vRealize Automation 8.x Extensibility Migration Guide

12 August 2021
vRealize Automation 8.4
vRealize Automation 8.5

vmware®

You can find the most up-to-date technical documentation on the VMware website at:

https://docs.vmware.com/

VMware, Inc.
3401 Hillview Ave.
Palo Alto, CA 94304
www.vmware.com

Copyright © 2021 VMware, Inc. All rights reserved. Copyright and trademark information.

Contents

1 vRealize Automation 8.x Extensibility Migration Guide 5

2 Sample Package and Dynamic Types Plug-in Generator 6

3 Accessing vRealize Automation Objects and Properties 7

- Persist and Manage vRealize Automation Orchestrated Hosts with Their Credentials 9
- Pass Credentials from a vRealize Automation User to the vRealize Automation Plug-in for vRealize Orchestrator 10
- vRealize Automation 8.x Finder Objects 10
- vRealize Automation 8.x Scripting Objects 13
- vRealize Automation Scripting Objects and REST Queries 14
- Actions and Workflows Supporting Common Operations 15

4 Customizing Machine Provisioning 17

- Customize Machine Properties or Deployments with Extensibility Topics 17
- Customize Machine Properties or Deployments using the vRealize Automation API 19

5 Day 2 Operations on IaaS Entities 21

- Custom Form API Call Examples 24

6 Using Dynamic Types with Custom Resources in vRealize Automation Cloud Assembly 30

- Creating the Dynamic Types Configuration 31
- Dynamic Types Object and Custom Resource Requirements 32
- Create the Dynamic Types Custom Resource 32

7 Lifecycle Extensibility 35

- Migrating Subscriptions from vRealize Automation 7.x to vRealize Automation 8.x 37
- Creating a Subscription 38
- Create a Wrapper Workflow 39

8 Onboarding a Customer Organization 43

- Onboarding a Project 44
  - Sychronizing the Workspace ONE Access Directory 45
  - Creating a vRealize Automation Project 46
  - Associating a Tag with the Project 47
  - Add Cloud Zones to the Project 47
  - Assign Cloud Assembly and Service Broker User Roles 48

Assign Catalog Items to a Project 48
vRealize Orchestrator Implementation for Project Onboarding 49
Adding Resource Provisioning to a Project 51

9 Requesting Catalog Items 52
API Tag Filtering Examples 53
vRealize Orchestrator Action Example 53
Basic Sample Cloud Template 54
Associating an External Value with the getTagByKey Action 55
Example Service Broker Catalog Request 56
Requesting Catalog Items Programatically 56

10 Tags and Custom Properties 59

11 Using vRealize Automation XaaS Services 63
Differences between vRealize Orchestrator Forms and Service Broker Forms 63
Workflow Sample 67
Using Custom Resources 68
Resource Mappings 69
Custom Cloud Template Component 70

vRealize Automation 8.x Extensibility Migration Guide

1

The *vRealize Automation Extensibility Migration* guide provides information about functionality changes between vRealize Automation 7.x extensibility and vRealize Automation 8.x extensibility.

The *vRealize Automation 8.x Extensibility Migration Guide* includes use cases that demonstrate the extensibility functionality in vRealize Automation 8.x.

For information on migrating vRealize Automation 7.x to 8.x, see the *vRealize Automation 8 Transition Guide*.

Sample Package and Dynamic Types Plug-in Generator

2

To develop the use cases documented in this guide, you must download the required sample package. You can also download an updated Dynamic Types plug-in generator that can be used for third-party API integrations.

Sample Package

The use cases in this guide reference resources included in the sample package hosted on VMware {code}. To download the package, see vRealize Automation Extensibility Migration Guide Samples.

Dynamic Types plug-in generator version 3

When developing integrations with third-party APIs for your vRealize Automation 8.x environment, the Dynamic Types plug-in generator version 3 can be used to replace version 2 of the generator that was designed for vRealize Orchestrator 7.x. To download the version 3 of the plug-in generator, see Dynamic Types plug-in generator version 3.

Accessing vRealize Automation Objects and Properties

3

Most of the scenarios in this guide require access to the objects of the vRealize Automation services. This process is required so you can access and configure object properties or run operations on the objects.

You can access the vRealize Automation 8.x services through the REST API by using a REST Client, or through the vRealize Automation plug-in for vRealize Orchestrator

The vRealize Automation REST APIs are documented in:

- The built-in Swagger/OpenAPI documentation hosted on your vRealize Automation server. This documentation can be found at https://*your_vRA_FQDN*/automation-ui/api-docs/
- The vRealize Automation API Programming Guide

The vRealize Automation plug-in for vRealize Orchestrator 8.x is documented in *Using VMware vRealize Orchestrator 8.x Plug-ins* and the API Explorer built into the vRealize Orchestrator Client.

Different services can have different API behavior such as:

- The Infrastructure as a Service (IaaS) service has a different query service syntax and paging syntax from the other services.
- The IaaS service returns a payload of the object created on POST. Other services returns the object ID in the location header.

You can enable the developer view to capture calls to the vRealize Automation services to receive more information about:

- Operations (GET, PUT, POST, PATCH, DELETE)
- Base URL (/service/api/resource)
- Parameters (paging, sorting, queries)
- Request payload (The JSON information passed to create / update objects)
- Response payload (The JSON information returned to describe the object and their properties)

However, there are some considerations and differences regarding the vRealize Automation API services:

- The user interface uses some proprietary service endpoints. These are not documented or supported and might be changed or become inaccessible outside the user interface in following product releases without notice.
- The enabled operations are different. For example, it is not possible to update machine custom properties in vRealize Automation 8.1. However, the functionality is available in vRealize Automation 8.2. In the user interface, it was always available in vRealize Automation 8.1.
- The response payload might be different as some properties might be missing.

The vRealize Automation 7.x user interface uses the public API fully and has more options to access service APIs, including software development kits (SDKs) for different languages. In vRealize Automation 7.x, all extensibility functions include Event Broker subscriptions, Anything as a Service (XaaS) cloud templates, custom resources, and resource actions. XaaS cloud template components, such as custom resources, leverage vRealize Orchestrator workflows. Custom forms leverage vRealize Orchestrator actions.

For many use cases in vRealize Automation 7.x, this vRealize Orchestrator based extensibility requires access to vRealize Automation to get further information from the payload passed from vRealize Automation to vRealize Orchestrator. The services also list vRealize Automation objects, so these objects can be used in extensibility. The most common approach to accessing vRealize Automation 7.x objects is from the vRealize Automation plug-in for vRealize Automation. This plug-in can be accessed either from the built-in REST client or through plug-in objects.

The vRealize Automation 7.x plug-in provides:

1 A way to persist and manage vRealize Automation orchestrated hosts with their credentials.

2 The ability to pass host and credentials from a vRealize Automation user to vRealize Automation plug-in to make API queries as this user.

3 An inventory of 92 objects allowing users to select objects by their name or properties in drop-down menus or tree-view.

4 Over 800 JavaScript scripting objects and their documentation (API explorer).

5 Hundreds of library actions and workflows supporting common operations.

The updated vRealize Orchestrator plug-in for vRealize Automation 8.4 available on market place and out of the box in vRealize Automation 8.5 provides:

1 A way to persist and manage vRealize Automation orchestrated hosts with their credentials.

2 The ability to pass host and credentials from a vRealize Automation user to vRealize Automation plug-in to make API queries as this user.

3 An inventory of 11 objects allowing users to select objects by their name or properties in drop-down menus or tree-view. These objects automatically manage vRealize Automation API paging capabilities.

4 Over 40 JavaScript scripting objects and their documentation available in the API Explorer.

5 Close to a hundred library actions and workflows supporting common operations.

This section of the guide discusses the implementation of the above use cases in vRealize Automation 8.x.

Many of the workflows triggered by vRealize Automation 7.x leverage the vRealize Automation plug-in to access vRealize Automation services. The workflow elements making use of these plug-ins as well as those using the vRealize Automation 7.x REST API must be rewritten.

To identify the workflow elements that require a rewrite after migrating to vRealize Automation 8.x, please use the vRealize Automation Migration Assistant. The migration assistant is available from https://*your_vRA_FQDN*/migration-ui. For more information on migrating to vRealize Automation 8.x, see the *vRealize Automation 8 Transition Guide*.

Another way to identify workflow elements that require a rewrite is to import the extensibility workflows in vRealize Automtion 8.x and use the workflow validation tool which identifies errors in each workflow item. Additionally, when importing content from a vRealize Orchestrator 7.6 environment to a vRealize Orchestrator 8.x environment, you might want to avoid importing the vRealize Automation 7.x plug-in library workflows. These workflows are in a read-only state and after importing them, you will be unable to delete them from the vRealize Orchestrator 8.x server.

This chapter includes the following topics:

- Persist and Manage vRealize Automation Orchestrated Hosts with Their Credentials
- Pass Credentials from a vRealize Automation User to the vRealize Automation Plug-in for vRealize Orchestrator
- vRealize Automation 8.x Finder Objects
- vRealize Automation 8.x Scripting Objects
- vRealize Automation Scripting Objects and REST Queries
- Actions and Workflows Supporting Common Operations

Persist and Manage vRealize Automation Orchestrated Hosts with Their Credentials

Learn how to persist and manage vRealize Automation orchestrated hosts.

vRealize Automation hosts can be persisted and managed with the vRealize Automation 8.x plug-in for vRealize Orchestrator. See *Add a vRealize Automation Host* in *Using VMware vRealize Orchestrator plug-ins*.

The examples provided in the sample section of this guide include a default vRealize Automation host created by using a shared session. The host is stored as `VRA:Host` and is saved in the `"vRA plug-in"` configuration element.

Pass Credentials from a vRealize Automation User to the vRealize Automation Plug-in for vRealize Orchestrator

This use case presents alternatives that you can use to pass user credentials without using the vRealize Automation plug-in "per user" session capability.

In some use cases, it is required that the roles and permissions for the workflow accessing back to vRealize Automation are the same as the user who initiated the workflow from vRealize Automation. Some of the use cases that require this are:

- For auditing purposes - when it is necessary to track which user made a change, even if this is through a workflow the user triggered in vRealize Automation.
- For presenting information that the user can access, such as the content of a drop-down menu run by XaaS operations or a query run within a workflow.
- For taking actions, modifying properties with the role and permissions of the user. For all operations, the user can trigger that through extensibility.

As of vRealize Automation 8.4, there is no solution to pass the vRealize Automation authentication from vRealize Automation to vRealize Orchestrator so it can be used to authenticate back in vRealize Automation. The vRealize Automation plug-in supports per user sessions, but the plug-in uses a service user role with limited rights.

The workaround for this limitation is to query the end user to reenter their credentials when they run the workflow. However, doing so exposes their credentials to the vRealize Orchestrator developer.

Another solution that prevents users from accessing unauthorized data is to use a service account and create action-based filters by project based on user permissions.

The sample workflows provided in the sample package use the default host provided when running the `Set vRA Host` workflow.

vRealize Automation 8.x Finder Objects

The vRealize Automation 7.x inventory includes 92 finder objects that can be used to select other inventory objects by name or properties.

The finder objects can be applied through either drop-down menus or a tree view.

You can implement drop-down menus or tree views for the plug-in objects that are documented in API Explorer under `vRA / Types` by adding them as workflow inputs. With the latest version of the plug-in for added in product version 8.4, these objects types are :

- `VRA:CloudAccount`
- `VRA:CloudAccountNsxT`
- `VRA:CloudAccountNsxV`
- `VRA:CloudAccountVsphere`

- `VRA:DataCollector`
- `VRA:Host`
- `VRA:Infrastructure`
- `VRA:Project`
- `VRA:Region`
- `VRA:Tag` (supports only drop-down menus)
- `VRA:Zone`

For the other object types you can implement a drop-down menu either with string-based inputs displaying object names bound to an action, or by using one of the following methods:

- Authenticating in vRealize Automation by passing the `VRA:Host` including the specific credentials. The object is included with the sample package.
- Creating a REST query to list the objects by applying filters if necessary.
- Return the object names from the JSON payload.

The string-based input drop-down menu can be populated with actions that return properties that include the ID of the object as a key and the name of the object as a value. By using these properties, you can access the object ID without needing to create actions that query the object by name. An example of such an action is the `getDeploymentsIdsAndNames` action included in the sample package.

To perform efficient queries to find objects, it is necessary to use the query service. Retrieving all objects and iterating through them in a loop is not a best practice, particularly for objects that can have hundreds or thousands of iterations.

Use filtering as much as possible to avoid using CPU, IO, memory, input/output (I/O), or network resources on both the vRealize Automation and the vRealize Orchestrator deployment.

The following example includes a query used to find an IaaS (Infrastructure as a Service) machine by name. The sample code snippet is taken from the sample action `getMachineByNameQS`.

```
var url = "/iaas/api/machines";
// Query service parameter
var nameFilter = "name eq '" + machineName + "'";
var parameters = "$filter=" + encodeURIComponent(nameFilter).replace("'", "%27");


var machines =
System.getModule("com.vmware.vra.extensibility.plugin.rest").getObjects(vraHost,url,parameters
);
if (machines.length == 1) return machines[0];
if (machines.length == 0) return null;

// More Machines returned than expected !
System.warn("getProjectByNameQS returned " +  projects.length + " projects");
return null;
```

You must encode any variable that can contain spaces or other special characters that are not accepted in the URL or from the server. For example, an apostrophe **(')** must be replaced with **%27**.

Paging must also be handled. The default number of object returned in a single query is limited. To get all objects, it is possible to:

- Change the default number of objects per page.

  Note There can be a maximum limit.

- Make different queries for different page numbers until all objects are received.

The samples actions `getIaaSObjects` and `getDeploymentObjects` provide samples on how to use the paging parameters with the IaaS and deployment services. Depending on the service in use, this is done either with the skip parameter or the page parameter.

The following sample includes the `getIaaSObjects` sample code:

```
if (vraHost == null || url == null) return null;

var iaasObject =
System.getModule("com.vmware.vra.extensibility.plugin.rest").getObjectFromUrl(vraHost,url,
parameters);
var content = iaasObject.content;

var skip = 0;
var elementsLeft = iaasObject.totalElements - iaasObject.numberOfElements;
var allContent = content;
var numberOfElements = iaasObject.numberOfElements

while (elementsLeft >0) {
  var skip = skip + numberOfElements;
  if (parameters == null) parameters = "$skip=" + skip;
  else parameters = parameters + "&$skip=" + skip;
  iaasObject =
System.getModule("com.vmware.vra.extensibility.plugin.rest").getObjectFromUrl(vraHost,url,
parameters);
  content = iaasObject.content;
  elementsLeft = elementsLeft - iaasObject.numberOfElements;
  allContent = allContent.concat(content);
}

 return allContent;
```

The following sample includes the `getDeploymentObjects` sample code:

```
if (vraHost == null || url == null) return null;

var object =
System.getModule("com.vmware.vra.extensibility.plugin.rest").getObjectFromUrl(vraHost, url,
parameters);
var content = object.content;
```

```
var page = 1;
var allContent = content;

while (object.last == false) {
  if (parameters == null || parameters == "") newParameters = "page=" + page;
  else newParameters = parameters + "&page=" + page;
  object =
System.getModule("com.vmware.vra.extensibility.plugin.rest").getObjectFromUrl(vraHost, url,
newParameters);
  content = object.content;
  allContent = allContent.concat(content);
  page++;
}

return allContent;
```

To avoid searching for which service is using which query service syntax for paging, the `getObjects()` action checks which query service format to use based on the properties of the JSON file and returns all objects.

As an example of providing an alternative to having inventory objects, the sample package includes the `Drop down` folder. The folder contains workflow examples with forms that use actions to populate the drop-down menus, including deployments, deployment resources, and deployment resource tags.

Another alternative to plug-in inventory objects is to create vRealize Orchestrator dynamic types for the required vRealize Automation objects. In this way, you can use an object as input supporting different properties or a tree view.

In some use cases, a single tree view is more convenient than multiple drop-down menus because you can filter for the object you want to select based on its parents.

vRealize Automation 8.x Scripting Objects

The vRealize Orchestrator plug-in for vRealize Automation includes several scripting objects with different functionalities.

The latest release of the vRealize Orchestrator plug-in for vRealize Automation include some scripting objects. The list of the managed object types is located in the API explorer under `vRA / Objects`. These object types can be used to:

- Define objects specification to create them by using `"VraProjectSpecification"`.
- Get objects details by using `"VraProject"`.
- Run methods on these objects by using `VraProject putCustomPropertiesItem`.

Using these object types simplifies workflow development. Objects are documented in API explorer and provide intelliSense capabilities displaying object properties and methods. When these are also used as inventory objects, troubleshooting is also simplified as their properties can be inspected in a workflow run.

vRealize Automation Scripting Objects and REST Queries

In vRealize Automation 8.x , you can use REST queries to substitute the scripting objects included in the vRealize Automation 7.x plug-in, which can be used to construct, access, and document all program-based objects.

The equivalent of these objects at the REST level is documented in the `Models` section in Swagger. The Swagger models include JSON examples for object properties that can be included in a vRealize Orchestrator action. The Swagger documentation is essential for understanding the properties of the objects returned by REST queries and constructing objects to pass as the body of PUT, POST, PATCH requests.

The following example, `createZone`, is used to create a zone:

```
if (vraHost == null || regionId == null || name == null) return null;

var customPropertiesObject =
System.getModule("com.vmware.vra.extensibility.plugin.rest.iaas").propertiesToCustomPrope
rtiesObject(customProperties);
var tagsObject =
System.getModule("com.vmware.vra.extensibility.plugin.rest.iaas").propertiesToTagsObject(ta
gs);
var tagsToMatchObject =
System.getModule("com.vmware.vra.extensibility.plugin.rest.iaas").propertiesToTagsObject(ta
gsToMatch);

var url = "/iaas/api/zones"
var zone =
{
  "customProperties": customPropertiesObject,
  "folder": folder,
  "regionId": regionId,
  "tagsToMatch": tagsToMatchObject,
  "name": name,
  "description": description,
  "placementPolicy": placementPolicy,
  "tags": tagsObject
}

var content = JSON.stringify(zone);
var operation = "POST";

try {

    var contentAsString =
System.getModule("com.vmware.vra.extensibility.plugin.rest").invokeRestOperation(vraHost,
operation, url, content);
    var object = JSON.parse(contentAsString);
  return object.id;
    } catch (e) {
         throw "POST " + url + "Failed" +
    \n Error : " + e;
}
```

The vRealize Automation plug-in included in 7.x also includes "singleton" objects that provide a global access point to properties (Enumerations : constants) and methods. The methods provide special functionalities. For example, methods to find objects by their properties. The latest version of the vRealize Orchestrator plug-in for vRealize Automation includes a special singleton object called `VraEntitiesFinder` including methods to get plug-in objects for a specific vRealize Automation host by type or by ID. These methods also support providing a string-based filter similar to the filters used by the vRealize Automation UI and documented in vRA API documentations.

For cases where `vRAEntitiesFinder` does not support the search you are looking for, it is possible to provide equivalent functionality through actions using REST queries The following example includes code from the sample action `getNetworksByTagsQS` that can be used to find networks.

```
if (vraHost == null) return null;

var tagsFilters = new Array();
for each (var tag in tags) {
  tagsFilters.push(getTagFilter(tag));
}

// Query service parameter
var tagsFilter = tagsFilters.join(" and ");
if (tags.length == 0) var parameters = "expand";
else var parameters = "expand&$filter=" + encodeURIComponent(tagsFilter).replace("'",
"%27");

var url = "/iaas/api/fabric-networks";
return
System.getModule("com.vmware.vra.extensibility.plugin.rest").getObjectsProperty(vraHost,
url, parameters, "name");

function getTagFilter(tag) {
  tag = tag.replace(":","*");
  return "(expandedTags.item.tag eq '*" + tag + "*'))"
}
```

Actions and Workflows Supporting Common Operations

Actions and workflows can be written by using the vRealize Orchestrator plug-in for vRealize Automation scripting objects and/or the REST client of the vRealize Automation plug-in.

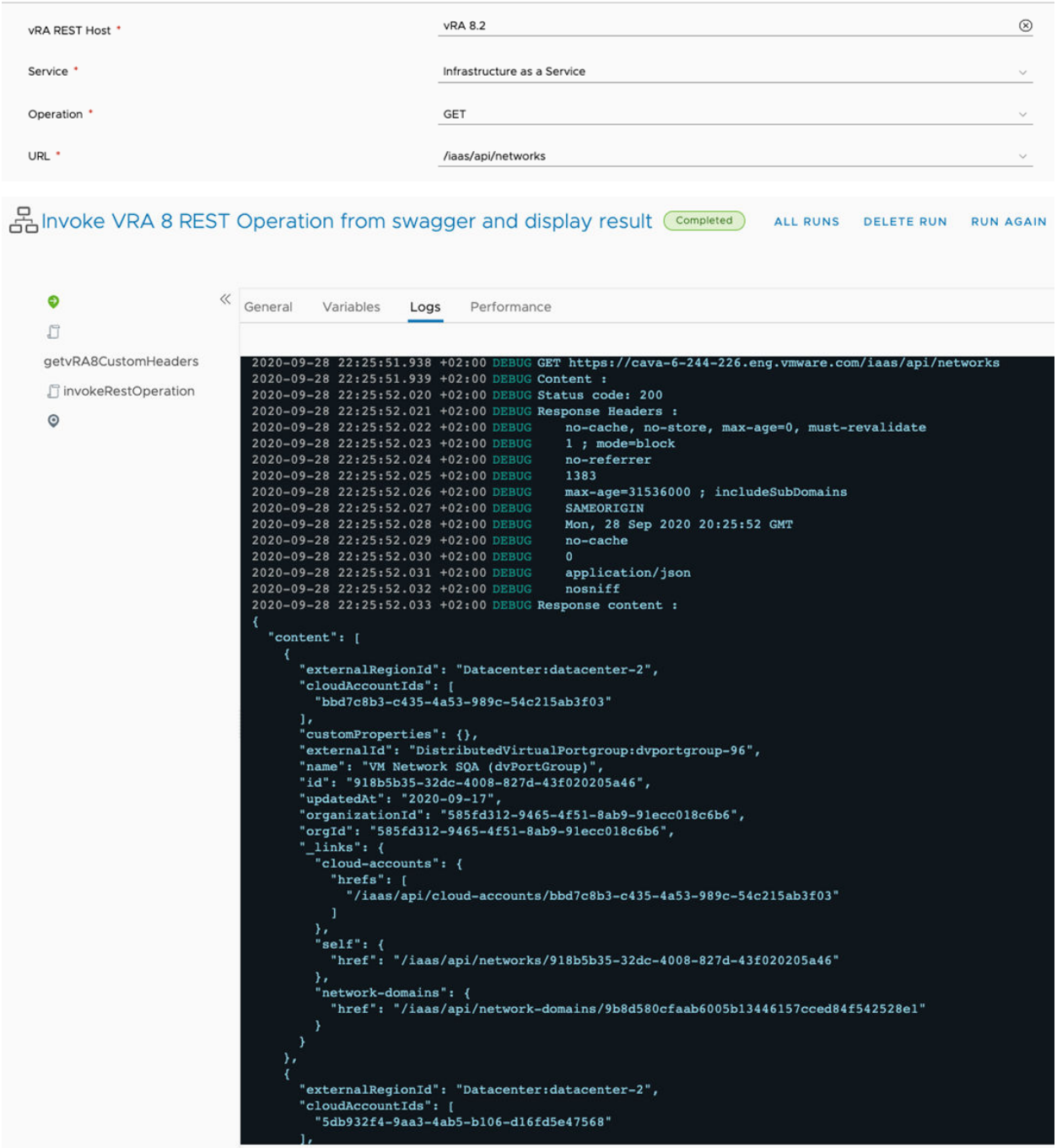
When writing actions and workflows by using the REST API, you must follow these guidelines:

- `Create action` and `Create workflows` must return the object ID received in the payload or in the "location" response header after invoking a POST operation.
- `Delete` and `Update` actions and workflows must have an ID input to pass to the REST query.
- Workflows run by the end user must have an input form that retrieves object names and IDs that return an array of properties.

To test REST API calls, you can use two sample workflows.

- The `Invoke VRA 8 REST Operation from URL` sample workflow allows you to enter free form URLs.
- The `Invoke VRA 8 REST Operation from swagger and display result` sample workflow provides a drop-down menu of services, operations, and URLs based on the vRealize Automation server Swagger.

Invoke VRA 8 REST Operation from swagger and display result



Customizing Machine Provisioning

4

In vRealize Automation 8.x, you can customize machine properties or deployments in two ways. You can use event topics that are already available in Cloud Assembly to modify custom properties during provisioning, or you can use the vRealize Automation API to trigger Day 2 operations on deployments that are already completed.

This chapter includes the following topics:

- Customize Machine Properties or Deployments with Extensibility Topics
- Customize Machine Properties or Deployments using the vRealize Automation API

Customize Machine Properties or Deployments with Extensibility Topics

You can update machine properties or deployments by using the available extensibility topics during the deployment life cycle.

To update the deployment payload, you can create new subscriptions using available extensibility topics such as `Provisioning Request` and `Disk Allocation` that call vRealize Orchestrator workflows or extensibility actions.

Prerequisites

Verify that you have access to the extensibility code samples package.

Procedure

1 To customize machine CPU or memory properties, create a new extensibility subscription.

   a Enter a subscription name. For example, **`Customize CPU/Memory`**.

   b In **Event Topic**, select **Provisioning request**.

   c Next to **Action/workflow**, set an output of type String called `flavor`.

   Note You cannot change the machine CPU and memory properties directly if you do not set a new flavor mapping. The output property must be called `flavor`, and the value must be an existing flavor mapping profile.

   The following code snippet is taken from a sample extensibility action.

```
def handler(context, inputs):
    outputs = {
      "flavor": "large"
    }

    return outputs
```

2 To customize disk allocation, create another extensibility subscription.

   a Enter a subscription name. For example, you can name it **`Disk size`**.

   b In **Event Topic**, select **Disk allocation**.

   c Next to **Action/workflow**, set an output of type Array called `diskSizesInGb`.

   The following code snippet is taken from a sample vRealize Orchestrator workflow.

```
// Customize the size of the first VM disk
var vm_disks = inputProperties.get("diskSizesInGb");
if (isParameterReadOnly("diskSizesInGb") == false) {
    vm_disks[0]=30;
}
diskSizesInGb = vm_disks;
```

3 Request a new virtual machine from the vRealize Automation Catalog Item.

What to do next

Once the deployment is ready, navigate to the virtual machine settings. Verify that the CPU, memory, or disk size are set to the values configured in the extensibility action or vRealize Orchestrator workflow that are used in the subscriptions you created.

Customize Machine Properties or Deployments using the vRealize Automation API

To customize machine properties on already completed deployments in vRealize Automation 8.x, you can use third party tools or vRealize Orchestrator workflows to trigger day 2 operations with API calls.

The following examples use Swagger. You can access the VMware Service Broker API at https://*your_VRA_FQDN*/deployment/api/deployments/{depId}/resources/{resourceId}/requests

Procedure

1 Update the CPU/memory values for a machine resource.

Submit a resource action request.

```
POST /deployment/api/deployments/{depId}/resources/{resourceId}/requests
```

The following code snippet is a sample body:

```
{
    "actionId": "Cloud.vSphere.Machine.Resize",
    "targetId": "e9d88d23-2edb-4dcb-812b-b3593368b164",
    "inputs": {"cpuCount": 4, "totalMemoryMB": 4096}
}
```

Note The `actionId` depends on the Machine object type. For vSphere machines, the object is `Cloud.vSphere.Machine`. The `targetId` is the machine resource object ID. You can access both from the machine resource object custom properties in the vRealize Automation Client.

2 Update the disk size value for a Disk resource.

Submit a resource action request.

```
POST /deployment/api/deployments/{depId}/resources/{resourceId}/requests
```

The following code snippet is a sample body:

```
{
    "actionId": "Cloud.vSphere.Disk.Disk.Resize",
    "targetId": "710f6d3b-4fdc-4883-8acf-08129c2ad07a",
    "inputs": {"capacityGb":30}
}
```

Note The `actionId` depends on the Disk resource object type. For a vSphere disk the object is `Cloud.vSphere.Disk`. The `targetId` is the Disk resource object ID. You can access these from the disk resource object custom properties in the vRealize Automation Client.

3 Update the Deployment Lease.

Submit a resource action request.

```
POST /deployment/api/deployments/{depId}/resources/{resourceId}/requests
```

The following code snippet is a sample body:

```
{
    "actionId": "Deployment.ChangeLease",
    "targetId": "2da7675d-a791-4a4a-bc4f-5817b5c5e9d2",
    "inputs": {"Lease Expiration Date":"2020-09-20T13:07:00.000Z"}
}
```

Note The `targetId` is the deployment ID. You can access it from the deployment URL in the vRealize Automation Client.

Results

Verify that the POST request is successful in the vRealize Automation Client.

Day 2 Operations on IaaS Entities

5

This section discusses changes between vRealize Automation 7.x and vRealize Automation 8.x actions.

vRealize Automation actions can be separated in three different categories. The examples presented here use the Postman API platform.

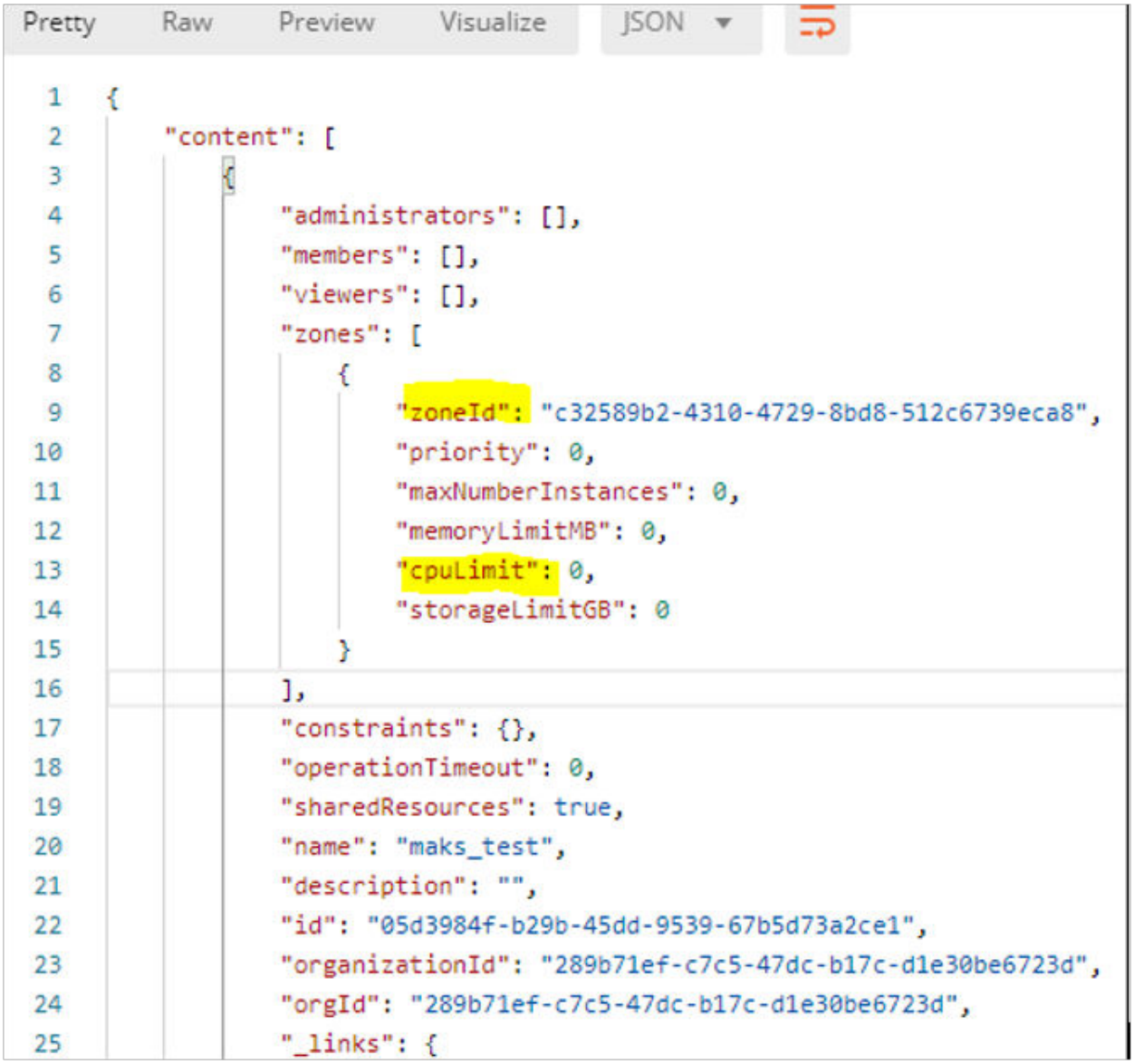
Out of the box actions

Out of the box actions can be categorized as follows:

- There are equivalent actions in vRealize Automation 8.x, such as **Create Snapshot**. For this category of actions, no changes are required.
- There is no replacement, such as **Get Expiration Reminder**. You must either remove these missing actions from your development lifecycle, or create custom actions to perform the required function.
- There are new actions in vRealize Automation 8.x, such as **Revert To Snapshot**. Since these actions are new to vRealize Automation, no changes are required.

Custom actions

Custom actions are user written workflows. You export these workflows to vRealize Orchestrator 8.x as part of a package and, if applicable, replace the vRealize Automation 7.x API calls. For example, to add a vCPU to a virtual machine, you look up the cloud zone quota before adding the new resource. The following sample includes the output of a GET call to `/iaas/api/projects/{id}`. The zone `cpuLimit` has a value of zero which means that there is no limit to this parameter.



In Cloud Assembly, you create a resource action and add a binding between the vRealize Orchestrator `VC:VirtualMachine` input type used in the workflow and the vRealize Automation Cloud Assembly `Cloud.vSphere.Machine` resource type. To account for the other input parameters in the workflow, you can customize the request form that users see when they request the action. For an example of implementing custom actions, see *How to create a vRealize Automation Cloud Assembly custom action to vMotion a virtual machine* in *Using and Managing vRealize Automation Cloud Assembly*.

vRealize Automation 7.x specific actions

Some vRealize Automation 7.x concepts are not valid in vRealize Automation 8.x, such as ownership per virtual machine. Instead, vRealize Automation 8.x has ownership per deployment. Deployments can be shared among project members.

As a vRealize Automation 7.x user, you can write a day 2 action that changes virtual machine ownership from a provisioning user to an end user. To do the same in vRealize Automation 8.x, you must:

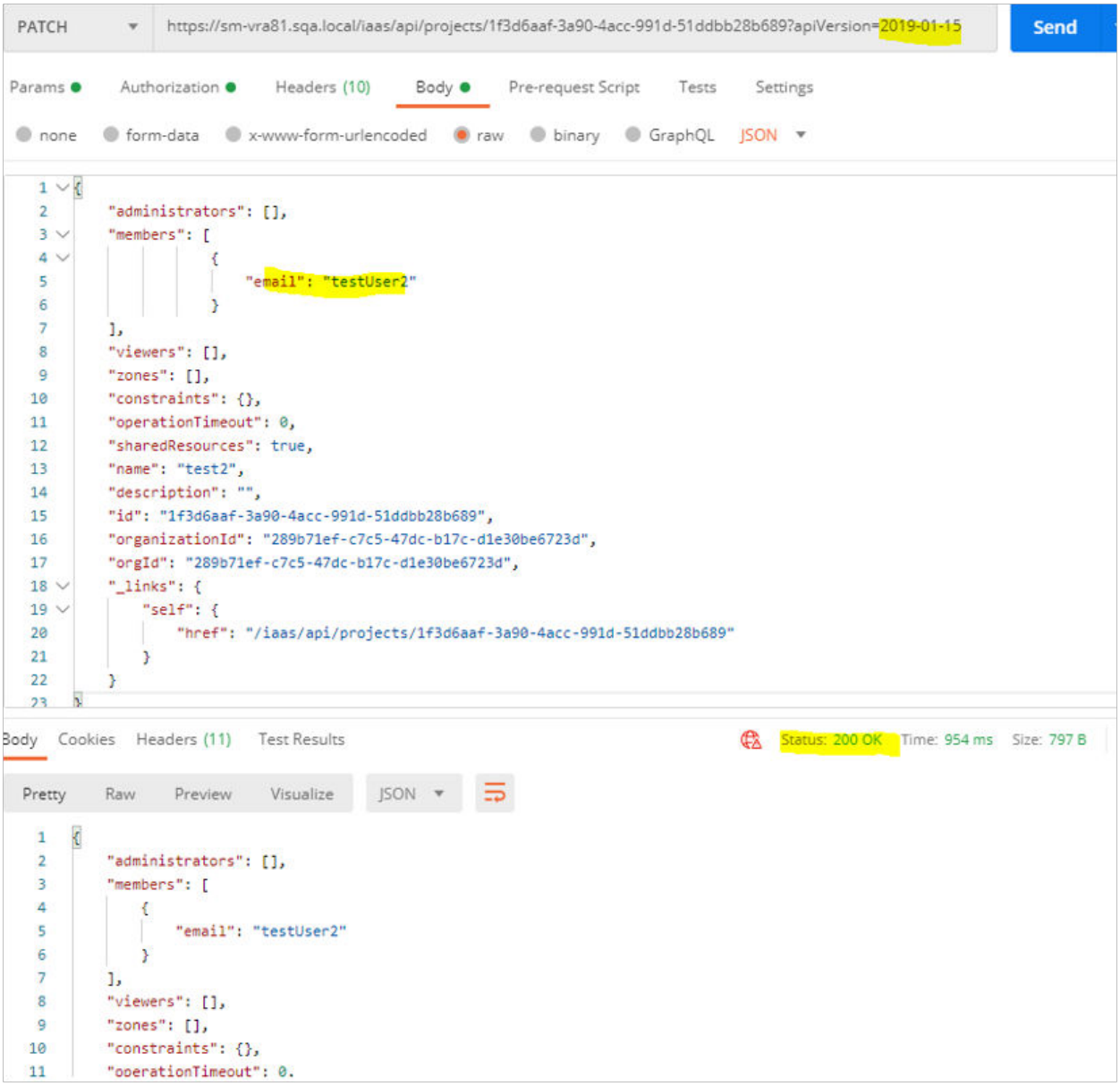
1 Enable deployment sharing for your project.

2 Add all users to the project.

Alternatively, you can write a workflow that adds a user to the project by using the following project API call:

```
PATCH /iaas/api/projects/{id}
```

This workflow must have two input parameters, one for project name and one for user name. You create a catalog item for this workflow, and you apply a request form to the catalog item. This request uses another workflow to retrieve project names. For example, this API call adds `testUser2` to a project:



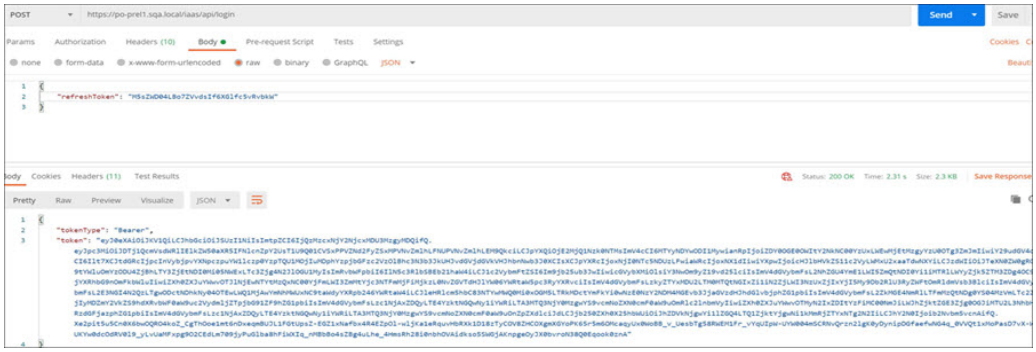
This chapter includes the following topics:

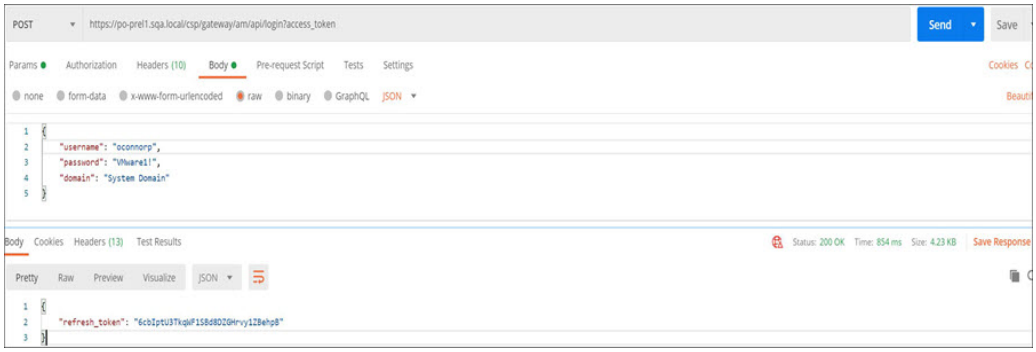
- Custom Form API Call Examples

Custom Form API Call Examples

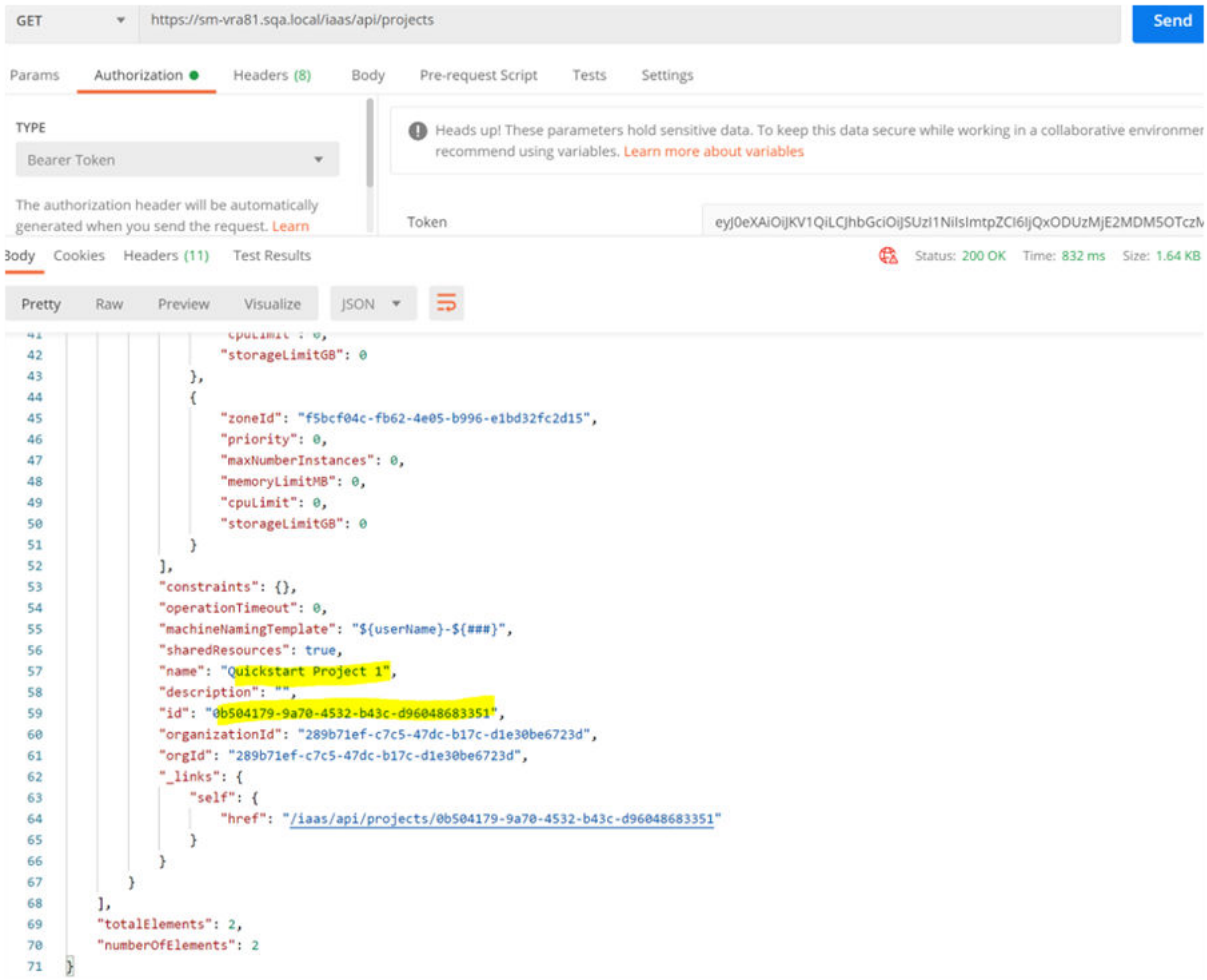
You can request a vRealize Automation 8.x catalog item with custom forms that use API calls. All examples in this scenario use the Postman API platform.

Obtain a bearer token and refresh token

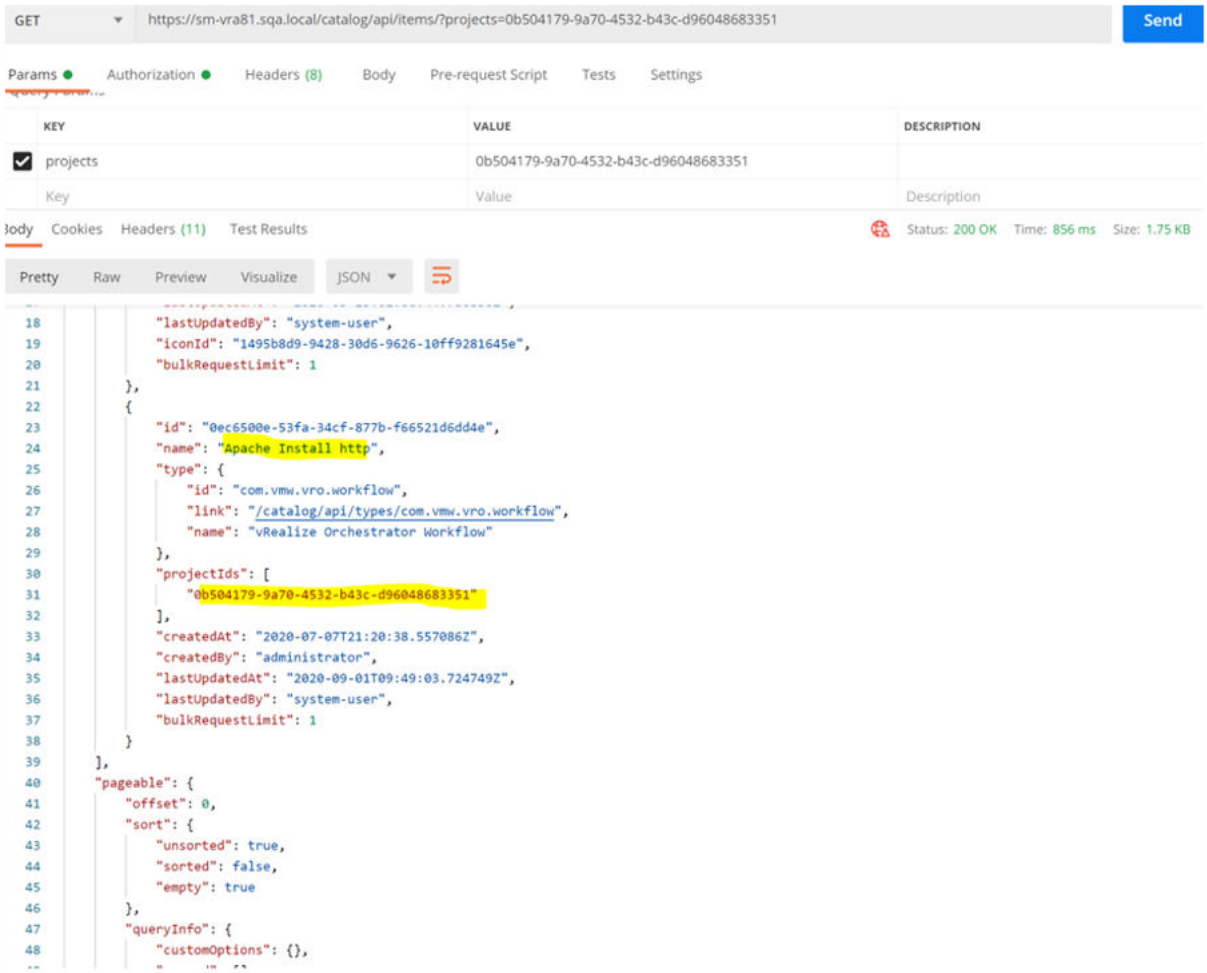




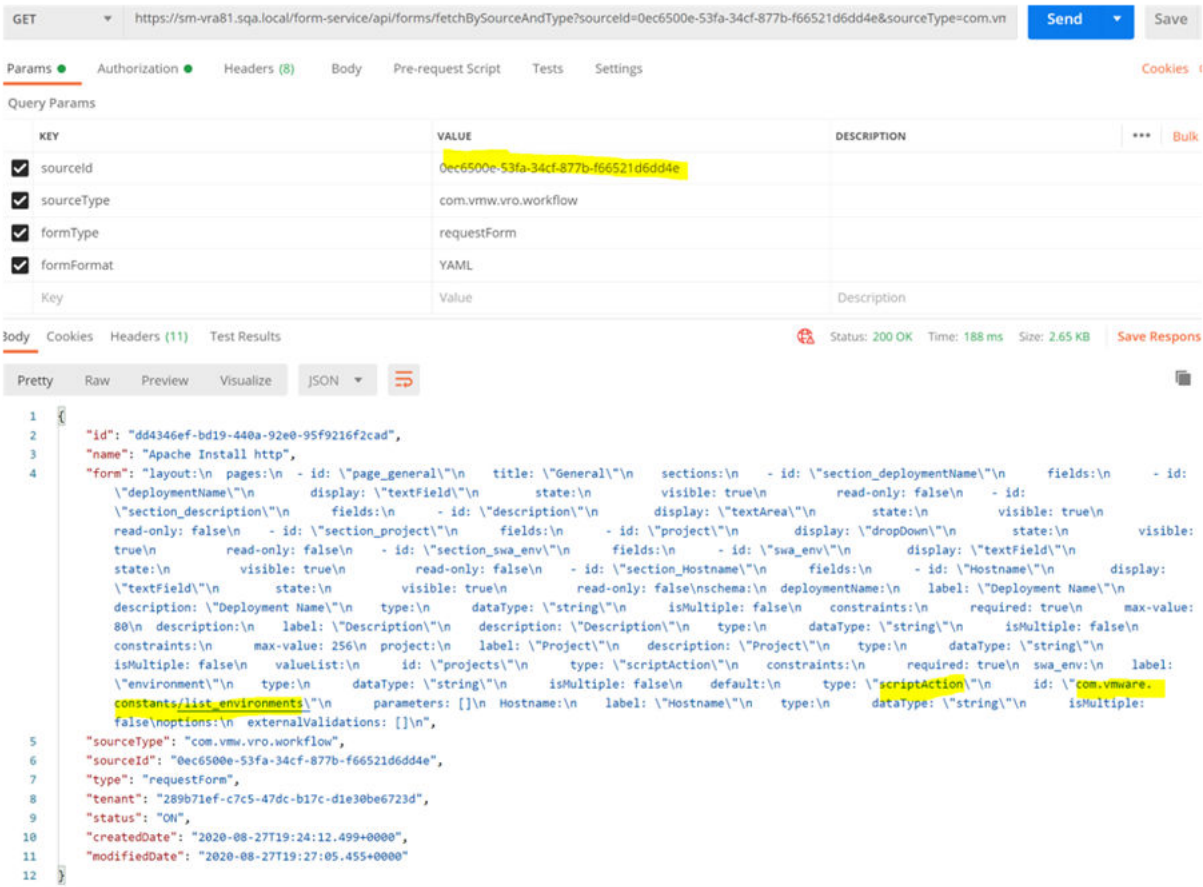
Retrieve a project ID



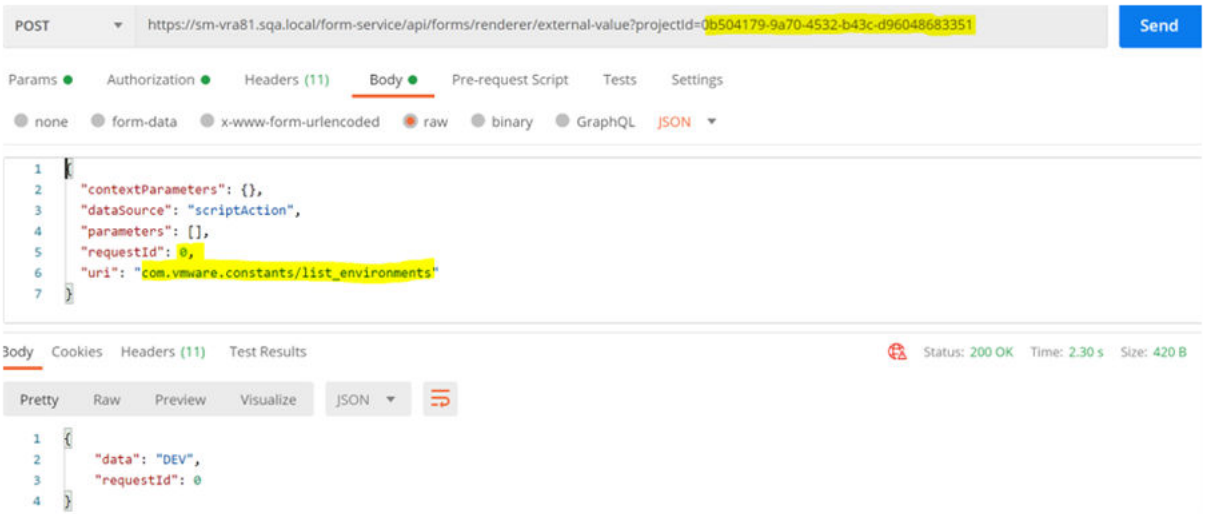
Retrieve a list of catalog items by using a project ID



Look up a catalog item that uses a custom form



Run a script action in a custom form to retrieve data

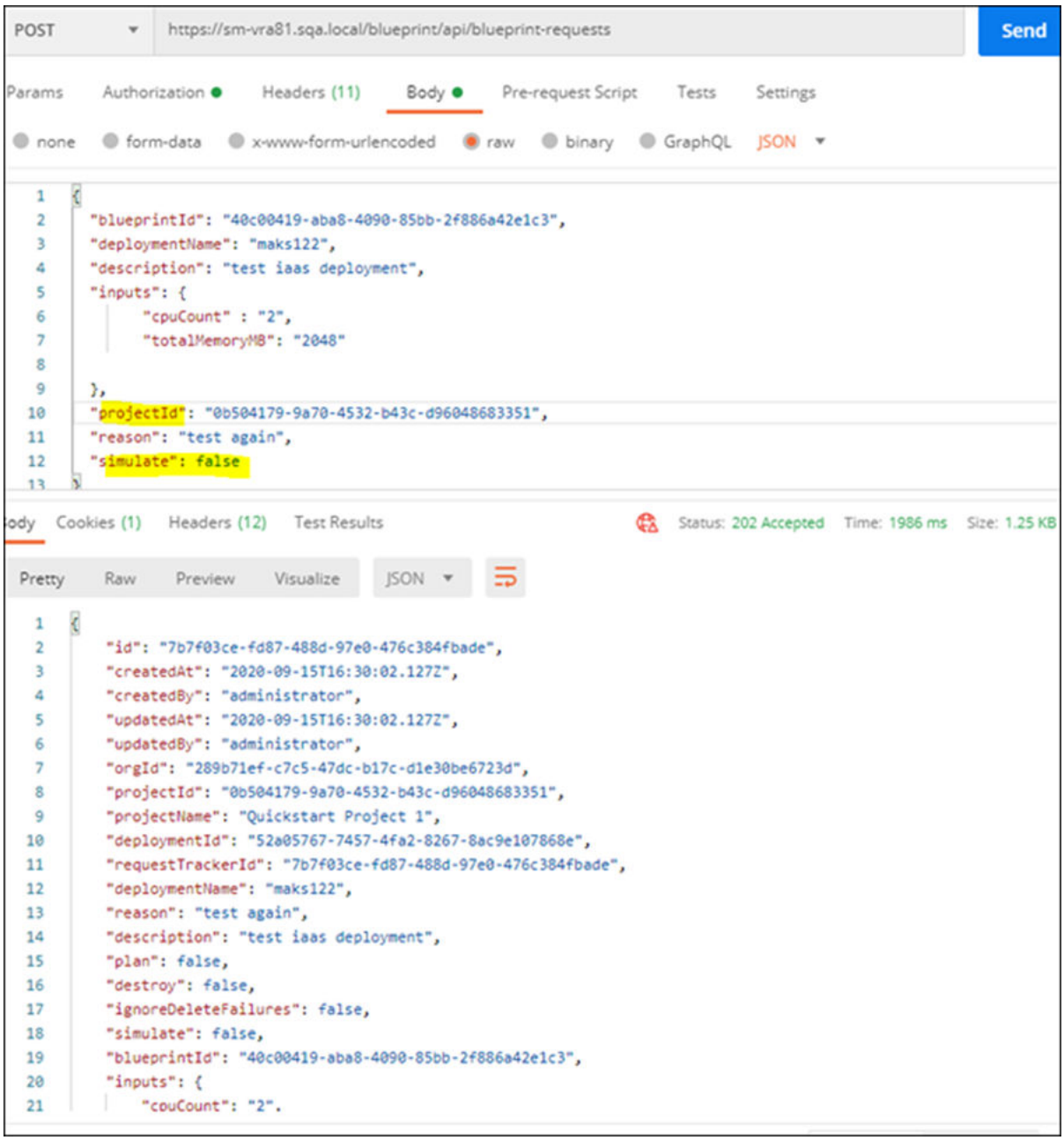


Submit a cloud template request

The forms service API in vRealize Automation 8.1 does not support form execution. You cannot request a catalog item that uses a custom form to capture user inputs. As a workaround, you can use two API calls:

- Form service API to retrieve input data.
- Cloud template API to submit the request.

> **Note** Cloud templates were previously known as blueprints.



Using Dynamic Types with Custom Resources in vRealize Automation Cloud Assembly

6

You can expand the functionality of your vRealize Automation cloud templates by using dynamic types-based custom resources.

When you create cloud templates in vRealize Automation Cloud Assembly, you can use different resource types. Examples of resource types include Amazon S3 Buckets, Cloud Agnostics Machines, NSX networks, vSphere Virtual Machines, Microsoft Azure Resource Groups, and others.

You can use vRealize Automation Cloud Assembly to create custom resources for use cases that are not covered by the preconfigured resource types.

Each custom resource is based on a vRealize Orchestrator SDK inventory type and is created by a vRealize Orchestrator workflow that has an output which is an instance of your desired SDK type. Primitive types, such as `Properties`, `Date`, `string`, and `number` are not supported for the creation of custom resources. You can add custom resources to your cloud template design canvas for use during you lifecycle extensibility deployments.

Note SDK object types can be differentiated from other property types by the colon (":") used to separate the plug-in name and the type name. For example, `AD:UserGroup` is a SDK object type used to manage Active Directory user groups.

For more information on vRealize Automation Cloud Assembly custom resources, see *How to create custom resource types to use in vRealize Automation Cloud Assembly cloud templates* in *Using and Managing vRealize Automation Cloud Assembly*.

The sample workflows included with the sample package in this guide, contain a generic implementation for basic dynamic types objects. The dynamic types sample code creates the object definition, including the dynamic types namespace, if required. All instance of the defined objects are stored in a custom resource as a JSON string. This approach can help speed up vRealize Automation custom resource prototyping with dynamic types.

The current guide includes a use case that demonstrates this functionality with a example based on storing additional metadata related to web servers that are deployed by vRealize Automation 8.x. In this use case, you use a dynamic types based custom resource to store information about the website that the deployed web server hosts.

In addition to the example provided below, that demonstrates how you can create dynamic types from scratch, you can also leverage the Dynamic Types plug-in generator version 3 to automatically create a plug-in from a OpenAPI/Swagger definition or from individual HTTP REST GET URLs. To download the plug-in generator, see Dynamic Types plug-in generator version 3.

This chapter includes the following topics:

- Creating the Dynamic Types Configuration
- Dynamic Types Object and Custom Resource Requirements
- Create the Dynamic Types Custom Resource

Creating the Dynamic Types Configuration

Before you can begin creating your custom resource, you must first create the necessary dynamic types configuration.

The presented configuration is created through the dynamic types plug-in. To create the configuration, run the `ConfigureDynamicTypes` workflow included in the sample package. The dynamic type configuration has the following parameters:

Parameter Type	Value
Namespaces	Websites
Object Type	Site
Properties for Site object	domain, host, euro, lease **Note** The dynamic types plug-in only supports strings as property values.
Object Type	SiteFolder **Note** All dynamic types objects are required to have a parent folder, so you are required to create a SiteFolder object.
Relationship	SiteFolder-Site

The above parameters represent the inputs of the working example. You have the **`Websites`** namespace and an object type called **`Site`**. In addition to these input parameters, you are also specifying four additional properties, **`host`**, **`euro`**, **`domain`**, and **`lease`**, that are configured with the default name and ID properties. In the vRealize Orchestrator inventory the objects that are created are listed with the name property displayed.

When this workflow finishes running, you can navigate to the inventory section of the vRealize Orchestrator Client and review the dynamic types inventory. In the inventory, you should see the **`Websites`** namespace and the **`SiteFolder`** parent folder.

Dynamic Types Object and Custom Resource Requirements

After configuring the dynamic types plug-in, you can create some dynamic types object to test the new dynamic types configuration.

You can create an instance of your new dynamic types object by running the `Create Website Object` workflow that is included in the sample package. This workflow creates a dynamic types object called `Websites Site`, which includes the input parameters required for your custom resource.

Note The `Create Website Object` workflow generates an ID for the newly created object if no ID is supplied when first calling the action. Depending on your use case, you might want to supply the ID for these objects by specifying the ID property when creating the object.

Note The object types follows the `namespace.object` format. For this use case, the object type would be `Websites.Site`.

When the workflow finishes running, you should see the new object in the dynamic types plug-in inventory under the `Sites` folder.

The data backing the object displayed in the dynamic types inventory is saved as a custom resource under the `VMware/PVE/dynamictypes/dataPersistance` folder.

Regarding the custom resource itself, there are key requirements for Create and Delete workflows:

- The Create workflow must have string type inputs for each required object property.
- The Create workflow must have a dynamic types object as the only output for the workflow.
- The Delete workflow must have single dynamic types object input.

The sample package has workflows for both create and delete website objects.

Create the Dynamic Types Custom Resource

After configuring the dynamic types plug-in and creating some test objects, you must create the custom resource definition in Cloud Assembly.

Procedure

1 In Cloud Assembly, select **Design > Custom Resource**, and click **New Custom Resource**.

2 Provide the following values:

Setting	Sample Value
Name	**`Website`** This is the name that appears in the cloud template resource type palette. You can use another name if desired.
Resource Type	**`Custom.website`** The resource type must begin with **`Custom.`** and each resource type must be unique. Although the inclusion of **`Custom.`** is not validated in the text box, the string is automatically added if you remove it. This resource type is added to the resource type palette so that you can use it in the cloud template.
Activate	To enable this resource type in the cloud template resource type list, verify that **Activate** option is toggled on.
Scope	Define if you want this custom resource to be shared across projects or specific to a single project.
Lifecycle Actions - Create	Select the **Create Website Object** workflow. If you have multiple vRealize Orchestrator integrations, select the workflow on the integration instance you use to run these custom resources. After selecting the workflow, the external type drop-down menu becomes available. **Note** An external source type can be used only once if shared and once per project. In this use case, you are providing the same custom resource for all the projects. It does mean that you cannot use the same external type for any other resource types for all projects. If you have other workflows that require the selected type, you must create individual custom resources for each project.
Lifecycle Actions - Destroy	Select the **Delete Website Object** workflow.

3 To finish creating the custom resource, click **Create**.

Results

You have created a sample custom resource definition that uses the dynamic types plug-in.

What to do next

When you create a cloud template, the website object should now be available from the left resource pane and can be dragged into the cloud template canvas. After deploying the cloud template, a instance of the site object is displayed in the dynamic types plug-in inventory. Similarly, if the deployment is destroyed, the instance of the site object is removed from the from the dynamic types plug-in inventory.

Lifecycle Extensibility

7

vRealize Automation provides pre-defined application and services life cycles operations with some level of applicable configurations. However each user has specific processes and integrations that require customizing this life cycle via extensibility.

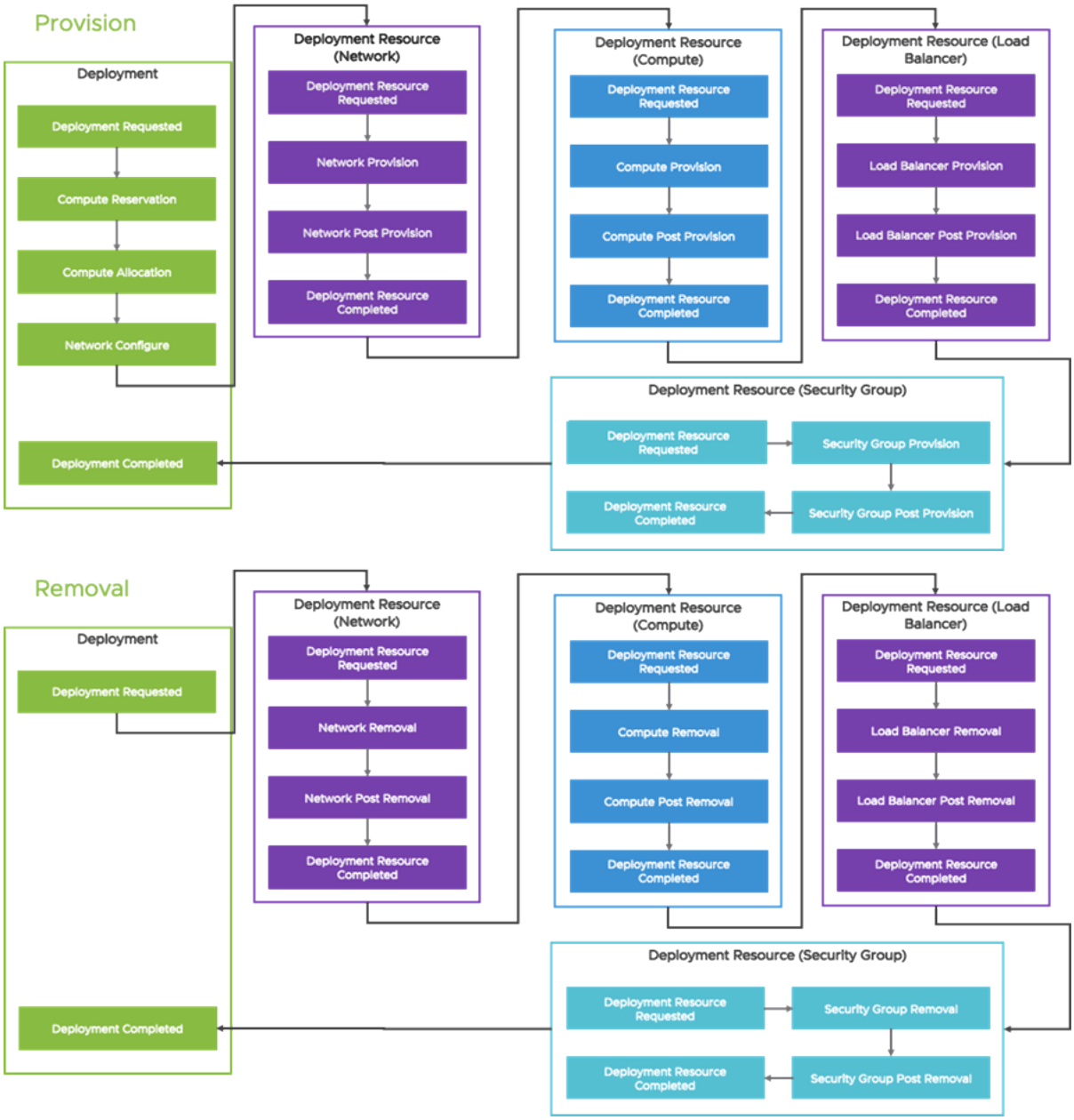
In vRealize Automation 7.x and vRealize Automation 8.x lifecycle extensibility is applied through the Event Broker service. The most common use case are related to the machine provisioning that supports different subscriptions including for example:

- Pre-provisioning to take action on third party systems or to modify the provisioning configuration.
- Post-provisioning to run an operation on the provisioned resources.
- Notify or record request provisioning data in external systems.

Event Broker subscriptions exist in vRealize Automation 8.x, but they:

- Use different event topics. vRealize Automation 8.x event topics are similar to vRealize Automation 7.x event topics, but are not identical.
- vRealize Automation 8.x subscriptions use different a payload to pass parameters.
- vRealize Automation 8.x subscriptions use different metadata.
- vRealize Automation 8.x subscriptions use a different approach to create criteria to filter the cases where the subscription start a workflow.
- Action-based extensibility can be used to provide function as a service (FaaS) operations for on-premises and cloud deployments.

vRealize Automation 8.x provisioning event topics are redesigned with a set of high level topics calling deployment resources topics:



This chapter includes the following topics:

- Migrating Subscriptions from vRealize Automation 7.x to vRealize Automation 8.x
- Creating a Subscription
- Create a Wrapper Workflow

Migrating Subscriptions from vRealize Automation 7.x to vRealize Automation 8.x

You can migrate Event Broker subscriptions from vRealize Automation 7.x to vRealize Automation 8.x.

For vRealize Automation 7.x event topics that have equivalent topics in vRealize Automation 8.x, you can use the vRealize Automation Migration Assistant.

You can also migrate event topics manually by using the following mapping as reference:

vRealize Automation 7.x Workflow State	vRealize Automation 8.x Event Topic	Notes
Catalog request received	Deployment requested	None
N/A	Compute.Reservation.Pre	Changes Placement
N/A	Compute.Allocation.Pre	Overrides Allocations
N/A	Network.Configure	Issued for network selection. The event overrides the IPAM Integration.
Blueprint component requested	Deployment resource requested	None
Requested WaitingToBuild BuildingMachine PRE	Compute.Provision.Pre	Deployed before the instance was deployed.
InitialPowerOn PRE InitialPowerOn EVENT	compute.initial.power.on	Issued before initial power on. Used for additional VM customization.
BuildingMachine POST MachineProvisioned MachineActivated	Compute.Provision.Post	Posts the machine online.
Blueprint component completed	Deployment resource completed	None
Catalog Item Requested completed	Deployment completed	None
N/A	Deployment requested "eventType": "DESTROY_DEPLOYMENT"	None
N/A	Deployment resource requested "eventType": "DELETE_RESOURCE"	
Deactivate Unprovision Disposing Pre	Compute.Removal.Pre	Issued before a machine is destroyed.
Disposing Event Disposing Post	Compute.Removal.Post	Ussed after a machine is destroyed.
N/A	Deployment resource completed	None
N/A	Deployment resource requested	None

N/A	Network removal	None
N/A	Network post removal	None
N/A	Deployment resource completed	None
N/A	Deployment completed	None
Power Off		None
On – EVENT On – POST TurningOff – PRE	Deployment resource action requested "actionName": "PowerOff" "status": ""	None
TurningOff – POST Off – PRE	Deployment resource action completed "actionName": "PowerOff" "status": "FINISHED"	None
Power On		None
Off – EVENT Off – POST TurningOff – PRE	Deployment resource action requested "actionName": "PowerOn" "status": ""	None
TurningOn – POST On – PRE	Deployment resource action completed "actionName": "PowerOn" "status": "FINISHED"	None

Creating a Subscription

You can use event topics as part of Event Broker subscriptions to define lifecycle extensibility.

To select the most appropriate event topic, it is important to evaluate if the event is triggered at the right step of the process and if it carries the payload necessary to perform the extensibility operation.

The payload can be identified with selecting the different event topics.

The **Read Only - No** tag is used for properties that support both read and write operations. With read and write operations, it is possible to use a workflow output to set the property back in vRealize Automation. To do this, it is mandatory to set the subscription to be blockable. For more information on blockable extensibility subscriptions, see *Blocking event topics* in *Using and Managing vRealize Automation Cloud Assembly*.

The following are some of the event topics support setting properties:

- **Compute reservation** is used to change the placement.
- **Compute allocation** is used to change resource names or hosts
- **Compute post provision** is used to after deployment resources are provisioned.
- **Network configure** is used to configure the network profile and individual network settings.

For more information on event topics included in vRealize Automation 8.x, see *Event topics provided with Cloud Assembly* in *Using and Managing vRealize Automation Cloud Assembly*.

Extensibility subscriptions in vRealize Automation 8.x work similarly to the subscriptions included in vRealize Automation 7.x. However, there are some key differences:

- You cannot bind a workflow for all events anymore.
- The conditions for running the subscription are now based on JavaScript.
- Subscriptions can be scoped to individual projects or configured to be shared across all projects in a given organization.
- You can set a recover workflow in case the subscription workflow fails.
- Timeout behavior is similar with some differences:
  - vRealize Automation uses a timeout for the workflows being started by Event Broker blocking subscriptions. If a workflow run lasts more than the set timeout period, then it is considered failed by vRealize Automation.
  - In vRealize Automation 7.x, the default timeout value for all states and events is 30 minutes and is configured in the vRealize Automation global settings.
  - In both vRealize Automation 7.x and vRealize Automation 8.x a timeout value can be set at the subscription level.

    Note The default timeout period in vRealize Automation 8.x is 10 minutes and that you should change the project request timeout if it is lower than the subscribtion timeout.

  - In vRealize Automation 7.x, it is also possible to configure individual state and event timeout values by changing configuration files in the IaaS server.
- Priority defines the order of running blocking subscription where 0 means highest priority and 10 means lowest priority. The default value is 10.

Create a Wrapper Workflow

Some vRealize Automation operations require you to create a wrapper workflow in vRealize Orchestrator.

You can design a wrapper workflow from scratch or duplicate the sample `Event Broker template` workflow included in the sample package and modify it as required.

We call it the "wrapper" workflow because it is often a workflow that connects vRealize Automation to vRealize Orchestrator workflows. For example, extracting data from the payload, finding a VM object in the vRealize Orchestrator inventory by ID, and starting another workflow by taking the action on this VM.

The first requirement for creating a wrapper workflow is that it must have the single payload input of the Properties type named `inputProperties`. This is different from vRealize Automation 7.x where the input can be named anything as long as it was of the Properties type.

In this wrapper workflow, you might need to retrieve specific information from the `inputProperties` input or system context metadata. Similarly to vRealize Automation 7.x, this is done with the `inputProperties.get(parameterName);` and `System.getContext().getParameter("metadataName");` methods, except the parameter and metadata names are changed and can be identified in the **Event Topic** and **Workflow Run** tabs in Cloud Assembly.

A good practice for wrapper workflows is to have a first "Get payload and execution context" element, as either a scriptable task or action element, that retrieves the required information. You can bind these elements as a output to the workflow variables and use them as input parameters in subsequent elements, such as scriptable tasks, actions, and workflows.

Retrieving the individual properties from from the Properties type `InputProperties` is done through the GET method.

The returned properties value can be of the type string, number, boolean, or an array of any of these or complex properties which maps to the Properties type in vRealize Orchestrator.

Many of these properties are object IDs that need further processing to retrieve useful information.

For example, retrieving some information from the catalog is done as follows (code snippet from the `Create an Event Broker subscription` workflow sample):

```
var catalogItemId = inputProperties.get("catalogItemId");
if (catalogItemId != null && catalogItemId !="") {
    var catalogItemObject = getObjectFromUrl("/catalog/api/items/" + catalogItemId);
    if (catalogItemObject != null) {
        System.debug(getPropertiesText(object2Properties(catalogItemObject), "Catalog
Item\n", 1));
        System.log("CatalogItem ID : " + catalogItemObject.id);
        System.log("CatalogItem name : " + catalogItemObject.name);
        System.log("CatalogItem description : " + catalogItemObject.description);
        System.log("CatalogItem type name : " + catalogItemObject.type.name);
        System.log("CatalogItem created By : " + catalogItemObject.createdBy);
    }
}
```

This example can be used to retrieve a vCenter VM (code snippet from the `Create an Event Broker subscription` workflow sample):

```
try {
    if (inputProperties.get("componentTypeId") == "Cloud.vSphere.Machine") {
        var vcUUID = inputProperties.get("customProperties").get("vcUuid")
        var vmUUIDs = inputProperties.get("externalIds");
        for each(var vmUUID in vmUUIDs) {
            vCenterVM =
System.getModule("com.vmware.vra.extensibility").getVCenterVMByUUID(vcUUID, vmUUID);
            if (vCenterVM != null) {
                System.log("Got vCenter VM " + vCenterVM.name + " with ID " + vCenterVM.id);
            }
```

```
        }
    }
} catch (e) {
    System.warn(e);
}
```

This example can be used to retrieve metadata properties below (code snippet from the `Create an Event Broker subscription` workflow sample):

```
// The execution context is where the vRA extensibility metadatas are passed
var executionContext = System.getContext();


// Getting specific execution context parameters
var eventTopicId = executionContext.getParameter("__metadata_eventTopicId");
var eventId = executionContext.getParameter("__metadata_id");
var isEventBlocking = executionContext.getParameter("__metadata_hdr_blocking");
var orgId = executionContext.getParameter("__metadata_orgId");
```

Read and write parameters can be configured by creating workflow outputs matching their name and types.

Another important element of working with wrapper workflows is using tags. The following example shows you how you can add a tag:

```
// Adding TAG
tags = inputProperties.get("tags");
if (tags == null) tags = new Properties();
tags.put("serviceLevel", "Gold");
```

The payload and metadata parameters values and the output values set by your workflow can be monitored by navigating to **Extensibility > Activity > Workflow Runs**.

The sample workflows include a `Create an Event Broker subscription` workflow, which can be used to automate the creation of subscriptions, and a `Create sample "Event Broker Template" subscriptions` workflow, that creates a subscription for each event topic starting the `Event Broker Template` workflow. This workflow provides the following capabilities:

- Displaying the content of the payload.
- Displaying the content of metadata.
- Provides an example on reaching back to vRealize Automation to retrieve the properties of the objects provided as IDs in the payload.
- Provide an example on converting payload IDs to vRealize Orchestrator objects to bind the operation workflow on the object. You can use this to convert to `VC:VirtualMachine` to create a snapshot.
- Display the parameters that support being changed with workflow outputs.
- Update custom properties.

- Update tags.
- Update VM names.
- Get host selections.

  There is a different Event Broker template workflow under the `inventory objects` folder that retrieves the plug-in inventory objects available from the payload. This allows you to set workflow variables to objects such as projects to bind them as input parameters of other workflows.

Onboarding a Customer Organization

8

There are several key concepts and requirements, you must be aware of before onboarding a customer organization.

There is a significant amount of configuration involved in setting up IaaS infrastructure, so it can make resources available to end users. In vRealize Automation 7.x, configuration is done with business groups, reservations, and so on.

Many vRealize Automation users have automated onboarding for customers. Automated onboarding includes importing data from other systems and using specific naming conventions.

In vRealize Automation 8.x, the concepts for assigning resources and providing entitlements to content have changed. Now this is done with projects, zones, and flavors. All of these components are listed under the **Infrastructure** tab of Cloud Assembly.

vRealize Automation 8.x includes a guided setup wizard that guides you through the steps needed to create a cloud account, project, zone, and images by assigning a default configuration. To access this wizard, click **Guided Setup** on the top-right of the user interface.

While this wizard is useful for getting started, it does not address the organization onboarding scenario as some steps require end user inputs, external integrations and further granularity for some settings.

Configuring the onboarding organization infrastructure is done by automating the creation of the required objects, such as cloud accounts, projects, zones and others. This process can include the entire configuration or only the setting up the components where automation and integration provides more value.

This must be done by creating individual workflows or actions that create each object as needed. Afterwards, these objects are incorporated on a master workflow, that automates the whole process.

This process must follow as specific order, because some objects are dependant on other objects existing first. The order is as follows:

1 Cloud account

2 Zone

3 Project

4 Flavor mapping

5 Image mapping

6 Network profile

7 Storage profile

Deleting objects must also follow a specific order, because you cannot delete an object that is being used or referenced by another object. This process also has more steps than the deployment workflow as it requires deleting all the objects that can possibly be created by end users, such as deployments, Code Stream pipelines and others. The following is a non-exhaustive list of objects based on deletion order.

1 Integrations

2 User operations

3 Pipelines

4 Endpoints

5 Variables

6 Action runs

7 Workflow runs

8 Subscriptions

9 Extensibility actions

10 Storage profiles

11 Drafted cloud templates

12 Deployed cloud templates

13 Deployments and resources

14 Projects

15 Zones

16 Cloud accounts

Note The delete operation might not always finish before the deletion of the resources. It might be necessary, in case of deletion failure because of a dependent resource is not yet deleted, to wait further in the workflow before retrying deletion. Also, to delete a project, you must first patch it to remove the dependencies on all its zones.

This chapter includes the following topics:

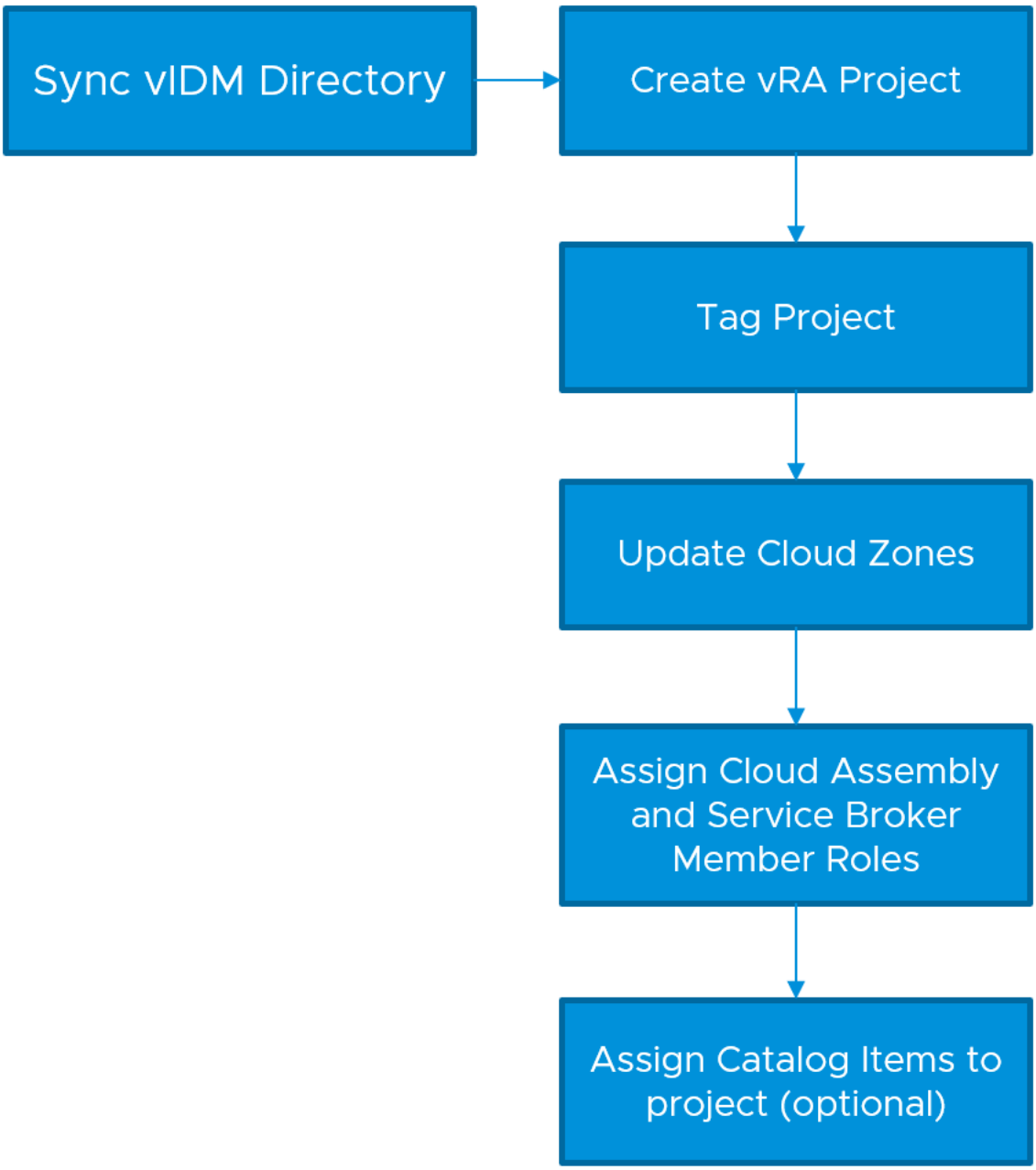
- Onboarding a Project

Onboarding a Project

This topic includes a scenario that demonstrates how you can onboard a project.

This scenario goes through the steps needed to automate the repeatable components of onboarding a project in vRealize Automation to enable self-service consumption.

The following diagram presents the main flow of this scenario:



Sychronizing the Workspace ONE Access Directory

Synchronize with the Workspace ONE Access directory by using a POST API call.

This scenario is optional depending of your end to end process, but if you are also automating the creation of Active Directory (AD) groups for your project, these groups must be synchronized in the Workspace ONE Access service (previously known as VMware Identity Management) before they can be associated to a project for user access. In this use case, you are using the Lifecycle Manager (LCM) API to perform the synchronization operation. LCM performs the downstream synchronization call to Workspace ONE Access .

The call is perform by using the POST method with the following URL:

```
/lcm/authzn/api/idp/dirConfigs/syncprofile/sync
```

The request itself has the following content:

```
{
 "directoryConfigId": directoryId,
 "directoryType": "ActiveDirectory",
 "isGetBeforeUpdate": true,
 "isTenantConfiguredByPath": true,
 "vidmAdminPassword": password,
 "vidmAdminUser": username,
 "vidmHost": hostname
}
```

Creating a vRealize Automation Project

While creating a project, you can also specify the users and administrators that are part of the project.

You can create a project by using a POST API call that uses the `/project-service/api/projects` URL to create a project.

```
{
 "administrators": [
 {
 "email": "${ADMINISTRATOR_VIDM_GROUP}",
 "type": "group"
 }
 ],
 "members": [
 {
 "email": "${USER_VIDM_GROUP}",
 "type": "group"
 }
 ],
 "viewers": [],
 "zones": [],
 "constraints": {},
 "operationTimeout": 0,
 "sharedResources": true,
 "name": "${PROJECT_NAME}",
 "description": "",
 "orgId": "${ORGANISATION_ID}",
```

```
  "properties": {}
 }
```

To run this command, the following information is required. AD groups should be synced by using a directory configuration in LCM.

- The group name to be assigned the project administrators role.
- The group name to be assigned the project member role.
- A name for the new project.
- The vRealize Automation organization ID. This ID can be retrieved through with the vRealize Automation API.

Associating a Tag with the Project

In this example, you are associating a tag with the newly created project. The tag is inherited onto the workloads provisioned from this project.

To associate a tag with a object, you can use a PATCH API call that uses the `/iaas/api/projects/${projectId}/resource-metadata` URL.

```
{
 "tags": [
 {
 "key": "costCode",
 "value": "${costCode}"
 }
 ]
}
```

Add Cloud Zones to the Project

The next step in the onboarding scenario is to add one or more cloud zones to the project. With the API, you can add cloud zones.

You can add cloud zones by using a PATCH API call that uses the `/iaas/api/projects/${NEW_PROJECT_ID}` URL.

```
{
 "zoneAssignmentConfigurations": [
 {
 "storageLimitGB": 0,
 "cpuLimit": 0,
 "memoryLimitMB": 0,
 "zoneId": "${CLOUDZONE_ID1}",
 "maxNumberInstances": 0,
 "priority": 0
 },
 {
 "storageLimitGB": 100,
 "cpuLimit": 100,
 "memoryLimitMB": 100,
```

```
 "zoneId": "${CLOUDZONE_ID2}",
 "maxNumberInstances": 20,
 "priority": 0
 }
 ]
}
```

Assign Cloud Assembly and Service Broker User Roles

Aside from assigning users at the project level, you must also assign the organisation role to users within vRealize Automation Identity and Access Management service. You assign roles so users have access to their required vRealize Automation services. In this use case, the services are Cloud Assembly and Service Broker.

You can assign roles in vRealize Automation when you first log in as an administrator by navigating to **Identity and Access Management** and assigning the required service roles to the user. For more information on editing user roles from the vRealize Automation user interface, see *How do I edit user roles in vRealize Automation* in *Administering vRealize Automation*.

You can also assign roles by using a POST API call that uses the `/csp/gateway/portal/api/orgs/${ORGANISATION_ID}/groups` URL.

```
{
 "ids": [
 "${GROUP_ID}"
 ],
 "organizationRoleNames": [
 "org_member"
 ],
 "serviceRoles": [
 {
 "serviceDefinitionId": "${CLOUD_ASSEMBLY_SERVICE_ID}",
 "serviceRoleNames": [
 "automationservice:user"
 ]
 },
 {
 "serviceDefinitionId": "${SERVICE_BROKER_SERVICE_ID}",
 "serviceRoleNames": [
 "catalog:user"
 ]
 }
 ]
}
```

Assign Catalog Items to a Project

You can assign catalog items to a vRealize Automation 8.x project by using a API call.

To run the following API call, you must have the following information:

- The project ID. This is returned by the API call when the project is created.

- The cloud template ID. This ID can be attained by listing the cloud templates with GET commands. If the values of these cloud templates are consistent, then this value can be stored in vRealize Orchestrator or in a extensibility action.

You can assign catalog items to a project by using a POST API call that uses the `/catalog/api/admin/entitlements` URL.

```
{
 "projectId": "${projectId}",
 "definition": {
 "type": "CatalogItemIdentifier",
 "id": "${BLUEPRINT_ID}",
 "name": "",
 "description": "",
 "numItems": 0,
 "sourceType": ""
 }
}
```

vRealize Orchestrator Implementation for Project Onboarding

Use a vRealize Orchestrator workflow to implement project onboarding.

The previous scenario for assigning a catalog item is implemented as a vRealize Orchestrator workflow provided in the sample section under **vRA plug-in > REST > Organization infrastructure onboarding > Project Onboarding**. It covers all aspects of onboarding, excluding the optional Workspace ONE Access synchronization.

The following workflow creates IaaS project with:

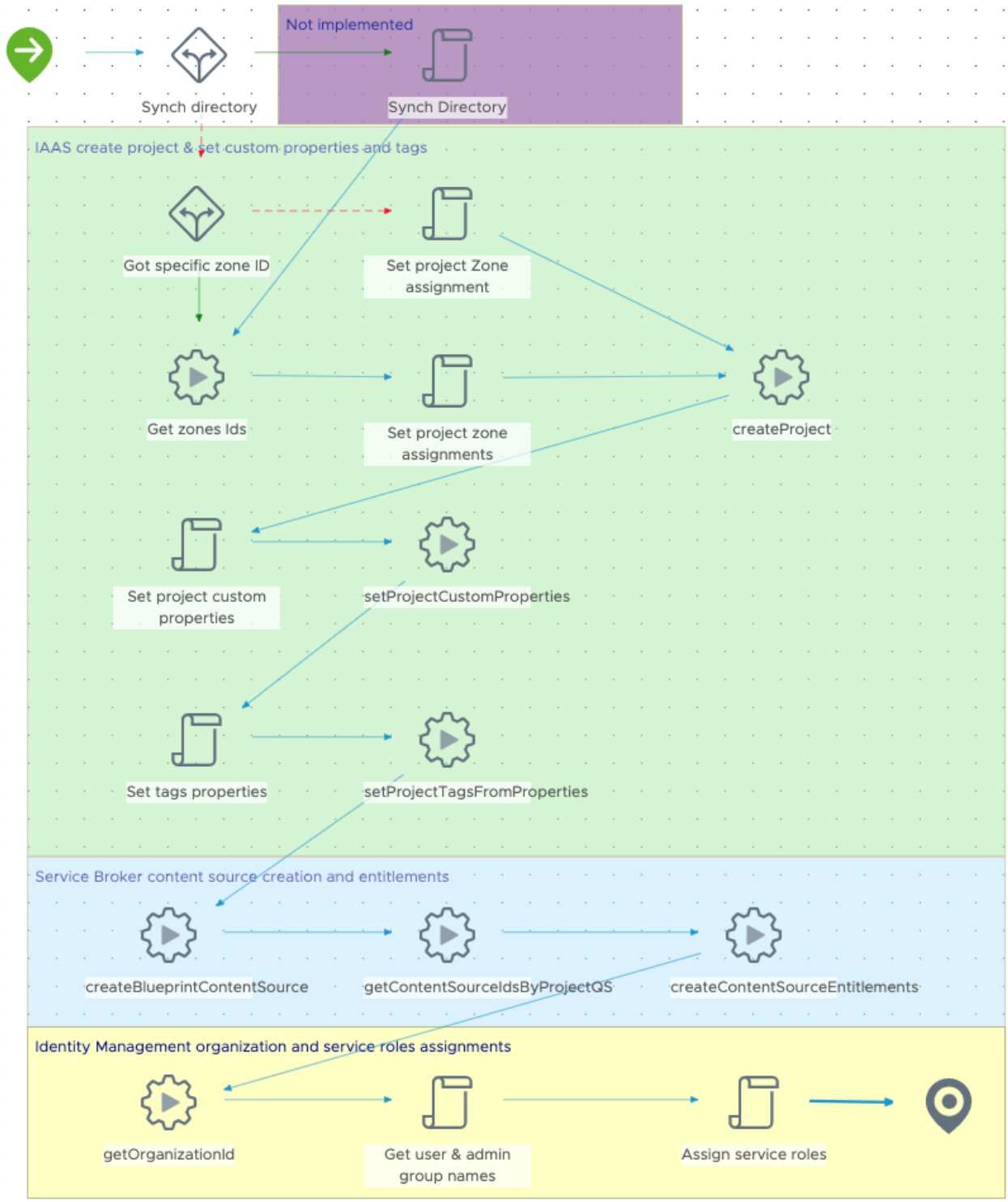
- Setting a single zone if this was provided as input. If no specific zone is provided as an input, the workflow configures all available zones.
- Set the network constraint passed as a input parameter.
- Set tags and custom properties for cost code and project name provided as input.
- Set specific folders for the project and for the environment.
- Set a specific naming template.
- Set administrator and user groups.

The workflow also performs the following Service Broker updates:

- Create a specific content source for the project.
- Share this content source and other sharable content in this project.

Finally, the workflow performs the following Identity Management configurations:

- Assign admin group Cloud Assembly, Service Broker, and Orchestrator administrator roles.
- Assign user group Cloud Assembly and Service Broker user roles.



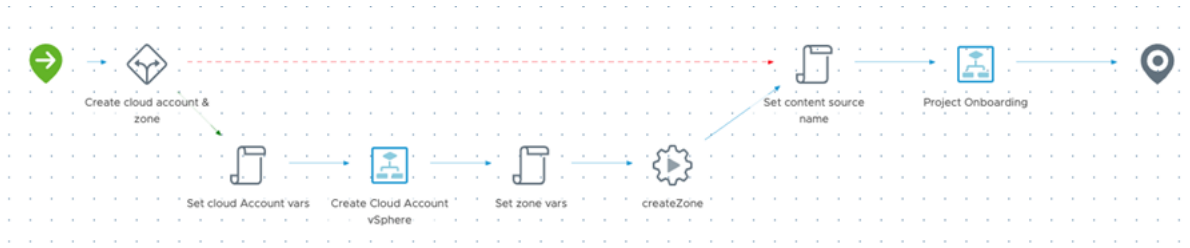
There is another project onboarding workflow under **vRA plug-in > inventory objects > Organization infrastructure onboarding > Project Onboarding** where the custom create project action delivered as part of the samples is replaced by the **Create Project** workflow that is part of the vRealize Orchestrator plug-in for vRealize Automation.

Adding Resource Provisioning to a Project

You can use a vRealize Orchestrator workflow to add resource provisioning to your project.

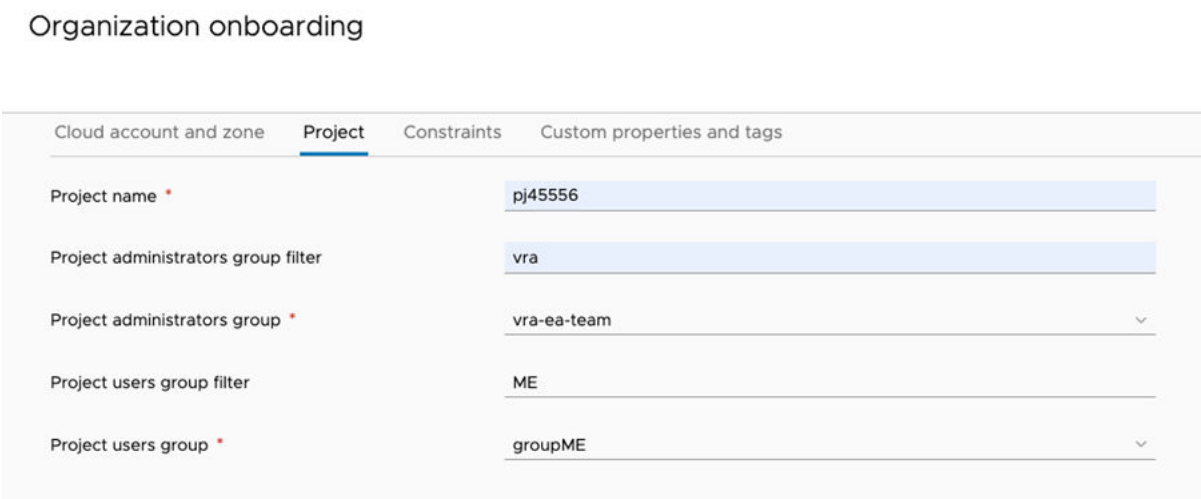
The sample workflow `Organization infrastructure onboarding / Organization Onboarding` is a front end for the project onboarding workflow that enables you to:

- Create a vSphere Cloud account.
- Create a zone for this cloud account.



The workflow inputs are separated in different tabs and include group search with filters.

If a cloud account and a zone are created they are assigned to the project, otherwise the available zones from existing cloud accounts are assigned to the project.



Requesting Catalog Items

9

A key functionality of vRealize Automation is requesting catalog items.

A common scenario seen in vRealize Automation 7.x includes an XaaS catalog item requesting a composite blueprint. While it is still possible to request a catalog item with the API service, this scenario focuses on achieving the same result through customizing the request at runtime using custom forms with dynamic selection of tags.

In vRealize Automation 8.x, form constructs are standardized around the vRealize Automation custom forms designer. Given the change in vRealize Automation APIs with 8.x, this is a good opportunity to remove previous technical debt as you transition and also standardize cloud template deployment by using custom forms.

The below table summarizes some of the common patterns used for placement selection to contrast the options and changes now available in vRealize Automation 8.x.

vRA 7.x - placement (custom logic)	vRA 8.x – placement (standardized on tags)

■ XaaS, custom forms, or IaaS forms
  ■ API - Compute population of reservation policy name and ID selection in code.
  ■ API - Network – Based on naming conventions and reservations population of applicable network profiles.
  ■ API -Storage – Storage reservation population through code.
	■ Custom forms with tag based inputs
	  ■ API - Dynamic Tag selection based on keys and filters
	  ■ Alternatively. using Event Broker read or write properties.

In vRealize automation 8.x, placement logic is standardized by using tags, resources are tagged with capabilities and constraints are applied to a cloud template for the placement engine to select the relevant downstream resources.

This chapter includes the following topics:

- API Tag Filtering Examples
- vRealize Orchestrator Action Example
- Basic Sample Cloud Template
- Associating an External Value with the getTagByKey Action
- Example Service Broker Catalog Request
- Requesting Catalog Items Programatically

API Tag Filtering Examples

As a prerequisite to dynamically populating tag data through the API, this section provides some examples of using the vRealize Automation IaaS API return tags and filters to return suitable tags based on known keys.

Get All Tags

Return a list of all available tags, keys and values.

```
GET /iaas/api/tags
```

Filter Tags by Key

Return all available tags with a key of location. For example the tag can return the `location:newyork` and `location:sydney` values. The core element in this example is that you can return all relevant tags based on your defined key.

```
GET /iaas/api/tags?$filter=key eq 'location’
```

Filter Networks by Tag Key and Value

Filter networks based on the key and value to then find suitable subsequent tags for further filtering.

```
GET /iaas/api/fabric-networks?$filter=tags.item.key eq 'environment' and tags.item.value eq
'dev’
```

Filter Networks by Cloud Account ID and Environment

Filter networks based on cloud account ID and tag key and value, this can be used when you have different placement logic between public cloud and on-premises deployments to display the relevant tags for the target cloud.

```
GET /iaas/api/fabric-networks?$filter=cloudAccountIds.item eq 'ec4822a755a755906c6b3822b2'
and tags.item.key eq 'environment’ and tags.item.value eq 'dev'
```

vRealize Orchestrator Action Example

You can create a generic vRealize Orchestrator action to be used in custom forms to populate external values.

In this scenario, you want the ability to return a list of suitable tags for placement based on an input of tag key.

Having the tag key as an action makes this action reusable regardless of the tag you must return. In this case, you authenticate with vRealize Automation by using a REST host stored in a configuration element. You then make a call to find all suitable tags matching the supplied tag key.

These tag keys and values are pushed into an array to return the values for our drop-down menu. This action returns an array of strings containing the filtered tags.

The `getTagByKey` action below is included in the samples.

```
var url = "/iaas/api/tags"
var parameters = encodeURI("$filter=key eq " + tagKey);

var tags = System.getModule("com.vmware.vra.extensibility.plugin.rest").getObjects(vraHost,
url, parameters);

var tagArray = new Array();
for each (var tag in tags) {
  tagArray.push(tag.key + ":" + tag.value);
}

return tagArray;
```

Basic Sample Cloud Template

You can bind two constraints as inputs in your cloud template.

To demonstrate a simple version of this scenario, you can bind two constraints as inputs in the cloud template yaml, platform, and environment. These constraints enable your cloud account and compute placement.

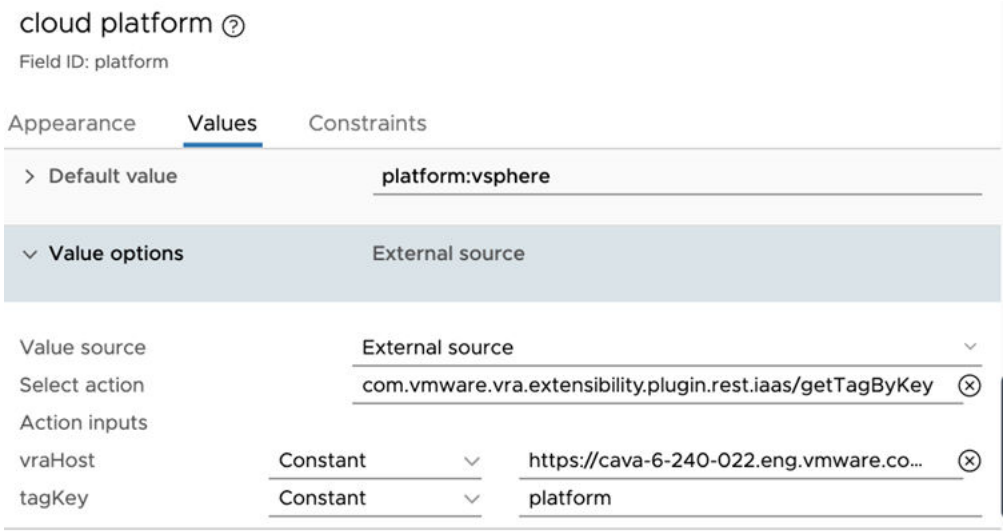
```
formatVersion: 1
inputs:
  platform:
    type: string
    title: cloud platform
  environment:
    type: string
    title: environment
resources:
  Cloud_vSphere_Machine_1:
    type: Cloud.Machine
    properties:
      image: centos
      flavor: small
      customizationSpec: Linux
      constraints:
        - tag: '${input.platform}'
        - tag: '${input.environment}'
      networks:
        - network: '${resource.Cloud_vSphere_Network_1.id}'
  Cloud_vSphere_Network_1:
```

```
    type: Cloud.Network
    properties:
      networkType: existing
```

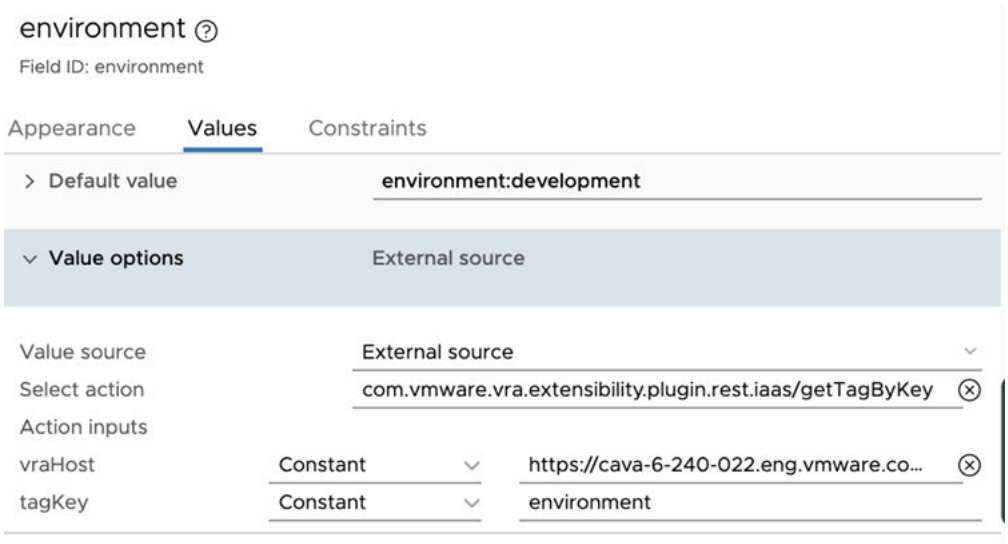
Associating an External Value with the getTagByKey Action

When you have versioned and released your cloud template, you must then import the content into Service Broker to allow you to customize the custom forms.

In custom forms, you associate the generic `getTagByKey` action as an external value for your drop-down menu fields. You supply the input tag key as a constant in the following example as a "platform":



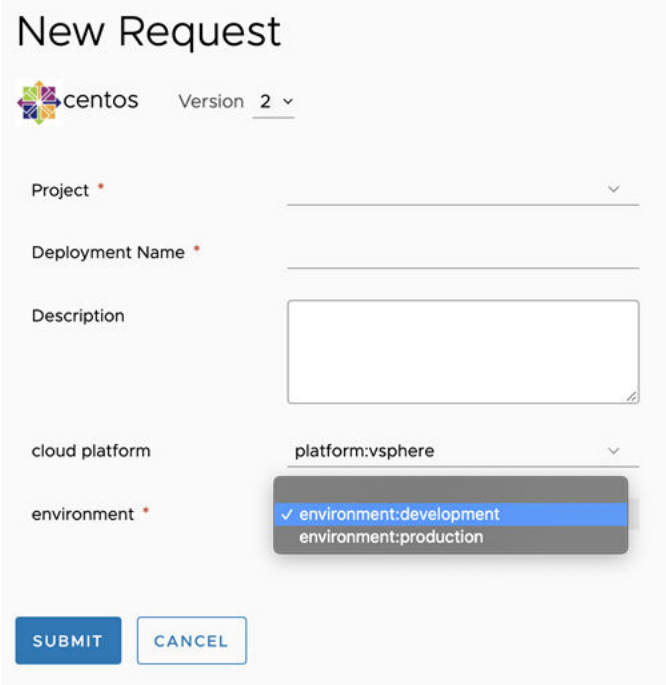
The same action is reused for the environment field.



Example Service Broker Catalog Request

After requesting your catalog item, you can now see your constraint tags to enable placement are now dynamically loaded.

The following screenshot shows an example Service Catalog request:



Requesting Catalog Items Programatically

While custom forms allow you to customize the requests, there are scenarios requiring you to use a programatic approach when handling requests. For example, when vRealize Orchestrator is used for external integrations requiring triggering requests or when automating requests.

In vRealize Automation 7.x, you can manage requests with the vRealize Automation plug-in or matching REST API:

- From a catalog item get the provisioning request data by using `getProvisioningRequestForCatalogItem()` and `getProvisioningRequestData()`. The provisioning request data is a type of template similar to the vRealize Automation 8.x cloud template YAML, but here formatted in JSON.
- Update the provisioning request data.
- Use `requestCatalogItemWithProvisioningRequest(catalogItem, provisioningRequest)`.

This provisioning request data is a complex object including many fields that are not necessarily matching what the end user would see at request time. For example to change the number of CPUs it is necessary to change `provisioningRequestData.ComponentName.data.cpu = cpuNb`. It is also mandatory to set some fields like the business group ID, which is the equivalent of a vRealize Automation 8.x project.

In vRealize Automation 8.x, requesting a catalog item programmatically is simpler. The request is done by using the Service Broker API `/catalog/api/items/{id}/request`. The body of the request includes:

- `deploymentName`
- `projectId`
- `requestCount`
- The request inputs defined in the YAML.

The requests returns an array of deployment IDs since some cloud templates support more than 1 request.

The following is an example of a request body:

```
{
  "deploymentName": "TestRequest",
  "projectId": "1628469a-3f98-44f1-ba80-e9ee610686a3",
  "bulkRequestCount": 1,
  "inputs": {
    "platform": "platform:vsphere",
    "environment": "environment:production"
  }
}
```

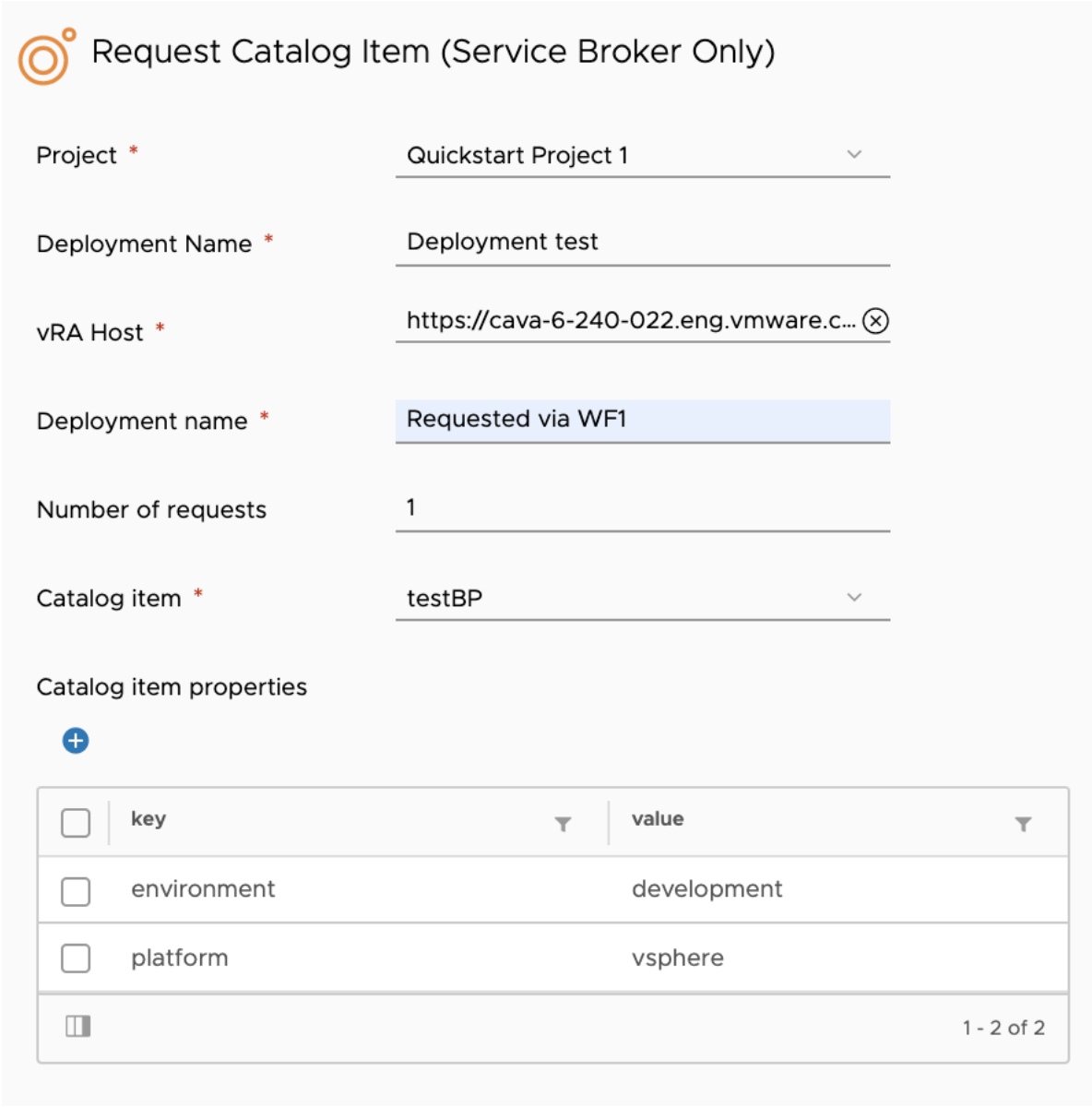
The input keys can be obtained with a GET `/catalog/api/items/" + catalogItemId` call. The sample action `getCatalogItemInputProperties` does that and outputs the list of inputs in a data grid.

The following example includes sample code from the `createCatalogItemRequest` action:

```
var url = "/catalog/api/items/" + catalogItemId + "/request";
var requestBody =
{
  "deploymentName": deploymentName,
  "projectId": projectId,
  "bulkRequestCount": bulkRequestCount,
  "inputs": inputProperties
}

var content = JSON.stringify(requestBody);
var operation = "POST";

try {
         var contentAsString =
System.getModule("com.vmware.vra.extensibility.plugin.rest").invokeRestOperation(vraHost,
```

```
operation, url, content);
    } catch (e) {
         throw "POST " + url + "Failed" +
    "\n Error : " + e;
}
var deployments = JSON.parse(contentAsString);
var deploymentsIds = new Array();

for each (var deployment in deployments) {
  deploymentsIds.push(deployment.deploymentId);
}
return deploymentsIds;
```

The sample workflow `Request Catalog Item (Service Broker Only)` lists the catalog items in a drop-down menu. The list of inputs is preconfigured so the value can be edited and when submitted, run the `createCatalogItemRequest` action.



Tags and Custom Properties

10

You can use tags and custom properties to further configure your vRealize Automation components and deployments.

In vRealize Automation 7.x, custom properties are responsible for:

- Providing information about the deployment.
- Modifying deployment configuration elements, such as VM hardware and OS configurations.
- Modifying configuration elements for vRealize Automation integrations.
- Attaching information to deployments for use in reporting and for additional payload properties to use in extensibility.
- Modifying the deployment placement.

Custom properties function as both custom key and value pairs, and also as reserved properties. For more information on reserved properties, see the *Custom Properties Reference* guide.

These custom properties can be set at different levels including endpoint, reservation, compute resource, business group, cloud template, and property group.

They can also be set at request time in the input forms, changed with Event Broker using the `virtualMachineAddOrUpdateCustomProperties` workflow output or using the `addUpdatePropertyFromVirtualMachineEntity` parameter.

vRealize Automation 8.x offers similar functionality with some changes:

- The properties are now part of the Cloud Assembly cloud template designer schema. They can also be set at deployment time through input form inputs.
- The names and meanings have changed and are documented in the vRealize Automation Resource Type Schema. The properties that impact the deployment on change are documented as `recreateOnUpdate: true`.
- Some extensibility features can also use predefined custom properties, such as the AD integration.

As an example of this custom properties functionality, you can use the scenario for setting the folder name in vCenter that machine will deploy to. In vRealize Automation 7.x, this can be done with the `VMware.VirtualCenter.Folder` property. This property specifies the name of the inventory folder in the data center in which to put the virtual machine. The default folder is `VRM`,

which is also the vSphere folder in which vRealize Automation places provisioned machines if the property is not used. This value can be a path with multiple folders, for example production or email servers. A proxy agent creates the specified folder in vSphere if the folder does not exist. Folder names are case-sensitive. This property is available for virtual provisioning

The equivalent property in vRealize Automation 8.x is `folderName`.

```
folderName          string
                    minLength: 1
                    recreateOnUpdate: true
                    title: VM folder for provisioning

                    The path to the folder where the virtual machine is provisioned, relative to the datacenter that the resource pool is in.
```

In vRealize Automation 8.x, Event Broker can modify properties with the `customProperties` workflow output on many events and dedicated outputs as described in the Event Broker section.

In vRealize Automation 7.x, tags have a minor function. There are custom use cases where a vRealize Orchestrator workflow or a PowerShell cmdlet can update a vCenter VM tag that can be used during or after the deployment.

In vRealize Automation 8.x, tags have a larger function.

- Capability tags define the placement logic during provisioning. They can be set on compute resources, cloud zones, images and image maps, networks, and network profiles.
- Constraint tags are set on cloud templates and projects so they can match the resources set with capability tags.
- Standard tags are used to filter, analyze, monitor, and group deployed resources.

Tags are included in different endpoints such as vSphere, Amazon Web Services (AWS), and Microsoft Azure, or created in vRealize Automation. Tags can be set at deployment time by Event Broker by using the `tags` workflow output parameter. vRealize Automation 8.x also allows you to update tags as day 2 operations on projects, deployment resources, and machines.

The following example can be used to update tags provided as a properties input on a deployment by using the deployment resource action `EditTags`. The sample is included in the `setDeploymentResourceTagsFromProperties` vRealize Orchestrator action that can be run as part of the `Edit deployment tags` workflow.

```
if (vraHost == null || deploymentId == null || resourceName == null) return null;

var operation = "POST";
var url = "/deployment/api/deployments/" + deploymentId + "/requests";

var object = {
  "actionId": "Deployment.EditTags",
  "targetId": deploymentId,
  "inputs": {}
}

object.inputs[resourceName] = new Array();
for each (var key in tags.keys) {
```

```
  var tag = {"key": key,"value": tags.get(key)};
  object.inputs[resourceName].push(tag);
}

var content = JSON.stringify(object);

try {
         var contentAsString =
System.getModule("com.vmware.vra.extensibility.plugin.rest").invokeRestOperation(vraHost,
operation, url, content);
         var object = JSON.parse(contentAsString);
    } catch (e) {
         throw("Unable to POST object url : " + url + "\n" + e + "\nWith Content : " +
content);
}
```

The following example can be used to update tags on a project by using the PATCH operation. The sample is included the in the `setProjectTagsFromProperties` vRealize Orchestrator action that can be run as part of the `Edit project tags` workflow.

```
if (vraHost == null || projectId == null) return null;

var operation = "PATCH";
var url = "/iaas/api/projects/" + projectId + "/resource-metadata";

var object = {"tags":[]};
for each (var key in tags.keys) {
  var tag = {"key": key,"value": tags.get(key)};
  object.tags.push(tag);
}

var content = JSON.stringify(object);

try {
         var contentAsString =
System.getModule("com.vmware.vra.extensibility.plugin.rest").invokeRestOperation(vraHost,
operation, url, content);
         var object = JSON.parse(contentAsString);
    } catch (e) {
         throw("Unable to Patch object url : " + url + "\n" + e + "\nWith Content : " +
content);
}
```

The following example can be used to update custom properties on a machine by using the PATCH operation. The sample is included in the `setMachineCustomPropertiesFromProperties` vRealize Orchestrator action that can be run as part of the `Edit machine custom properties` workflow.

```
if (vraHost == null || machineId == null) return null;

var url = "/iaas/api/machines/" + machineId;
var object = new Object();
```

```
var customPropertiesObject = {"customProperties" : customProperties};
var content = JSON.stringify(customPropertiesObject);
System.debug("Updated Custom properties : " + content);

var operation = "PATCH";

try {
         var contentAsString =
System.getModule("com.vmware.vra.extensibility.plugin.rest").invokeRestOperation(vraHost,
operation, url, content);
         var object = JSON.parse(contentAsString);
    } catch (e) {
         throw("Unable to Patch object url : " + url + "\n" + e + "\nWith Content : " +
content);
}
```

For more custom property examples, see Update the Custom Properties of a Machine.

Using vRealize Automation XaaS Services

11

vRealize Automation includes a XaaS capability that can be used to further automate your environment.

With Event Broker and other automation and integration scenarios, the vRealize Orchestrator workflows run in the back-end. There are also use cases that require the end users to trigger the workflows from the vRealize Automation user interface. This capability is called Anything as a Service (XaaS).

This chapter includes the following topics:

- Differences between vRealize Orchestrator Forms and Service Broker Forms
- Workflow Sample
- Using Custom Resources

Differences between vRealize Orchestrator Forms and Service Broker Forms

There are key differences between using vRealize Orchestrator and vRealize Automation Service Broker forms.

Starting with vRealize Automation 8.2, Service Broker is capable of displaying input forms designed in vRealize Orchestrator with the custom forms display engine. However, there are some differences in the forms display engines.

Amongst the differences, the following features supported in vRealize Orchestrator are not yet supported in Service Broker:

- The inputs presentations developed with the vRealize Orchestrator Legacy Client used in vRealize Orchestrator 7.6 and earlier, are not compatible. vRealize Orchestrator uses a built-in legacy input presentation conversion that is not available from Service Broker yet.
- The inputs presentation in vRealize Orchestrator has access to all the workflow elements in the workflow. The custom forms have access to the elements exposed to vRealize Automation Service Broker through the `VRO-Gateway` service, which is a subset of what is available on vRealize Orchestrator.
    - Custom forms can bind workflow inputs to action parameters used to set values in other inputs.

- Custom forms cannot bind workflows variables to action parameters used to set values in other inputs.

It is possible to work around vRealize Automation not having access to workflow variables by one of the following options :

- Using a custom action returning the variable content.
- Binding to an input parameter set to not visible instead of a variable.
- Enabling custom forms and using constants.

The widgets available in vRealize Orchestrator and in vRealize Automation vary for certain types. The following table describes what is supported.

	vRealize Automation			vRealize Orchestrator		
Input Data Type	Possible Form Display Types	Action return type for Default Value	Action return type for Value Options	Possible Form Display Types	Action return type for Default Value	Action return type for Value Options
String	■ Text, TextField, Text Area ■ Dropdown, Radio Group	String	■ Array of String ■ Properties ■ Array of Properties (value, label)	■ Text, TextFleld, Text Area ■ Dropdown, Radio Group	String	■ Array of String
Array of String	Array Input (vRA 8.2), Dual List, Multi Select	■ Array of String ■ Properties ■ Array of Properties	Array of String	Datagrid, Multi Value Picker	■ Array of String ■ Properties ■ Array of Properties	Array of String
Integer	Integer	Number	Array of Number	Not supported	Not supported	Not supported
Array of Integer	Array Input (vRA 8.2), Datagrid (vRA 8.1)	Array of Number	Array of Number	Not supported	Not supported	Not supported
Number	Decimal	Number	Array of Number	Decimal	Number	Array of Number
Array/ Number	Array Input (vRA 8.2), Datagrid (vRA 8.1)	Array of Number	Array of Number	Datagrid	Array of Number	Array of Number
Boolean	Checkbox	Boolean	Not supported	Checkbox	Boolean	-
Date	Date Time	Date	Array of Date	Date Time	Date	Array of Date

Array of Date	Array Input (vRA 8.2), Datagrid (vRA 8.1)	Array of Date	Array of Date	Datagrid	Array of Date	Array of Date
Composite/ Complex	Datagrid, Object Field (vRA 8.3)	Composite, Properties, Array/ Composite, Array/ Properties	Array of Composite	Datagrid	■ Composite(columns...) ■ Array/Properties	Array of Composite
Array of Composite	Datagrid, Multi Value Picker	Composite, Properties, Array/ Composite, Array/ Properties	Array of Composite	Datagrid, Multi Value Picker	■ Array/ Composite(columns...) ■ Array/Properties	Array of Composite
Reference / vRO SDK Object type	Value Picker	SDK Object	Array of SDK Object (vRA 8.2)	Value Picker	SDK Object	Array of SDK Object
Array of Reference	Multi Value Picker (vRA 8.3)	Array of SDK Object	Array of SDK Object (vRA 8.3)	Datagrid	Array of SDK Object	Array of SDK Object
Secure String	Password	String	Not supported	Password	String	Not supported
File	Not supported	Not supported	Not supported	File Upload	Not supported	Not supported

For use cases where the widget specified in vRealize Orchestrator is not available from Service Broker, a compatible widget is used.

Because the data being passed to and from the widget might expect different types, formats, and values in the case they are unset, the best practice to develop workflows targeting Service Broker is to:

1 Develop the vRealize Orchestrator workflow. This can include both the initial development of the workflow or changes of inputs.

2 Version the workflow manually.

3 In Cloud Assembly, navigate to **Infrastructure > Connections > Integrations** and select your vRealize Orchestrator integration.

4 Start the data collection for the vRealize Orchestrator integration. This step, along with versioning up your workflow, ensure that the `VRO-Gateway` service used by vRealize Automation has the latest version of the workflow.

5 Import content into Service Broker. This step generates a new default custom form.

6 In addition to the input forms designed in vRealize Orchestrator, you can, if needed, develop workflow input forms with the custom forms editor.

7 If these forms call actions, develop or run these from the vRealize Orchestrator workflow editor.

8 Test the inputs presentation in Service Broker.

9 Repeat from step 5 as many times as needed.

10 Repeat from step 1, in case workflows inputs or forms need to be changed.

Either distribute and maintain the custom forms or alternatively, design vRealize Orchestrator inputs by using the same options or actions as in the custom forms (the above step 1), and then repeat the steps 2 to 8 to validate that the process works.

Using this last option means that:

- Running the workflow from vRealize Orchestrator can lead to the input presentation not working as expected when started in vRealize Orchestrator.
- For some cases, you must modify the return type of the actions used for default value or value options so these values can be set from the vRealize Orchestrator workflow editor and, when the workflow is saved, revert the action return types.

Designing the form in the workflow has the following advantages:

- Form is packaged and delivered as part of the workflow included in a package.
- Form can be tested in vRealize Orchestrator as long as the compatible widgets are applied.
- The form can optionally be versioned and synchronized to a Git repository with the workflow.

Designing the custom forms separately has the following advantages:

- Being able to customize the form without changing the workflow.
- Being able to import and export the form as a file and reusing it for different workflows.

For example, a common use case is to have a string based drop-down menu.

Returning a Properties type can be used in both the vRealize Orchestrator input form presentation and vRealize Automation custom forms presentation. With the Property type you can display a list of values in the drop-down menu. After being select by the user, these values pass an ID to the parameter (to the workflow and the other input fields that would bind to this parameter). This is very practical to list objects when there is no dedicated plug-in for them as this avoids you having to select object names and having to find object IDs by name.

Returning an array of Properties types has the same goal as returning Properties but does give control on the ordering of the element. It is done by setting for each property in the array the `label` and `value` keys. For example, it is possible to sort ascending or descending properties by label or by keys within the action.

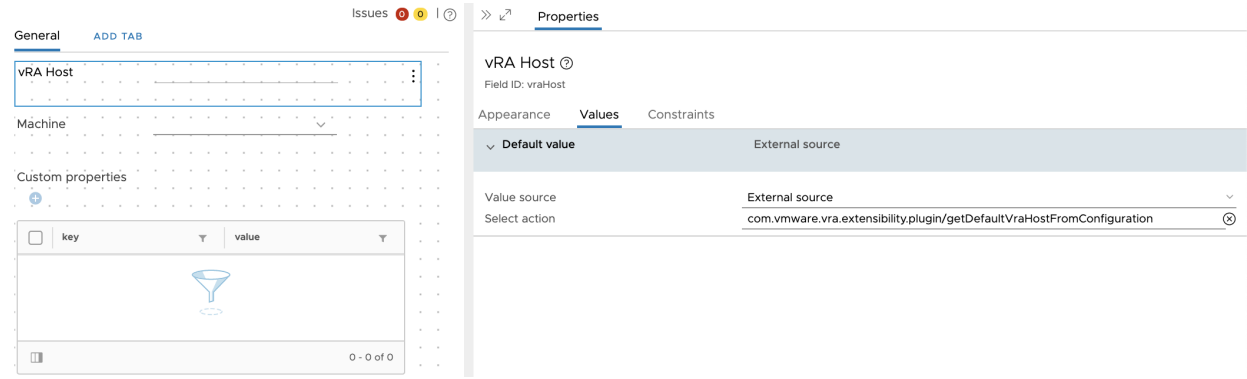
All the workflows included in the "`drop down`" folder of the sample package include drop down menus created with actions that have array of Properties set as the return type.

Workflow Sample

The following workflow sample demonstrates how you can edit the custom properties of a virtual machine.

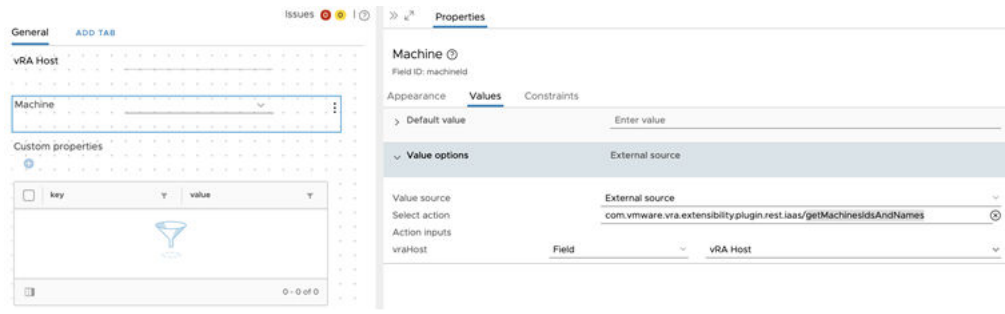
This workflow sample works in both vRealize Orchestrator and vRealize Automation Service Broker. The following form implementation created in the vRealize Orchestrator input form designer is mandatory.

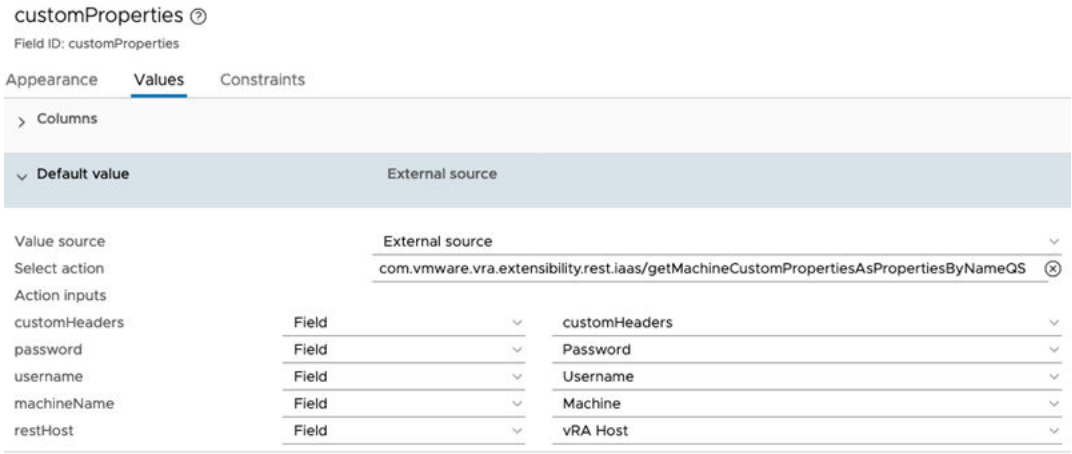
For the default value to work in Service Broker, the `vRA Host` value is set by an action returning the content of the variables of the configuration element.



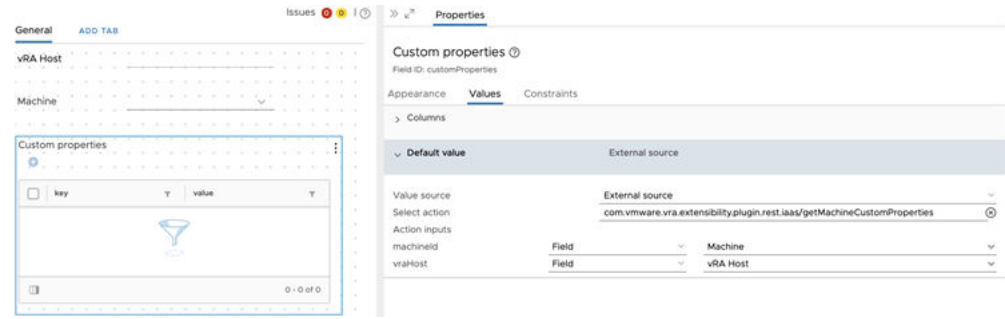
If the default value was set in vRealize Orchestrator with a field variable binding, you would have to enable custom forms and set the constants after importing the workflow.

The `machineName` input labeled as `Machine` is defined as a drop-down menu that uses `getMachineIdsAndNames` as value options. This action returns an array of properties with the machine ID as a label and the machine name as a value.





The `customProperties` data grid uses a `getMachineCustomProperties` action that returns a Properties type that can be bound directly to the `machineid` input.



Using Custom Resources

One of the primary features of using XaaS in vRealize Automation is using custom resources.

In vRealize Automation 7.x, it is possible to create an XaaS service blueprint. The service blueprint can be used to define and use a vRealize Orchestrator workflow from vRealize Automation. The service blueprint can be published in the catalog service and entitled, and it can be used in the blueprint design canvas. In both cases, it can provision a custom resource as an option. This resource can:

- Appear in the **Items** tab (for versions earlier than vRealize Automation 7.6) or **Deployments** tab (for vRealize Automation 7.6) when request from the catalog.

- Appear as one of the components of the deployment when requested as part of a composite blueprint.

Provisioning a custom resource allows you to track the custom resource and its realtime properties from the user interface. It also enables you to perform day 2 operations known as resource actions.

In vRealize Automation 7.x resource actions can run a workflow in context of a custom resource for:

- Delete operations (Using a disposal option)
- Update operations (No option)
- Copy operations (Using a provisioning option)
- Move or stage operations (Using a provisioning & disposal option)

In vRealize Automation 8.x, custom resources offer similar capabilities in comparison to vRealize Automation 7.x. A custom resource in vRealize Automation 8.x has the following mandatory requirements:

- You must have a provisioning workflow that must output an vRealize Orchestrator plug-in type that matches the type defined in the custom resource
- You must have a decommission workflow.

vRealize Automation 8.x custom resources allow you to use a specific vRealize Orchestrator type once per project or having it shared with all projects once. It is not possible, for example, to use different provisioning workflows outputting the same custom resource type.

vRealize Automation 8.x resource actions have the following differences with vRealize Automation 7.x:

- Provisioning and decommissioning options.
- No capability to bind custom resources to vRealize Orchestrator action parameters in the request form.

A core difference is that the availability of the resource action in vRealize Automation 8.x can be defined programmatically based on the resource properties. For example some actions might not be available if the state of the element is `"OFF"`. The equivalent feature in vRealize Automation 7.x has less options because it is user interface based.

Resource Mappings

A resource mapping defines how a native vRealize Automation resource is converted to a vRealize Orchestrator type.

In vRealize Automation 8.x, there are more resource types being supported than in comparison to vRealize Automation 7.x. Each of these resources has a schema defining the resource properties. You can define a vRealize Orchestrator action that binds its inputs to these properties and return the equivalent vRealize Orchestrator object. For example, the vSphere components have a `vCenterUuid` and `uuid` property that can be used to return a vRealize Orchestrator type, such as `VC:VirtualMachine`. Another good example of this functionality is the built-in `findVcVMByVcAndVMUuid` action introduced in vRealize Automation 8.x.

When the resource mapping is created, it is possible to add new day 2 operations on these resources that are workflows that use the matching vRealize Orchestrator type as inputs.

The main difference in comparison to vRealize Automation 7.x, is that it uses a workflow for resource mapping. It might be necessary to migrate the workflows to actions to reuse their functionality in vRealize Automation 8.x.

Another difference is that vRealize Automation 8.x is that the resource mapping can be defined on a resource action basis. For each input of the resource action, it is possible to either expose the input at request time, map it to one of the schema resource properties, or associate a mapping action that uses one or more schema resource properties and returns the same type as the workflow input it binds to. This is useful to pass information from the schema directly without having to change the workflow or create a wrapper workflow that must be used to add the scripting logic to these required to query the vRealize Automation resource from vRealize Orchestrator.

Custom Cloud Template Component

The application of custom components is a key element of cloud template development.

An important limitation of the XaaS components used in the cloud template designer is that, being custom resources-based, it must provision a custom component. This is different from vRealize Automation 7.x where the service can start any workflow, even if it was not outputting a custom resource.

If your environment includes vRealize Automation 7.x blueprints components that are, for example, implementing some configuration changes, it will be necessary to use other means to trigger this workflow. The alternative is to use Event Broker to do so.

Another area that is very different in vRealize Automation 8.x is the way that these components inputs and outputs can be bound to other components on the schema. In vRealize Automation 7.x, these bindings only supported simple types and were controlled through the user interface without any program based approach. In vRealize Automation 8.x, the Create workflow inputs define the properties in the YAML schema and these can be scripted. These inputs can be mapped to the cloud template input properties even if these are of complex types. With this you can, for example, use an input of a given resource type that can be searched.