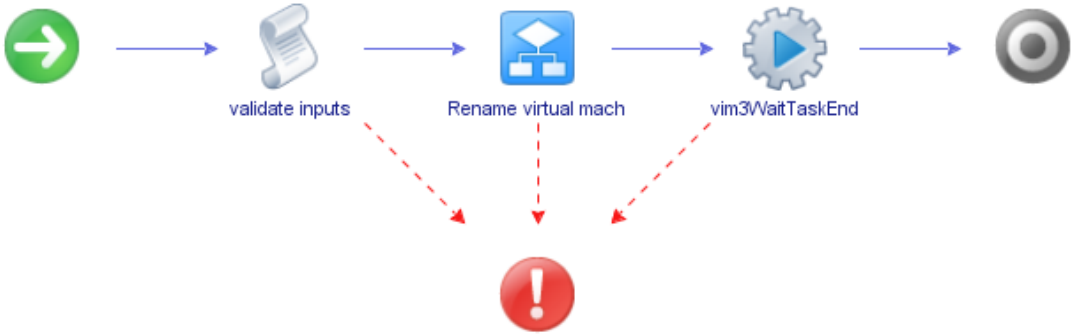
- Be aware about challenges with concurrency when dynamically reading or writing configuration elements from within workflows. Use Resource Elements as a means of programmatic storage.

Error Handling

During the Design phase of the orchestration cycle, Error or Exception handling is a checklist item that is critical to perform. How you handle errors has an impact on the stability, performance, and perception of the entire orchestration and automation system. Use the following guidelines to develop a well-designed error handling design and implementation:

- Do not think of only the "happy path" for the orchestration and automation. Imagine where orchestration might break due to user, system, or third-party integration errors. The error handling implementation is your insurance for when such events occur, and allows you to recover gracefully and restore service in an SLA.
- Unit test incorrect inputs for workflows, scriptable tasks, and actions.
- Ensure that the error messages of the other system are well formatted in vRealize Orchestrator, or back up to vRealize Automation or the calling system. Wrap the exception messages to clearly indicate where the issues happened. For example, "An error occurred while trying to xxxx. YYYY cannot complete. Underlying exception from ZZZZ was: AAAAAAAA"
- Connectivity between calling system and vRealize Orchestrator may not always be stable so consider that in all of your calls. When such timeouts occur, be sure to raise the issue to the end user or inject the error in API response, such that the administrators are made aware of the situation. If the system fails silently you will have a large task of trying to debug the cause of strange symptoms.
- External systems might return results, but could still be invalid. Null or empty result sets are returned, or simply incorrect data for your use case, so validation of the results might be necessary.
- Throw exceptions in Library elements, such as sub-workflow or actions. Catch and react in frontend workflows.
- Avoid passing null when asked to return a list of objects, return an empty array instead as the consumers are likely to loop through the list of objects and an empty array will not trigger exceptions.
- Avoid passing null as a means of communicating an exceptional condition. For example, if asked to get a property of object in an external system, do not pass null to indicate the object no longer exists.
- Translate error messages from library elements to user-friendly messages from frontend workflows, as in a typical Java stack trace.
- Exception binding only allows to transfer a plain string, typed exceptions are not available. Consider introducing artificial error codes in that string in larger workflow solutions, so that you can correlate these specific error codes to conditions you planned for.
- Rollback changes you made, if the workflow is to throw an exception. This must be done across all workflows, including the provisioning, reconfigure, and retire Master Workflows. If this is not done, clearly indicate that in the solution or design documents provided with the workflows.
- Close external connections if necessary, such as with raw JDBC connections to a database. If developing in JavaScript, close the connection with a 'finally' block of a try-catch statement. If it is for a connection that spans multiple workflow elements, ensure all paths close out the connection and even all error scenarios close out the connection which can be done by adding a default error handler.
- Similarly, ensure Locks that are obtained are always unlocked in all conditions. Use the default error handler to ensure the Lock is released in all scenarios.
- If failing during a loop, potentially continue the loop to the next element and log via error or warn level with the failed elements. This should be a clear design decision and documented behavior.
- Do not use exceptions for expected paths in logic.
- Possibly reschedule the workflow with same inputs so that it can "Try again later" when the issue might not be present, such as from communication exceptions.

- Do not perform error handling on errors that the other system handles anyway, just propagate the error message up cleanly. This ensures that your code allows for the external system to behave differently in the future. For example, if a system does not support specifying a CPU value greater than eight, yet in the future the system provides support for a value of 16 then your code should not have had a check for a max of eight as any change to the external system would then also require a change in your code.
- Use "Resume from failed behavior" on a Workflow to have vRealize Orchestrator provide a User Interaction when the workflow fails. For more information, see the VMware vSphere 6.0 Documentation Center. This allows you the opportunity to diagnose the issue better and even allows the user to correct the issue and continue.
- Possibly send notifications of the exceptions, perhaps through email or SNMP.
- Do not bind all exception paths to a "Throw exception" element. This is the default behavior.



- If the desired behavior is to have a default exception handler which perhaps always unlocks a lock and emails a notification, use the new Default Error Handler workflow element:

Input Validation

Provide some initial input parameter restrictions by using the Presentation tab. Whether the input is mandatory, the required ranges for numbers, and so on. These are used for the User Interaction only.

Entries in the Presentation tab guides the user who is interactively running it, however do not rely on the Presentation tab to do your validation. Provide a validation scriptable element for each top-level workflow to cover the inputs and any attributes that are linked to configuration elements, as a workflow run can skip the restrictions on the Presentation tab and have invalid inputs in the following scenarios:

- Another workflow calling into this workflow.
- An external system executing the workflow via vRealize Orchestrator API.
- A user scheduling a Workflow who picks all the correct inputs, but when the workflow runs the inputs are no longer valid. An example would be if you pick an object which gets deleted by the time the workflow runs.
- A user updating a configuration element linked to your attribute.

Logging

Logging that is clear to understand and provide depth of detail to debug issues is critical in a well-integrated Orchestration workflow. There are a multitude of logs generated by vRealize Orchestrator, below is a listing of the most important logs to monitor and the source of their data.

- Server.log: This is the main log of vRealize Orchestrator server. It contains the same information as other logs from the internal application server, in addition to more data. Everything that happens on the vRealize Orchestrator server is seen here. This should be the first log file that you can start debugging with.
- Scripts-log.log: Contains the thread from all the executions of workflows and actions. It lets you isolate these executions from normal vRealize Orchestrator operations.
- Vso.log: This is the client log. It contains connection issues with the server and events on the client side.

The log files above capture information from the vRealize Orchestrator server, workflows, scriptable tasks, and actions. Guidelines for logging within these elements include:

- Use debug-level logging liberally. There is a performance overhead with logging, but most vRealize Orchestrator servers are configured to save only the info level logging so debug level logging is only visible while you work in the client or if the server is set up to retain debug logs.
    - Depending on the client / environment, the logging level might not be changed in a production system so you might have to log some debug-level logs to the info level but be considerate of the performance implications.
    - **Tip**: Placing the server to record debug level logging automatically shows the inputs and outputs of every workflow / action. Similarly, it displays the name of the executing action in the logs.
- When logging object states, be sure to include something that identifies the object and the state.
- Custom decision blocks can be used to log the paths of decisions.
- vRealize Orchestrator displays the action name in the logs so there is no need to log entry into the action.
- Introduce a context-ID when calling asynchronous workflows in order to track the overall workflow.
- Consider creating an action to provide consistent logging.
- Potentially write a reduced set of logs to third-party systems via an action, which can support retries.
- Avoid Server.log() or use it very sparingly, as it writes the logs to the vRealize Orchestrator database forever, which can have storage and performance implications.

Testing

Testing is a critical part of the overall Orchestration lifecycle, it is only in this phase of the process that validation between software and use cases occurs. Results of this phase can feed valuable input into other portions of the cycle, and thus we recommend testing be done early and often.

Sub workflows or actions should be designed to be tested on their own. Make sure they can be executed individually, that is, Properties as inputs can be an issue and may be avoided for your testable workflows.

Unit Tests

vRealize Orchestrator is a system intended to integrate with other products which results in most testing being performed as integrated testing. It can be very valuable to write Unit Tests to test out the sub-workflows or actions free from interactions with other systems. There is no unit testing framework in vRealize Orchestrator yet, so you must write tests on your own.

- Another reason why splitting up workflows into sub workflows and actions is desirable. Create simple workflows which can be executed standalone with a simple set of inputs.
- Unit tests document your code in a manner of speaking, and can be used by others to see expected behaviors.
- Check the results in your unit tests so that they can be used for automated testing.

- Keep unit tests in a separate package which is checked into the source code control system.
- Avoid setting 'test' flags within your workflows which result in a different path being taken. This is not a clear test of the overall workflow.

Sample Use Case

Provide a workflow to migrate all VMs from a smaller datastore to a larger datastore. After completion, rename the source datastore by appending "–DECOM" to the name, denoting to the IT team that it can be decommissioned. A vSphere datastore can be 42 characters only, which brings up scenarios where names might clash. The approach should then truncate the name to fit within 42 characters while having the suffix -DECOM. If the new name still would clash, truncate the name further and append a number to the truncated portion before adding the -DECOM so that it still fits into 42 characters. Repeat the clash algorithm until the name is unique.

With this use case there are many scenarios to test: happy path, name exists without truncation, name exists with truncation, name exists with truncation and after number has been added, name exists with truncation and after number has been added, name exists with truncation and after number has been added multiple times (over 10, so that name had to be truncated multiple characters to fit), and so on. To test all of these scenarios it would require an environment with many configured datastores, and the tests would be very tedious to setup, so it is unlikely all the scenarios would be tested. However, in reality the use case really only cares about the previous name and the new name of a datastore, given a list of all datastores in the system, and it does not need to test the tried-and-true rename function of datastores from within vRealize Orchestrator.

Therefore, create a workflow with inputs of (1) current datastore name and (2) a list of existing datastore names, and as an output just provide the target datastore name. The workflow has now severed its connection with vSphere both inbound and outbound and is relying on the parent workflow to do the setup such as getting the list of datastore names from vSphere, and it is relying on the parent workflow to simply perform the rename given the new name. This allows for unit testing of the interesting functionality of the use case by passing in different combinations. This can be accomplished by having a separate workflow which goes through all the scenarios so that there are repeatable tests. The tests can even check the results and throw exceptions if the results do not match expectations.

Avoiding External Communication

With a well-designed solution, there will likely be a set of building-block actions and workflows which perform the actual communication with a particular external system. If all of these can be converted to actions within a particular module (action folder), then it can be possible to pass in a mock module into your parent workflows which has all the same actions defined but does not perform the communication with the external system. This allows for the testing of the interesting business-logic layer without having to change the parent workflows to place them into a test mode.

Integration Tests

Testing integrations in vRealize Orchestrator workflows and to external systems is essential. Integration testing must include:

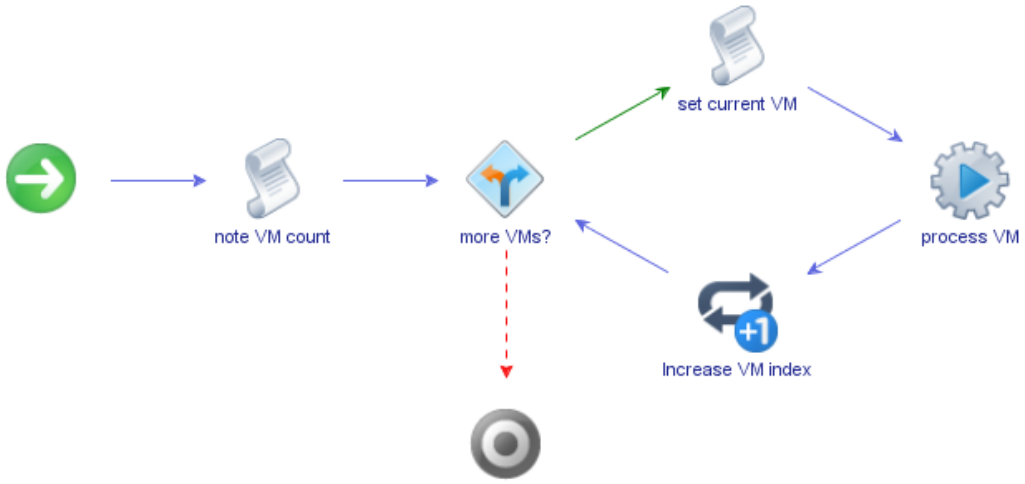
- Single end-to-end workflow mimicking test input to exercise all of the integrations between workflows, actions, and scriptable tasks.
- External system service account validation, as external systems often rotate service account credentials, and any integrations you perform are invalidated unless the service account used by vRealize Orchestrator is also rotated to the new credentials.
- Integrations with email and database systems that often have exceptions that are thrown specific to space or domain requirements. Integration tests must be able to catch the predicted exceptions, and allow for predictable behavior for the Orchestration to recover gracefully and remain functional in an SLA.

vRealize Orchestrator Platform Awareness

Serialization and Resuming Workflows

vRealize Orchestrator serializes the inputs and outputs of each workflow element to the database so that if the server goes down the workflow can be resumed from the serialized state. Normally, the deserialization is skipped in order to minimize performance and memory impact, however portions of your workflows may have issues which are not clear unless deserialization is exercised, therefore:

- Do not try tricky JavaScript that might cache variables and objects between workflow elements in a non-obvious way. That is, stick to inputs and attributes, do not try to pass around JavaScript objects with functions that cannot be serialized or anything similarly creative.
- Ensure you explicitly provide outputs from sub workflows, as they will not update the parent workflow objects. It does not work like other programming languages where you can change in-memory objects.
- If in doubt about sub, nested, or external workflows, call them asynchronously for a quick test. This forces vRealize Orchestrator to serialize the inputs from your workflow and deserialize them in the other workflow.
- Look at any workflow element and consider a scenario where the element may have been cut off mid run and if it can be rerun from the start safely. For example, are there any big implications with re-adding values into an external system - do they overwrite values, do they add duplicates, is there a session object created in the external system at the beginning of the workflow which would need to be re-created, and so on.
- Loop through smaller sets of objects by using the looping mechanism at the workflow level, which will ensure the workflow processes one item at a time:

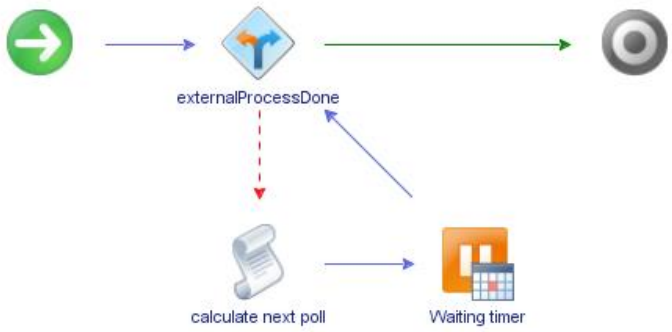


Performance Considerations

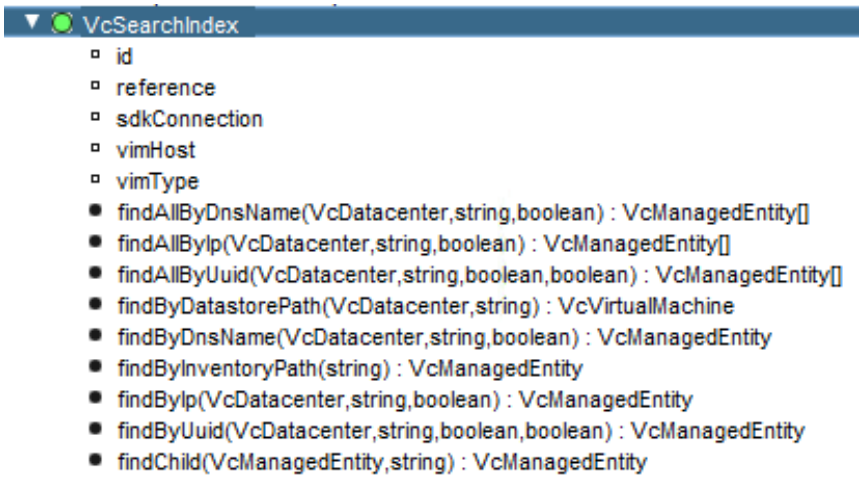
Best practice for time optimization is to first engineer for the longest automation task within the orchestration cycle. Once this automation task is optimized, the overall orchestration cycle sees a subsequent improvement in performance. The longest automation task is the Long Pole in your orchestration process. Do not prematurely optimize for performance and try to make readable, clean code first, then apply performance tweaks. Also, consider if your workflow performance matters for the use case. For example, making your workflow 3 seconds faster is not valuable if it kicks off a task that runs for 15 minutes, or it is scheduled to execute in the middle of the night.

Performance of the combined Automation and Orchestration system is dependent on multiple drivers, including:

- Third-party integration time. Any system integrated with vRealize Orchestrator, for example, IPAM, will have inherent time required to complete its tasks before returning to vRealize Orchestrator with the results. Performance design for vRealize Orchestrator workflows must take this integration time, as well as return states into account.
- Timeout defaults in the vRealize Orchestrator, vRealize Automation, and vSphere platforms.
- Timeout values in database of your implementation.
- End user dependent workflows including approvals.
- Codify the timeout for each external system as a configuration item in vRealize Orchestrator.
- Have a global configurable max timeout to wait for an external system. If the external systems do not support a timeout during the requests, consider making the calls with an asynchronous workflow so that the parent workflow can have a timeout as it waits for the results of the asynchronous workflow.
- Fail workflows when first the system-specific timeout fails, followed by the larger global-configured timeout.
- If external systems require a timeout of more than 3600 seconds, edit the vRealize Orchestrator default timeout in configurations to encompass this value.
- Workflows that require pausing or sleeping:
    - For short sleeps, you only need to do a System.sleep to avoid serializing and deserializing the workflow inputs, attributes, and so on.
    - Sleeping for an extended period of time should be done with a Workflow sleep, by using a Waiting Timer versus a sleep within a scriptable task.



- Finding elements:
    - Do a bulk query where possible and keep the objects cached, rather than looping and querying individual items. This applies for many external systems, such as REST, vSphere, Database.
    - Find by ID or try to find another faster method of finding an object. Some plug-ins accept a query that runs on the server side for the query service, examples of which are shown in the posts for vRealize Automation Cafe, vRealize Automation IaaS, vCloud Director, and vCenter.



- Consider using Properties to do easy lookups. For example, finding a set of unique datastores for a given set of VMs using Properties:

```
var datastoreIdToDatastore = new Properties();
for each (var vm in vms) {
    for each (var datastore in vm.datastore) {
        datastoreIdToDatastore.put(datastore.id, datastore);
    }
}
return getUniqueDatastores(datastoreIdToDatastore);
function getUniqueDatastores(datastoreIdToDatastore) {
    var datastores = new Array();
    for each (var datastoreId in datastoreIdToDatastore.keys) {
        datastores.push(datastoreIdToDatastore.get(datastoreId));
    }
    return datastores;
}
```

- Looping through all objects retrieved is performance prohibitive to resolve the issue.
    - Break from a loop, if you find the item you are searching for, such that you do not incur a performance penalty for the remainder of the loop.
    - Look for a different approach, such as the find-by-ID methods discussed above.
- Have an understanding of how the plug-in works, especially as it relates to caching of objects. It can be a bit hard to find out, so ask in the communities or look for any posts related to the plugin's caching.
    - vSphere has a cache in vRealize Orchestrator.
    - vRealize Automation does not have a cache of the Entity objects so it does a lookup every time. Passing around the objects can cause a lot of performance issues.

- Looping through big arrays.
    - Loop through big arrays in Scriptable Tasks, if you are not doing any writes, such as finding an element or a set of elements.
    - Consider restructuring some data structures using a Properties to do lookups as shown in the example above.
    - O(n) vs O(n^2) - looping through a loop multiple times is not as big of a deal as nesting loops.

- Since vRealize Orchestrator 5.5.1, there is a cache that you can use to put something into cache or retrieve from cache, this improves performance at an incremental scale.
    - Methods that can be used to access cache, putInCache / getFromCache.
    - Good to use for Presentation.
    - This cache is not shared among vRealize Orchestrator systems, events of failover will not replicate the cache to another vRealize Orchestrator server, careful use of the cache is advised. Best practices for cache failures are to do the lookup as advised above.
    - Should take precautions when something is not in the cache, do the lookup. This can happen, if vRealize Orchestrator needs to resume your workflow. The cache will not be there anymore.

vCenter Plugin Querying

Use the vCSearchIndex provided by the plugin to query where possible, as mentioned above. If those are not sufficient then often the vCenter plugin's 'getAll...(...)' methods are used. Each of these methods accepts an optional array of Strings specifying additional properties to retrieve when querying for the objects, and an optional query by using an xpath definition. For example, getAllVirtualMachines([additionalPropertyFilters], query).

Using the xpath querying is the preferred approach as it applies the filtering on the vCenter servers before constructing a set of objects and returning them. For example:

```
var vmName = "vcova";
// XPath expression for VM name exactly matching the given string
// var xpath = "xpath:name='" + vmName + "'";
// XPath expression for VM name starting with the given string
// var xpath = "xpath://name[starts-with(.,'"  + vmName +"')]";
// XPath expression for VM name containing the given string as a substring
var xpath = "xpath:name[contains(.,'"  + vmName +"')]";

var vms = VcPlugin.getAllVirtualMachines(null, xpath);
if (vms == null) {
    System.log("No VMs found");
} else {
    for each (var vm in vms) {
        System.log("Found VM: " + vm.name);
    }
}
```

For performance reasons, the returned objects are not fully populated with all of the properties but a pre-defined set of properties. The initial set can be determined by looking at the plugin inventory – all properties displayed in the inventory are in the initial properties set. If additional properties are needed, specify them during the getAll...(...) call so that the request from the vCenters can get a combined set of properties upfront. If this is not done and extra properties are referenced by the workflow or action then a remote call is made each time to vCenter to create a vCenter filter for the tuple <vm, property name>, which wastes resources both for remote execution and for filter management on vCenter side.

As these are customized objects that are returned by the call, they do not match with what is exactly in the inventory, storing them as workflow attributes, inputs or outputs would serialize them throughout the workflow. If the returned set is large it can have an effect on the system performance. One approach might be to store them in a shared resource like a vRealize Orchestrator configuration element or resource element and then just reference the location where they are saved to minimize serialization and deserialization. Be aware that the objects may not exist if you process them at a later stage.

Calling Actions

Actions are usually called with the following code:

```
var result = Server.getModule("action.package").action(param1, param2);
```

This is perfectly fine if this action is called once. But, if you call this action in a loop there will be a performance issue, because for every call to Server.getModule(...) requires vRealize Orchestrator to make a call to database to retrieve content of that module. In these cases, a better approach would be to retrieve the module once:

```
var module = Server.getModule("action.package");
var result = module.action(param1,param2);
```

Note: As noted above, certain non-standard ways of calling actions may mean the action might not be picked up, if the calling workflow or action is added to the package, so ensure you capture these actions manually.

XML

Two XML parsers are available in vRealize Orchestrator: DOM (legacy) and E4X (dot notation). However, it is strongly recommended to use only E4X.

Note: This parser is now deprecated by Mozilla foundation, but as the Rhino engine version used by vRealize Orchestrator is not the latest version, it is safe to use it.

E4X introduces a new type, "XML", which holds an XML element.
It is possible to create literal XML values by writing XML directly:

```
var xmlObj = <root>
   <child1 attr1="value1" attr2="value2">
      <grandchild1>my value 1</grandchild1>
      <grandchild1>my value 2</grandchild1>
   </child1>
</root>;
```

To parse a string representing an XML object, the constructor XML can be used.

```
var xmlObj = new XML(xmlString)
```

As XML type is not automatically serialized by vRealize Orchestrator between workflows or actions, it must be converted to string before being used as output of a workflow.

```
var xmlString = xmlObj.toXMLString()
```

JavaScript variable can be easily added in the XML object during the construction using {} notation.

```
var myvalue = "my value 1";
var xmlObj = <root>
   <child1 attr1="value1" attr2="value2">
      <grandchild1>{myvalue}</grandchild1>
      <grandchild1>my value 2</grandchild1>
   </child1>
</root>;
```

XML Header

E4X removes the xml header when a string representing an XML object is parsed. When the XML object is converted back to a string the xml header disappears.
Some APIs refuse an xml object if the header <?xml version="1.0" encoding="UTF-8" standalone="yes" ?> is not present, in that case, it must be added manually.

XML Namespaces

Declaring namespace is really important, because most of the XML files contain a default namespace, without this namespace declared in the code, E4X parser does not return any results. If there is only one namespace declared on the root node, the following code can be used before any XML operations.

```
default xml namespace = xmlObj.namespace();
```

Otherwise a namespace can also be declared manually:

```
ovf = new Namespace("http://schemas.dmtf.org/ovf/envelope/1");
vcloud = new Namespace("http://www.vmware.com/vcloud/v1.5");
xsi = new Namespace("http://www.w3.org/2001/XMLSchema-instance");
```

Navigating XML

Once the XML object is created, it is possible to access its name and namespace, children, attributes, and text content using the familiar Javascript dot operator.
Note that the type of object returned is not a string/number, to get a value it is safer to add "toString()" and eventually parse it for number, when it is used as an output attribute.

Getting node value	`xmlObj.child1.grandchild1.toString()` `return "my value 1" (without quotes)`
Getting attribute value	`xmlObj.child1.@attr1.toString()` `return "value1" (without quotes)`
Getting node with explicit namespace	`xmlObj.namespace::child1`
Getting element part of a list	`xmlObj.child1.grandchild1[1].toString()` `return "my value 2" (without quotes)`

More details in the following tutorial: E4X Quick Start Guide.

JSON & JavaScript objects

JavaScript objects can be used between workflow elements by passing around Any object. You can use the JavaScript Object Literal notation to declare objects and new properties in the object:

- ex1: `var obj = { key1: value}`
- ex2: `obj.newProperty = value;`

vRealize Orchestrator can serialize and deserialize nested and complex objects; however since vRealize Orchestrator is running on Java, using Mozilla's JavaScript engine, the serialized versions of the objects get serialized into Java objects and do not deserialize back into JavaScript objects. The implication to this is that nested or complex JavaScript objects cannot be transformed into JSON, so if the intent is to have a complex JavaScript object passed along to an external system as JSON, you will have to serialize the JavaScript object into a String and deserialize it between workflow elements, by using JSON.stringify and JSON.parse respectively.

Tip: Use Properties objects for simple Key-value pairs instead of JavaScript objects because the contents can be viewed in the workflow execution's Variables table by clicking the small  icon

Create or use JavaScript objects and then use JSON.stringify to make it into a JSON. Do not construct JSON manually through the use of string concatenation, which can get exceptionally difficult to read.

INCORRECT JSON CONSTRUCTION ELSE	`var jsonTxt = "{ key1: " + value1 + ", key2: " + value2`
	`+ ", key3obj: { subkey1: " + subValue1 + " }}";`
	`var jsonObj = eval(jsonTxt);`

PROPER – USE JAVASCRIPT OBJECT	`var jsonObj = {`
	`  key1: value1,`
	`  key2: value2`
	`};`
	`// shown for example - could have been part of initial jsonObj:`
	`jsonObj.key3obj= {`
	`  subkey1: subValue1`
	`};`
	`var jsonTxt = JSON.stringify(jsonObj);`

Recommended Reading

There are a lot of programming, software design, and JavaScript books out there. The following list of books is highly recommended for workflow developers.

- VMware vRealize Orchestrator Cookbook, Daniel Langenhan, Packt Publishing - ebooks
- Eloquent Javascript, Marijn Haverbeke, No Starch Press, 2nd edition (December 14, 2014)
- The Pragmatic Programmer, Andrew Hunt, David Thomas, Addison-Wesley Professional, 1st edition (October 30, 1999)
- Code Complete, Steve McConnell, Microsoft Press, 2nd edition (June 19, 2004)
- Clean Code, Robert C. Martin, Prentice Hall, 1st edition (August 11, 2008)
- The Secrets of Consulting, Gerald M. Weinberg, Dorset House Publishing, 1st edition (January 1986)

VMware, Inc. 3401 Hillview Avenue Palo Alto CA 94304 USA Tel 877-486-9273 Fax 650-427-5001 www.vmware.com