

---

# REST API v2

## Getting Started Guide



## Contents

|   |    |
|---|----|
| Introduction .....  | 5  |
| Version compatibility .....   | 5  |
| Changes in this release .....   | 5  |
| REST API capabilities .....   | 5  |
| Cloud automation .....  | 5  |
| System configuration and status .....                                 | 6  |
| VM and infrastructure inventory management.....                       | 6  |
| Costing .....   | 6  |
| User management .....   | 6  |
| Per-tenant API.....   | 7  |
| Where to find reference information .....                             | 7  |
| How does the REST API work? .....                                     | 7  |
| Security .....  | 8  |
| Exception and error handling.....                                     | 8  |
| Data model.....   | 8  |
| Installation .....  | 9  |
| Where to install the PowerShell client .....                          | 9  |
| 1. Download the REST PowerShell client .....                          | 9  |
| 2. Optional: Download the PowerShell Logging module .....             | 9  |
| 3. Download the Microsoft .NET Framework version 4.5 or higher .....  | 9  |
| 4. Specify that the downloaded .zip files are "trusted" content ..... | 9  |
| 5. Install the REST PowerShell Module .....                           | 10 |
| 6. Install the PowerShellLogging module.....                          | 11 |
| 7. Verify the installation.....                                       | 11 |
| 8. Set permissions on folders.....                                    | 11 |
| Create a user account for the REST API .....                          | 11 |
| If your use of the API affects service request automation .....       | 12 |
| Using the REST API .....  | 12 |
| Base Service URL .....  | 12 |
| Important: Log out after every session.....                           | 13 |
| Using the REST API PowerShell client .....                            | 13 |
| Other custom clients.....   | 18 |
| Using the REST API with a Service Portal account .....                | 18 |

|  |    |
|--|----|
| Restricting REST API access to a specific host name or IP address.....                       | 19 |
| Using a search query to restrict the data returned .....                                     | 19 |
| Examples .....   | 20 |
| Constructing the query .....   | 20 |
| Examples .....   | 22 |
| Example 1: Connect your automated build machine to the vCommander Service Catalog .....      | 22 |
| Example 2: Initiate a service request from an external system.....                           | 23 |
| Example 3: Approve and deploy a service request from an external system .....                | 24 |
| Example 4: Add a datastore to automated deployment destinations .....                        | 26 |
| Example 5: Create an organization and assign cost quota .....                                | 26 |
| Example 6: Create an organization and assign resource quota.....                             | 27 |
| Example 7: Set a VM's compliance state .....   | 28 |
| Example 8: Set VM ownership .....  | 28 |
| Example 9: Create an IP pool without specifying networks .....                               | 29 |
| Example 10: Create an IP pool and specify networks.....                                      | 29 |
| Example 11: Create a deployment destination for vCenter .....                                | 30 |
| Example 12: Create a deployment destination for SCVMM .....                                  | 35 |
| Example 13: Create a deployment destination for Amazon EC2.....                              | 37 |
| Example 14: Create a Compliance Policy .....   | 38 |
| Example 15: Create a Default Attributes Policy .....   | 39 |
| Example 16: Create an Expired VM Policy .....  | 40 |
| Example 17: Create a Default Ownership Policy.....   | 42 |
| Example 18: Create a cost model .....  | 43 |
| Example 19: Create an approval workflow .....  | 44 |
| Example 20: Create a completion workflow .....   | 45 |
| Example 21: Create a command workflow .....  | 46 |
| Example 22: Connect a media file to a VM's CD-ROM drive .....                                | 47 |
| Example 23: Disconnect a media file from a VM's CD-ROM drive .....                           | 47 |
| Example 24: Create a global media folder.....  | 47 |
| Example 25: Create an organization-specific media folder .....                               | 48 |
| Example 26: Create relationships between custom attributes .....                             | 48 |
| Example 27: Update a custom attribute that has a relationship with a sublist attribute ..... | 49 |
| Example 28: Add a VM to the service catalog and configure the service.....                   | 50 |
| Example 29: Request a service.....   | 50 |
| Example 30: Retrieve available quota for an organization or organization member.....         | 51 |

|  |    |
|--|----|
| Example 31: Run a command workflow .....   | 51 |
| Example 32: Share a VM .....   | 51 |
| Example 33: Create a completion workflow for shared VMs .....                                      | 52 |
| Examples for limiting results using query syntax.....  | 52 |
| Troubleshooting.....   | 53 |
| “Client requires security protocol TLSv1.2 or TLSv1.1 to communicate with server” error.....       | 54 |
| “Could not load file or assembly” error .....  | 54 |
| “Module file not found ” error when attempting to run Import-Module .....                          | 55 |
| “Untrusted certificate” error .....  | 56 |
| “Content is not allowed in prolog ” error when attempting to call an API .....                     | 56 |
| Error: JAXBException occurred : The reference to entity "{0}" must end with the ';' delimiter..... | 58 |
| Appendix: PowerShell script credential encryption.....   | 59 |
| To encrypt your credentials .....  | 59 |
| To decrypt your credentials .....  | 59 |
| The encryption process.....  | 60 |
| Appendix: vCommander REST API security and authentication reference .....                          | 61 |
| Client authentication .....  | 61 |
| Session overview.....  | 63 |

## Introduction

This guide provides information on installing and using the Embotics® vCommander® REST API v2.

### Version compatibility

| REST API version | Compatible vCommander versions                              |
|------------------|---|
| 2.9              | 7.0 and higher  |
| 2.8.1            | 6.1 and higher  |
| 2.8              | 6.0 and higher  |
| 2.7.4            | 5.7.8 and higher  |
| 2.7.0 – 2.7.3    | 5.7.7 and below   |
| 2.6              | 5.6 and higher  |
| 2.5              | 5.6 and higher  |
| 2.4              | 5.5 and higher  |
| 2.3              | 5.2 and higher  |
| 2.2              | 5.1.4 and higher  |
| 2.1              | 5.1.3 and higher  |
| 2.0              | 5.1.2, 5.1.1 and 5.1  |
| 1.0              | 5.1.2, 5.1.1, 5.1 and 5.0.x<br>(Version 2.x is recommended) |

Note: As the focus has shifted to the REST technology, the previously available SOAP API has been removed.

### Changes in this release

See the file Changelog.txt for a list of added, changed and deprecated functionality in this version.

## REST API capabilities

The vCommander REST API provides a comprehensive set of features, detailed below. These capabilities are highly beneficial to dev/test lab automation, multi-tenant customer on-boarding, and synchronization of asset inventory with third-party systems, to name a few typical examples.

### Cloud automation

- Publish and edit services in the Service Catalog
- Manage placement attributes, including assigning them to deployment destinations and published services
- Import and export request forms
- Submit, update and fulfill new service requests and change requests
- Retrieve the state of a service request
- Manage workflow definitions
- Import and export workflow definitions

- Configure automated deployment destinations
- Manage policies, such as Expiry (decommissioning) and Default Ownership
- Share VMs by automatically creating a Service Catalog entry and sending a request hyperlink to recipients via email
- Retrieve and assign groups (expiry, guest OS scan, maintenance, power schedule and rightsizing)
- Manage quota reservation
- Manage storage reservation

## System configuration and status

- Add or update custom attributes
- Retrieve VM Access Proxy settings
- Configure IP pools
- Manage and apply network zones
- Query task status
- Activate a standby vCommander (for more information on high availability configuration, contact [support@embotics.com](mailto:support@embotics.com))

## VM and infrastructure inventory management

- Add and update managed systems
- Synchronize the inventory of a managed system
- Manage VMs, virtual services and public cloud instances
- Set VM metadata (such as ownership, expiry date and custom attribute values)
- Retrieve items such as hosts, clusters and datastores
- Manage media folders
- Connect/disconnect media files to a VM's virtual hardware
- Manage AWS Stacks (includes retrieval, deletion, and metadata assignment)

## Costing

- Manage cost models
- Retrieve billing records

## User management

- Manage local user accounts
- Retrieve roles
- Manage organizations
- Manage quota

## Per-tenant API

A REST API user with a Service Portal role can:

- retrieve services assigned to an organization
- retrieve service requests
- request a service
- submit a change request for a VM
- manage VM snapshots
- retrieve quota information for an organization and its members (including quota usage)
- share a VM
- execute a command workflow on a VM

## Where to find reference information

Reference information on vCommander REST API functions, managed objects and data transfer objects is available in the online help of any running instance of vCommander (5.2 and higher). The reference page lists operations and their arguments, as well as input and output parameters. The API reference can be accessed through any browser using the following URL:

[https://<vCommander\\_host\\_name\\_or\\_IP>/apihelp](https://<vCommander_host_name_or_IP>/apihelp)

## How does the REST API work?

The API consumes and produces XML documents via standard HTTP methods (GET, PUT, POST, DELETE). While you can use any REST client to communicate with the vCommander REST service, Embotics provides a PowerShell client. This client converts the XML input/output parameters into PowerShell objects.

Note that the XML consumed by vCommander is case sensitive.

The following example of an XML input/output parameter instructs vCommander to deploy a service:

```
<DeploymentParams>
  <createLinkedClone>False</createLinkedClone>
  <highAvailability>False</highAvailability>
  <datastoreName>devTesting</datastoreName>
  <targetHostName/>
  <resourcePoolName>VMDeploymentPool</resourcePoolName>
  <folderName>VMDeploymentFolder</folderName>
  <snapshotId/>
  <clusterName>Engineering Cluster</clusterName>
  <name/>
  <managementServerName>manta</managementServerName>
  <approved>True</approved>
  <suspect>True</suspect>
  <endOfLife>True</endOfLife>
  <memoryInMB>2048</memoryInMB>
  <numCPU>2</numCPU>
</DeploymentParams>
```

## Security

vCommander REST API calls are privileged in the sense that the logged-in user must be authenticated before access is granted. The REST API supports HTTP basic authentication as well as token-based authentication. With token-based authentication, a security token is returned to the client after initial login. This security token must be delivered in the header for every subsequent HTTP request.

The vCommander REST API supports the TLSv1.1 and TLSv1.2 security protocols.

## Exception and error handling

vCommander uses standard HTTP status codes to indicate the status of API calls:

200: The call was successful; everything is OK. A payload may also be returned.

400: An error occurred between the caller and the REST service. For example, invalid URL or bad parameters.

500: An error occurred while vCommander was processing the request. For example, “Unable to deploy to this location because <reason>.”

For 400 and 500 responses, the returned XML structure is as shown:

```
<APICallErrorResult statusCode="500" >
  <Error errorCode="{1}" moreInfo="{3}">{2}</Error>
</APICallErrorResult >
```

{1} The error code; this will not change across releases. Use this code to look up additional details on this error.

{2} The error message; this can change between releases. A short summary of the error.

{3} A link to get additional information.

Note: You can check the vCommander log for more error details.

## Data model

The vCommander inventory includes managed objects. Each managed object has an ID, a name, a type and additional type-specific properties. Managed objects may also have relationships to other objects (for example, a host has relationships to VMs).

The relationships between objects are represented by object references — lightweight versions of the full managed object, containing only the ID, name and type. As object references are lightweight representations of the full managed objects, you can use them when the performance impact of returning full managed objects is significant (for example, when querying for thousands of VMs).

vCommander provides several APIs to retrieve specific managed object types. There are also generic APIs that can be used to retrieve any managed object with query parameters. In PowerShell, for example, the generic cmdlets are:

### Get-ManagedObject



## **Get-ManagedObjectById**

## **Get-ManagedObjectByName**

The vCommander REST API consumes and produces managed objects, managed object references and other objects; these are known as data transfer objects, or DTOs. The vCommander PowerShell module provides helper cmdlets to create new DTOs from a template.

# **Installation**

## **Where to install the PowerShell client**

The vCommander REST API PowerShell client is intended to be installed and run on a client machine. Typically, you will install the client on a machine other than the vCommander host server. However, if you will run the REST API through vCommander workflow scripts, then you need to install the client on the vCommander host.

Note: vCommander supports both PowerShell 3 and 4.

## **1. Download the REST PowerShell client**

To download the REST client, go to the [Embotics Knowledge Base Download REST Client Page](#).

**Note:** The PowerShell client requires Windows PowerShell 3.0 and higher. You can download PowerShell from the [Microsoft download site](#).

## **2. Optional: Download the PowerShell Logging module**

The vCommander PowerShell REST client includes support for Microsoft's [Enhanced Script Logging module v1.1.0.1](#). When installed, this module provides additional logging that you can use to troubleshoot execution of scripts.

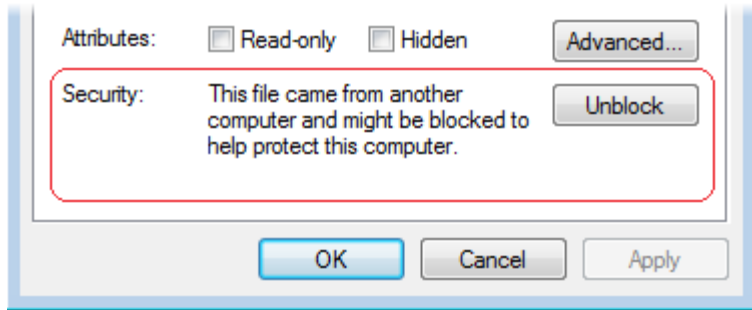
## **3. Download the Microsoft .NET Framework version 4.5 or higher**

The vCommander REST API supports the TLSv1.1 and TLSv1.2 security protocols. The Microsoft .NET Framework version 4.5 and higher includes TLSv1.1 and TLSv1.2. You can download it from the [Microsoft download site](#).

See also ["Client requires security protocol TLSv1.2 or TLSv1.1 to communicate with server" error](#).

## **4. Specify that the downloaded .zip files are "trusted" content**

Windows may block the use of files from external sources. To check the permissions on the downloaded .zip file, right-click it and select **Properties**. On the General tab of the Properties dialog, the following message indicates that the file is blocked:



To unblock the file, click **Unblock**.

**Note:** If you do not have full permissions on the folder where you saved the .zip file, the file will remain blocked. If this happens, move the .zip file to a location where you have full permissions (for example, your desktop), and unblock the .zip file there.

## 5. Install the REST PowerShell Module

Extract the zip files into the PowerShell module directory of the machine where you will run the PowerShell client.

The PowerShell module directory is specified by the PSModulePath environment variable.

PowerShell will look in the path(s) specified for the PSModulePath environment variable when searching for available modules on a system. By default, this variable points to:

```
%SystemRoot%\System32\WindowsPowerShell\v1.0\Modules\
```

For example:

```
C:\Windows\System32\WindowsPowerShell\v1.0\Modules\
```

### Installing on 64-bit Windows

On 64-bit Windows, there are two possible installation paths.

If you are using the **32-bit version** of PowerShell in vCommander workflow script steps, install the modules in:

```
%SystemRoot%\SysWOW64\WindowsPowerShell\v1.0\Modules\
```

If you are using the **64-bit version** of PowerShell in vCommander workflow script steps, install the modules in:

```
%SystemRoot%\system32\WindowsPowerShell\v1.0\Modules\
```

### Installing on 32-bit Windows

On 32-bit Windows, there is only one installation path:

```
%SystemRoot%\system32\WindowsPowerShell\v1.0\Modules\
```

## 6. Install the PowerShellLogging module

To enable file logging, extract the PowerShellLogging module at the same level of the PowerShell module directory as the VCommanderRESTClient.

On startup, the VCommanderRESTClient module will detect and import the PowerShellLogging module automatically.

To use logging, add the following to the top of the script:

```
$logFile = Enable-LogFile -Path $env:temp\test.log
```

This will cause log statements to be written to:

```
$env:temp\test.log
```

To close the logger at the end of script execution, add the following to the end of the script:

```
$logFile | Disable-LogFile
```

## 7. Verify the installation

To verify the installation, open PowerShell and enter the following command:

```
Get-Module -ListAvailable
```

You should see VCommander and VCommanderRestClient in the list of modules, as well as PowerShellLogging.

You should see the following subdirectories and files in the PowerShell Modules directory:

```
VCommander\VCommander.dll
VCommander\Security.psml
VCommander\VCommander.psd1
VCommander\VCommander.psml
VCommanderRestClient\VCommanderRestClient.psd1
VCommanderRestClient\VCommanderRestClient.psml
```

## 8. Set permissions on folders

If you installed the PowerShell client on the vCommander host, you need to grant Read and Execute permissions for the VCommander and VCommanderRestClient folders. If you're using a Microsoft SQL Server database, assign these permissions to the group in which the vCommander service account is a member. If you're using the default Postgres database, assign these permissions directly to the vCommander service account.

## Create a user account for the REST API

We recommend creating a dedicated local user account to access the REST service. This account must have either:

- a vCommander administrative role (either Enterprise Admin or Superuser) and appropriate access rights on the target managed systems, or

- a Service Portal role with organization membership. We recommend creating a custom Service Portal role for this purpose, with full permissions.

An account with a Service Portal role allows you to use vCommander in an organizational context. This means that you can take advantage of quotas and organization-based visibility for VMs, services, workflows and request forms. However, with a Service Portal role, the REST account can perform only the following REST API actions:

- log in and log out
- retrieve services assigned to the organization
- view and deploy service catalog entries
- view service requests
- submit a change request for a VM
- manage VM snapshots
- retrieve quota information for an organization and its members (including quota usage)
- share a VM
- execute a command workflow on a VM

See also [Using the REST API with a Service Portal account](#) below.

Because authentication is performed by vCommander, the account used to access the REST service does not need rights to execute PowerShell commands on the vCommander host.

## If your use of the API affects service request automation

If you will use the API to run commands related to the service request process, we recommend configuring dedicated workflows, request forms, service catalog entries and deployment destinations, and assigning them to the API user account.

## Using the REST API

### Base Service URL

#### vCommander version 5.1 and later

The base service URL for the version 5.1 and later REST service is:

<https://<vCommander host or IP>/webservices/services/rest/v2>

All requests must start with this address. For example, to retrieve a list of all VMs (up to 1000), the REST service path is:

<https://<vCommander host or IP>/webservices/services/rest/v2/vms?max=1000>

## vCommander version 5.0 and earlier

The base service URL for the version 5.0.x and earlier REST service is:

[https://<vCommander\\_host\\_or\\_IP>/webservices/services/VCommanderRestV10](https://<vCommander_host_or_IP>/webservices/services/VCommanderRestV10)

All requests must start with this address. For example, to retrieve a list of all VMs (up to 1000), the REST service path is:

[https://<vCommander\\_host\\_or\\_IP>/webservices/services/VCommanderRestV10/vms?max=1000](https://<vCommander_host_or_IP>/webservices/services/VCommanderRestV10/vms?max=1000)

## Important: Log out after every session

You must log out after every REST API session.

## Using the REST API PowerShell client

**Note:** You must escape any string that has special meaning in HTML (for example, the & symbol must be expressed as &amp;).

Check the [Troubleshooting](#) of this guide if you encounter errors.

To encrypt your credentials, see the [PowerShell Script Credential Encryption](#) section.

To see the list of commands provided by this module, enter the following command into the console:

```
Get-Command -Module VCommanderRestClient
```

Find the usage of a command using this command:

```
Get-Help <command> -Examples
```

For example:

```
Get-Help Get-PublishedServiceReferences -Examples
```

This will output to the screen the usage example(s) for this command.

**Note:** All API outputs are PowerShell objects.

## Connecting to the REST API with the PowerShell client

To connect using PowerShell, you must:

1. Import the VCommander and VCommanderRestClient modules
2. Configure variables.

**Note:** In the following example, credentials are included in plain text for simplicity; this is not recommended for production usage. Instead, you should encrypt your credentials as shown in [Appendix: PowerShell script credential encryption](#) below.

3. Log in
4. Important: Log out when finished

For example:

```
# Import the vCommander modules
Import-Module -Name VCommander
Import-Module -Name VCommanderRestClient

# Credentials
$secpasswd = ConvertTo-SecureString "secret" -AsPlainText -Force
$mycreds = New-Object System.Management.Automation.PSCredential ("superuser",
$secpasswd)

# Log in
Connect-Client2 -url "https://host.example.com/webservices/services/rest/v2" -
credentials $mycreds -ssltrustall

# Logout
Disconnect-Client
```

See also [Using the REST API with a Service Portal account](#) below.

## Constructing the data transfer object (DTO)

The vCommander PowerShell module provides a cmdlet to create a DTO from a template. This ensures that the object is created correctly and contains the required properties. The PowerShell cmdlet is:

```
New-DTOTemplateObject -DTOTagName "<dto tag name>"
```

For example:

```
New-DTOTemplateObject -DTOTagName "Account"
```

This creates a template Account DTO. Inspecting the properties of this DTO shows the following:

```
PS H:\> $accountTemplate = New-DTOTemplateObject -DTOTagName "Account"
Creating template DTO object for Account

PS H:\> $accountTemplate

Account
-----
System.Object

PS H:\> $accountTemplate.Account

id                : 0
emailAddress      : bob@embotics.com
enabled          : true
firstName         :
lastName         :
password         :
primaryPhoneNo    :
role             : System.Object
secondaryPhoneNo  :
securitySourceType : USER_DIRECTORY_USER
userid           : bob@embotics.com
class            : wsAccount
```

You can modify this DTO's properties as required before sending the DTO to the REST service:

```
PS H:\> $accountTemplate.Account.userid = "jane.smith@embotics.com"

PS H:\> $updatedAccount = New-Account -accountDto $accountTemplate.Account|
```

Most DTOs in vCommander can be created this way. The list of DTOs available is shown in the online API help under the section "Data Elements":

[https://<vCommander\\_host\\_name\\_or\\_IP>/apihelp/model.html](https://<vCommander_host_name_or_IP>/apihelp/model.html)

The DTO created from a template may contain properties that are not relevant for your use. In this case, you should clear the properties before submitting the DTO to vCommander.

For example:

```
PS H:\> $organizationTemplate = New-DTOTemplateObject -DTOTagName "Organization"
Creating template DTO object for Organization

PS H:\> $organizationTemplate

Organization
-----
System.Object

PS H:\> $organizationTemplate.Organization

id           : -1
name         : Finance Department
resourceQuota : System.Object
Members      : System.Object
class        : wsOrganization
```

As shown above, the template Organization DTO contains a "resourceQuota" property. This property is not applicable if you want to configure a cost quota, so you need to remove this property:

```
$organizationTemplate.Organization.PSObject.Properties.Remove("resourceQuota")
```

Then create a cost quota DTO:

```
$costQuotaTemplate = New-DTOTemplateObject -DTOTagName "CostQuota"
$costQuotaTemplate.CostQuota.dailyCost = 10000
```

Add this cost quota DTO to the Organization DTO:

```
Add-Member -InputObject $organizationTemplate.Organization -MemberType NoteProperty -
Name "costQuota" -Value $costQuotaTemplate.CostQuota -Force
```

**Note:** To learn the name of the property to use, consult the "Data Types" section of the API online help:

[https://<vCommander\\_host\\_name\\_or\\_IP>/apihelp/model.html](https://<vCommander_host_name_or_IP>/apihelp/model.html)

Looking at the `$organizationTemplate` variable again, you will see that the `resourceQuota` property has been removed and a new property `costQuota` has been added:

```
PS H:\> $organizationTemplate.Organization

id       : -1
name     : Finance Department
Members  : System.Object
class    : wsOrganization
costQuota : System.Object

PS H:\> $organizationTemplate.Organization.costQuota | fl

dailyCost : 10000
```

**Note:** Instead of using `New-DTOTemplateObject` API to create your DTOs, you can also create the DTOs using standard PowerShell functions, but instructions for this are not included in this guide.

## Navigating API models

All vCommander models are represented as properties of `System.Object` in PowerShell, so you can use PowerShell ISE's dot notation (.) to walk through the properties of the object.

For example:

```
PS H:\> $managementServers = Get-ManagementServers

PS H:\> $managementServers

ManagementServerCollection
-----
System.Object
```

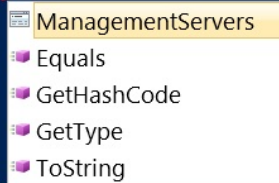


The above function retrieves a list of managed systems from a running vCommander instance. The PowerShell variable `$managementServers` contains an object of type `System.Object` called `ManagementServerCollection`. Using dot notation, you can navigate its properties as shown below:

```
PS H:\> $managementServers.ManagementServerCollection

ManagementServers
-----
{System.Object, System.Object, System.Object, System.Object...}

PS H:\> $managementServers.ManagementServerCollection.
```



Properties can also be filtered as shown below:

```
PS H:\> $ms = $managementServers.ManagementServerCollection.ManagementServers | Where-Object {$_.name -match "devvc"}
PS H:\> $ms

displayName      : devvc
id               : 163840
type             : MANAGEMENTSERVER
connected        : true
managementServer : System.Object
name             : devvc
reference        : System.Object
address          : devvc
createdDate      : 1407348515612
datastores       : {System.Object, System.Object, System.Object, System.Object...}
earliestRetrievedHistoricEvent : 1407348514934
mgmtServiceVersion : VMware vCenter Server 5.5.0 build-1378903
port             : 443
rootFolder       : System.Object
runtimeServers   : {System.Object, System.Object, System.Object, System.Object...}
serverType       : VMWARE_VC
```

**Note:** Any property that is of type `System.Object` can be navigated further using the dot notation.

## Debugging PowerShell

If you see errors, check the [Troubleshooting](#) section.

If you are not getting the expected result or an unknown exception is thrown, check your input parameters in a PowerShell console session:

```
$xml = Convert-ObjectToXml $psObject
```

The content of this `$xml` variable shows what has been sent to the vCommander REST Service, allowing you to correct the PowerShell DTOs you are sending to the service.

## Other custom clients

The vCommander REST service will work with any other third-party tools that support the following:

- basic HTTP request operations such as GET, POST, DELETE and PUT
- the ability to send a security token in HTTP request headers or HTTP Basic access authentication

The REST service output and inputs parameters are in XML format. If you need to create your own custom client, please use one of the provided clients to view the XML structure of DTOs. This is a good starting point. Creating a custom client is out of scope of this document.

## Using the REST API with a Service Portal account

When the account used to access the REST service has a Service Portal role (as opposed to a vCommander role), there are a few extra considerations.

### Supply the organizational context

You must supply the organizational context at login, using the parameter `organizationId`. Note that for security reasons, a vCommander account must use the REST API to retrieve the organization ID first. You can get a list all organizations ID or a specific organization ID. Here's a PowerShell example showing how to retrieve a single organization ID:

```
# Get the ID for the organization
$orgId = (Get-OrganizationByName -name "org_name").Organization.id
```

### PowerShell: Use Connect-Client2 to log in and Disconnect-Client to log out

When using the PowerShell client as a Service Portal user, you must log in to the REST API using the command `Connect-Client2`, rather than `Connect-Client`.

**Note:** Embotics recommends using `Connect-Client2` for all connections to the REST API. This API includes all of the functionality of its predecessor (`Connect-Client`), plus support for a subset of the functionality using Service Portal account credentials and organizational context.

For example:

```
Connect-Client2 -url "https://myhost/webservices/services/rest/v2" -credentials
$mycreds -ssltrustall -organizationId 123
```

**Note:** If an account has both a vCommander role and a Service Portal role, passing -1 or 0 for the organization ID when calling `Connect-Client2` will connect the REST API client to vCommander instead of to the Service Portal.

When using `Connect-Client2` to log in, make sure to disconnect your session when finished with the `Disconnect-Client` method.

## If you need to switch organizations

If you need to switch organizations, call `Switch-Organization` with a different value for the `organizationId` parameter. The user is then logged out of the first organization and logged into the second organization.

If the Service Portal role has the Manage Organizations permission, a call to retrieve quota information will return information for all organizations the user is a member of, as well as for all organization members. If the Service Portal role does not have the Manage Organizations permission, the call will return only their own and the organization's quota information.

## Restricting REST API access to a specific host name or IP address

When you restrict access to vCommander to a specific host name or IP address, you can also restrict access to the REST API to the same host name or IP address.

To restrict API access:

1. In vCommander, go to **Configuration > System Configuration**.
2. On the Service Access tab, under **vCommander and Service Portal**, click **Edit**.
3. Enable **Restrict REST API access to vCommander Host/IP**.

This option can only be enabled if you have configured the **vCommander Host/IP** field.

4. Click **Save Settings**, and click **Yes** to confirm the change.

For more information, click the **Help** button.

## Using a search query to restrict the data returned

A subset of the APIs support a search query to restrict the data returned. You can use querying to filter results by:

- request type
- request state, such as Pending Completion
- request date, or how long a request has been in progress
- workflow name
- workflow step name
- whether a workflow has a failed step

The search query starts with the "q" parameter, followed by an equals symbol and one or more search clauses.

For example, the following query retrieves all new service requests in the Pending Completion state where the workflow name contains "Linux" or "Windows" and where the workflow contains a step with the name "Run Chef Client" that has failed:

```
/requests?q=(state -eq "Pending Completion") -and (type -eq "New Service")
-and ((workflow.name -contains "Linux") -or (workflow.name -contains "Windows")) -and
(workflow.currentStep.name -eq "Run Chef Client") -and (workflow.currentStep.fail -eq
true)
```

The following query retrieves all new service requests in the Pending Completion state that have been in progress for at least five days (assuming today is 2016/03/05):

```
/requests?q=(state -eq "Pending Completion") -and (type -eq "New Service") -and
(requestDate -le "2016/03/01 00:00:00" )
```

## Examples

The following examples show the Curl, PowerShell and Java syntax for a simple query that retrieves new service requests in the Pending Completion state.

### Curl

**Note:** For clarity, this example is non-URL encoded. The search query must be URL encoded.

```
curl -sk -u "user:pass" "https://{host}/webservices/services/rest/v2/requests?q=state
-eq \"Pending Completion\""
```

### PowerShell

```
$q = 'state -eq "Pending Completion"'
$request = Get-ServiceRequest -query $q
```

### Java

```
String q = "state -eq \"Pending Completion\""
RestClient client = new RestClient()
WSRequestCollection requests = client.getRequests(q);
```

## Constructing the query

Each query clause consists of three elements: a field, an operator and a value.

### Field

The name of a property (field) in the modeled object. For example:

```
workflow.name
workflow.currentStep.name
```

Fields are case insensitive.

## Operator

A test that is performed on the data to provide a match. See [Available Operators](#) below. Operators are case insensitive.

## Value

The content of the field that is tested. Three value types are supported:

- **String** values must be surrounded by double quotes. For example: "Pending Completion". A **Date** is an instance of the String value type. The Date format is YYYY/mm/dd hh:mm:ss, for example, "2010/01/13 13:33:20"
- **Char** values must be surrounded by single quotes. For example, 't'.
- **Boolean** values are either **true** or **false** (without quotes).

## Compound clauses

Clauses can be combined using the relational operators **-and** and **-or**. Chaining of up to five relational operators is supported. You can use parentheses to group clauses.

## Count and pagination

Any API that allows querying includes the following default fields to limit the returned results:

```
size=100
offset=0
```

These fields are used for pagination and can be overridden. For example:

Find the number of requests in the Pending Completion state:

```
1) /requests/count?q=(state -eq "Pending Completion") returns 101
```

Return the first batch of Pending Completion requests:

```
2) /requests?q=(state -eq "Pending Completion")&size=100&offset=0
```

Return the second batch of Pending Completion requests:

```
3) /requests?q=(state -eq "Pending Completion")&size=100&offset=101
```

## URL encoding

All 'q' queries must be URL encoded. URL-reserved characters (such as &) must be URL encoded. The semicolon, comma and backslash must be backslash-escaped when they appear in an expression. Note that the vCommander REST API PowerShell client performs the URL encoding automatically.

## Available operators

| Operator | Description  |
|----------|--------------|
| -eq      | Equal to     |
| -ne      | Not equal to |

| Operator     | Description  |
|--------------|--|
| -gt          | Greater than; only applies to numerical and date fields                          |
| -ge          | Greater than or equals; only applies to numerical and date fields                |
| -lt          | Less than; only applies to numerical and date fields                             |
| -le          | Less than or equal to; only applies to numerical and date fields                 |
| -contains    | String containment expression; case insensitive                                  |
| -notcontains | String containment expression (negation); case insensitive                       |
| -and         | Logical AND; combines two conditions and requires both of them to be true        |
| -or          | Logical OR; combines two conditions and requires at least one of them to be true |

### Operators supported for each field

| Field                       | Supported Operators               | Supported value type                 |
|-----------------------------|-----------------------------------|--------------------------------------|
| type                        | -eq, -ne                          | String                               |
| state                       | -eq, -ne                          | String                               |
| requestDate                 | -le, -lt, -gt, -ge, -eq, -ne      | String                               |
| workflow.name               | -eq, -ne, -contains, -notcontains | String                               |
| workflow.currentStep.name   | -eq, -ne, -contains, -notcontains | String                               |
| workflow.currentStep.failed | -eq, -ne                          | Boolean (true/false, without quotes) |

## Examples

These examples use the PowerShell client.

### Example 1: Connect your automated build machine to the vCommander Service Catalog

This example shows how you can automatically add your latest software build to the vCommander Service Catalog.

- a) Delete existing published service; add a new one backed by the new VM created by the build machine:

```
##### Delete existing published service #####
#Look up and remove the existing published service named "Latest Build"
$existingPS = Get-PublishedServiceByName -name "Latest Build"
Remove-PublishedService -psId $existingPS.PublishedService.id

##### Create new published service #####
#Look up new VM (recently created by the build machine) called "BuildVM-001"
$vm = Get-VMS -vmName "BuildVM-001"

#Create a published service component DTO, backed by the VM we just found
$pscdto = New-DTOTemplateObject -DTOTagName "PublishedServiceComponent"
$pscdto.PublishedServiceComponent.description = "The latest VM created by the build machine"
$pscdto.PublishedServiceComponent.name =
$vm.VirtualMachineCollection.VirtualMachines[0].name
```

```

$pscDTO.PublishedServiceComponent.ref.displayName =
$vm.VirtualMachineCollection.VirtualMachines[0].displayName
$pscDTO.PublishedServiceComponent.ref.id =
$vm.VirtualMachineCollection.VirtualMachines[0].id
$pscDTO.PublishedServiceComponent.ref.type =
$vm.VirtualMachineCollection.VirtualMachines[0].type

#Create a published service DTO and add the component to it
$psDTO = New-DTOTemplateObject -DTOTagName "PublishedService"
$psDTO.PublishedService.description = "This is a single-VM service for the latest VM
created by the build machine"
$psDTO.PublishedService.name = "Latest Build"
$psDTO.PublishedService.serviceComponents = @()
$psDTO.PublishedService.serviceComponents += $pscDTO.PublishedServiceComponent
$psDTO.PublishedService.publishState = "PUBLISHED"
$psDTO.PublishedService.useStaticComponentForms = $false
$psDTO.PublishedService.serviceUsers = @()

#Add the service to the service catalog
$createPS = New-PublishedService -psDTO $psDTO

```

b) Update the existing published service with the new VM as its component:

```

##### Update existing published service #####

#Look up existing published service named "Latest Build"
$existingPS = Get-PublishedServiceByName -name "Latest Build"

#Look up new VM (recently created by the build machine) called "BuildVM-002"
$vm = Get-VMS -vmName "BuildVM-002"

#Create a published service component DTO, backed by the VM we just found
$pscDTO = New-DTOTemplateObject -DTOTagName "PublishedServiceComponent"
$pscDTO.PublishedServiceComponent.description = "The latest VM created by the build
machine"
$pscDTO.PublishedServiceComponent.name =
$vm.VirtualMachineCollection.VirtualMachines[0].name
$pscDTO.PublishedServiceComponent.ref.displayName =
$vm.VirtualMachineCollection.VirtualMachines[0].displayName
$pscDTO.PublishedServiceComponent.ref.id =
$vm.VirtualMachineCollection.VirtualMachines[0].id
$pscDTO.PublishedServiceComponent.ref.type =
$vm.VirtualMachineCollection.VirtualMachines[0].type

#Clear out component(s); add the new one we just created
$existingPS.PublishedService.serviceComponents = @()
$existingPS.PublishedService.serviceComponents += $pscDTO.PublishedServiceComponent

#Update the existing published service
$updatePublishedService = Update-PublishedService -psId
$existingPS.PublishedService.id -updatePS $existingPS

```

## Example 2: Initiate a service request from an external system

This example and the next show how you can integrate vCommander with an IT service management tool (such as ServiceNow or Remedy). This particular example shows how to initiate a vCommander service request from an external system.

```

##### Submit new service request #####

#Lookup existing published service named "Latest Build"
$existingPS = Get-PublishedServiceByName -name "Latest Build"

#Lookup required parameters for the service request
$requestParams = Get-PSRequestParams -psId $existingPS.PublishedService.id

```

```

#Pull out and fill in service form parameters (varies depending on the form
configuration)
$quantityElement =
$requestParams.PSDeployParamCollection.serviceformElements.RequestFormElements |
where-Object {$_.label -eq "Quantity"}

$pciApplicableElement =
$requestParams.PSDeployParamCollection.serviceformElements.RequestFormElements |
where-Object {$_.label -eq "PCI Applicable"}

#Configure correct form values
$quantityElement.value = "1"
$pciApplicableElement.value = "No"

#Pull out and fill in component form parameters (varies depending on the form
configuration). Repeat for all components
$component1DeployParam = $requestParams.PSDeployParamCollection.componentParams |
where-Object {$_.componentName -match 'win2k8-WebServer' }

#Pull out all the component form elements so we can pull out the data (repeat for all
components)
$expiryDateElement = $component1DeployParam.formElements.RequestFormElements | where-
Object {$_.label -eq "Expiry Date"}
$memorySizeElement = $component1DeployParam.formElements.RequestFormElements | where-
Object {$_.label -eq "Memory Size"}
$cpuCountElement = $component1DeployParam.formElements.RequestFormElements | where-
Object {$_.label -eq "CPU Count"}
$storageElement = $component1DeployParam.formElements.RequestFormElements | where-
Object {$_.label -eq "Storage"}

#Configure correct form values
$expiryDateElement.value = "0" #never expires
$memorySizeElement.value = 8192 #(MB)
$cpuCountElement.value = 2

# Storage : set up the existing disks
$disk1 = $storageElement.RequestedDisks | where-Object {$_.label -eq "Hard disk 1" }
$disk1.diskSizeInKB = 33554432
$disk1.storageTierLabel = "Storage Tier 1"

$disk2 = $storageElement.RequestedDisks | where-Object {$_.label -eq "Hard disk 2" }
$disk2.diskSizeInKB = 10485760
$disk2.storageTierLabel = "Storage Tier 1"

$disk3 = $storageElement.RequestedDisks | where-Object {$_.label -eq "Hard disk 3" }
$disk3.diskSizeInKB = 5242880
$disk3.storageTierLabel = "Storage Tier 1"

#Create a new disk (if applicable)
$thirdRequestedDisk = New-DTOTemplateObject "VMRequestDisk"
$thirdRequestedDisk.VMRequestDisk.diskSizeInKB = 102400
$thirdRequestedDisk.VMRequestDisk.storageTierLabel = "Storage Tier 5"
$storageElement.RequestedDisks += $thirdRequestedDisk.VMRequestDisk

#Submit the request
$result = New-ServiceRequest -psId $existingPS.PublishedService.id -requestParams
$requestParams

```

### Example 3: Approve and deploy a service request from an external system

This example shows how to approve and deploy a vCommander service request from an external system (such as an IT service management tool).

```

##### Approve new service request #####
#Retrieve all service requests
$serviceRequests = Get-ServiceRequests

#Check state of the most recent service request and approve

```



```

if ($serviceRequests.RequestCollection.Requests[0].state -eq "PENDING_APPROVAL") {
    Approve-Request -requestId $serviceRequests.RequestCollection.Requests[0].id -
    comment "Approved!"
} else {
    echo "Error: This request is not pending approval."
}

##### Deploy the service request #####

#Assign the recently approved service request ID to a variable
$requestId = $serviceRequests.RequestCollection.Requests[0].id

#Get the requested service
$requestService = Get-RequestedServices -requestId $requestId
$requestServiceId =
$requestService.RequestedServiceCollection.RequestedServices[0].id

#Get the requested component
$requestComponent = Get-RequestedComponents -requestId $requestId -
requestedServiceId $requestServiceId
$requestComponentId =
$requestComponent.RequestedComponentCollection.RequestedComponents.id

#Create a deployment params DTO
$deployParamsDTO = New-DTOTemplateObject -DTOTagName "DeploymentParams"

#Create a custom attribute DTO
$attributeDTO = New-DTOTemplateObject -DTOTagName "CustomAttribute"
$attributeDTO.CustomAttribute.allowedvalues = @() #not important
$attributeDTO.CustomAttribute.description = $null #not important
$attributeDTO.CustomAttribute.targetManagedObjectTypes = @() #not important
$attributeDTO.CustomAttribute.name= "SLA"
$attributeDTO.CustomAttribute.value = "Gold"

#Set custom attribute(s) for deployment
$deployParamsDTO.DeploymentParams.CustomAttributes = @()
$deployParamsDTO.DeploymentParams.CustomAttributes += $attributeDTO.CustomAttribute

#Create an ownership DTO
$ownershipDTO = New-DTOTemplateObject -DTOTagName "Ownership"

#Create a user; the only information we need is the loginId
$userDTO = New-DTOTemplateObject -DTOTagName "OwnerInfo"
$userDTO.OwnerInfo.id = -1
$userDTO.OwnerInfo.loginId = "superuser"
$userDTO.OwnerInfo.itContact = $false
$userDTO.OwnerInfo.primary = $false
$userDTO.OwnerInfo.email = $null
$userDTO.OwnerInfo.displayName = $null

#Clear out user(s); add the new one we just created
$ownershipDTO.Ownership.Owners = @()
$ownershipDTO.Ownership.Owners += $userDTO.OwnerInfo

#Look up existing organization called "Default Organization"
$org = Get-OrganizationByName -name "Default Organization"

#Create a reference to this organization. we really just need the ID and type.
$orgReferencedDTO = New-DTOTemplateObject -DTOTagName "ManagedObjectReference"
$orgReferencedDTO.ManagedObjectReference.id = $org.Organization.id
$orgReferencedDTO.ManagedObjectReference.type = "ORGANIZATION"

#Add the organization
$ownershipDTO.Ownership.organization = $orgReferencedDTO.ManagedObjectReference

#Set ownership for deployment
$deployParamsDTO.DeploymentParams.ownership = $ownershipDTO.Ownership

#Set other parameters for deployment
$deployParamsDTO.DeploymentParams.approved = $true
$deployParamsDTO.DeploymentParams.clusterName = $null
$deployParamsDTO.DeploymentParams.createLinkedClone = $false

```

```

$deployParamsDTO.DeploymentParams.datacenterName = "Engineering"
$deployParamsDTO.DeploymentParams.datastoreName = "SAN1"
$deployParamsDTO.DeploymentParams.endOfLife = $false
$deployParamsDTO.DeploymentParams.expiryDate = "2013/12/31"
$deployParamsDTO.DeploymentParams.folderName = "Engineering VMs"
$deployParamsDTO.DeploymentParams.highAvailability = $false
$deployParamsDTO.DeploymentParams.managementServerName = "vsphere1"
$deployParamsDTO.DeploymentParams.memoryInMB = 512
$deployParamsDTO.DeploymentParams.numCPU = 1
$deployParamsDTO.DeploymentParams.powerOn = $true
$deployParamsDTO.DeploymentParams.releaseTarget = $true
$deployParamsDTO.DeploymentParams.releaseTargetComment = "It's ready"
$deployParamsDTO.DeploymentParams.resourcePoolName = "Build VMs"
$deployParamsDTO.DeploymentParams.suspect = $false
$deployParamsDTO.DeploymentParams.targetHostName = "ESX1"

#Deploy the service
$result = New-RequestedServiceDeployment -requestId $requestId -requestedServiceId
$requestServiceId -deploymentParams $deployParamsDTO

```

## Example 4: Add a datastore to automated deployment destinations

This example shows how to add a datastore to automated deployment destinations.

```

#Add to destinations that have at least one datastore with same storage tier as "DS
200"
Add-DatastoreToDeploymentDestinations -datastore "DS 200" -matchexistingtier $true

#Add to destinations ignoring storage tier match
Add-DatastoreToDeploymentDestinations -datastore "DS 200" -matchexistingtier $false

```

## Example 5: Create an organization and assign cost quota

This example shows how to create an organization and configure cost quotas for the organization and its members.

```

#Set daily cost quota for organization
$costQuota = New-DTOTemplateObject -DTOTagName "CostQuota"
$costQuota.CostQuota.dailyCost = 10000

#Set daily cost quota for user
$user1Quota = New-DTOTemplateObject -DTOTagName "CostQuota"
$user1Quota.CostQuota.dailyCost = 1000

#Set organization name and daily cost quota
$orgDto = New-DTOTemplateObject -DTOTagName "Organization"
$orgDto.Organization.name = "Customer #3"
$orgDto.Organization.PSObject.Properties.Remove("resourceQuota")
Add-Member -InputObject $orgDto.Organization -MemberType NoteProperty -Name
"costQuota" -Value $costQuota.CostQuota -Force

#Set user account as organization member and set role and daily cost quota
$user1DTO = New-DTOTemplateObject -DTOTagName "OrganizationUser"
$user1DTO.OrganizationUser.userId = "manager"
$user1DTO.OrganizationUser.manager = $true
$user1DTO.OrganizationUser.portalRole = "Customer"
Add-Member -InputObject $user1DTO.OrganizationUser -MemberType NoteProperty -Name
"costQuota" -Value $user1Quota.CostQuota -Force

#Set another user account as organization member and set role
$user2DTO = New-DTOTemplateObject -DTOTagName "OrganizationUser"
$user2DTO.OrganizationUser.userId = "superuser"
$user2DTO.OrganizationUser.manager = $false
$user2DTO.OrganizationUser.portalRole = "Customer"

#Add members to organization

```

```
$orgDto.Organization.Members = @()
$orgDto.Organization.Members += $user1Dto.OrganizationUser
$orgDto.Organization.Members += $user2Dto.OrganizationUser
```

```
#Create new organization
$org = New-Organization -orgDto $orgDto
```

## Example 6: Create an organization and assign resource quota

This example shows how to create an organization and configure CPU and memory quotas for the organization and its members.

```
#Set resource quota for organization
$resourceQuota = New-DTOTemplateObject -DTOTagName "ResourceQuota"
$resourceQuota.ResourceQuota.CPUCount = 1000
$resourceQuota.ResourceQuota.memoryInGB = 2000
$resourceQuota.ResourceQuota.stoppedVmsAllowPowerOn = $true
$resourceQuota.ResourceQuota.stoppedVmsCalculateOnlyStorage = $true
$resourceQuota.ResourceQuota.storageInGB = 3000

#Set resource quota for user
$user1ResourceQuota = New-DTOTemplateObject -DTOTagName "ResourceQuota"
$user1ResourceQuota.ResourceQuota.CPUCount = 100
$user1ResourceQuota.ResourceQuota.memoryInGB = 200
$user1ResourceQuota.ResourceQuota.storageInGB = 300

#Set organization name and resource quota
$orgDto = New-DTOTemplateObject -DTOTagName "Organization"
$orgDto.Organization.name = "Customer #4"
$orgDto.Organization.resourceQuota = $resourceQuota.ResourceQuota

#Set user account as organization member and set role and resource quota
$user1Dto = New-DTOTemplateObject -DTOTagName "OrganizationUser"
$user1Dto.OrganizationUser.userId = "manager"
$user1Dto.OrganizationUser.manager = $true
$user1Dto.OrganizationUser.portalRole = "Customer"
Add-Member -InputObject $user1Dto.OrganizationUser -MemberType NoteProperty -Name
"resourceQuota" -Value $user1ResourceQuota.ResourceQuota -Force

#Set another user account as organization member and set role
$user2Dto = New-DTOTemplateObject -DTOTagName "OrganizationUser"
$user2Dto.OrganizationUser.userId = "superuser"
$user2Dto.OrganizationUser.manager = $false
$user2Dto.OrganizationUser.portalRole = "Customer"

#Add members to organization
$orgDto.Organization.Members = @()
$orgDto.Organization.Members += $user1Dto.OrganizationUser
$orgDto.Organization.Members += $user2Dto.OrganizationUser

#Create an organization
$org = New-Organization -orgDto $orgDto
```

## Example 7: Set a VM's compliance state

This example shows how to set compliance metadata for VMs.

```
$vm = Get-VMs -vmName "VM 001" -max 1
$vmId = $vm.VirtualMachineCollection.VirtualMachines[0].id

#Create a custom attribute DTO - only name-value pair is needed
$attribute1DTO = New-DTOTemplateObject -DTOTagName "CustomAttribute"
$attribute1DTO.CustomAttribute.allowedValues = @() #not important
$attribute1DTO.CustomAttribute.description = $null #not important
$attribute1DTO.CustomAttribute.targetManagedObjectTypes = @() #not important
$attribute1DTO.CustomAttribute.name= "SLA"
$attribute1DTO.CustomAttribute.value = "Gold"

#Create a custom attribute DTO - only name-value pair is needed
$attribute2DTO = New-DTOTemplateObject -DTOTagName "CustomAttribute"
$attribute2DTO.CustomAttribute.allowedValues = @() #not important
$attribute2DTO.CustomAttribute.description = $null #not important
$attribute2DTO.CustomAttribute.targetManagedObjectTypes = @() #not important
$attribute2DTO.CustomAttribute.name= "Cost Center"
$attribute2DTO.CustomAttribute.value = "Cost Center #1"

$complianceData = New-DTOTemplateObject -DTOTagName "ComplianceData"
$complianceData.ComplianceData.expiryDate = "2014/04/18"
$complianceData.ComplianceData.primaryOwner.loginId = "manager" #only the loginId is
relevant

#Add the attribute
$complianceData.ComplianceData.Attributes = @()
$complianceData.ComplianceData.Attributes += $attribute1DTO.CustomAttribute
$complianceData.ComplianceData.Attributes += $attribute2DTO.CustomAttribute

Set-ComplianceData -vmId $vmId -data $complianceData
```

## Example 8: Set VM ownership

This example shows how to assign ownership to VMs.

```
#Retrieve the VM
$vm = Get-VMs -vmName "VM 001" -max 1
$vmId = $vm.VirtualMachineCollection.VirtualMachines[0].id

#Create an ownership DTO
$ownershipDto = New-DTOTemplateObject -DTOTagName "Ownership"
$ownershipDto.Ownership.organization.displayName = $org.Organization.name
$ownershipDto.Ownership.organization.id = $org.Organization.id

#Set ownership properties for existing user account
$user1DTO = New-DTOTemplateObject -DTOTagName "OwnerInfo"
$user1DTO.OwnerInfo.id = -1
$user1DTO.OwnerInfo.loginId = "user1"
$user1DTO.OwnerInfo.itContact = $true
$user1DTO.OwnerInfo.primary = $false
$user1DTO.OwnerInfo.email = $null
$user1DTO.OwnerInfo.displayName = $null

#Set ownership properties for another existing user account
$user2DTO = New-DTOTemplateObject -DTOTagName "OwnerInfo"
$user2DTO.OwnerInfo.id = -1
$user2DTO.OwnerInfo.loginId = "manager"
$user2DTO.OwnerInfo.itContact = $false
$user2DTO.OwnerInfo.primary = $true
$user2DTO.OwnerInfo.email = $null
$user2DTO.OwnerInfo.displayName = $null

#Add the user to the ownership structure
$ownershipDto.Ownership.PSObject.Properties.Remove("organization")
```

```
$ownershipDto.Ownership.Owners = @()
$ownershipDto.Ownership.Owners += $user1Dto.OwnerInfo
$ownershipDto.Ownership.Owners += $user2Dto.OwnerInfo

Set-Ownership -vmId $vmId -dto $ownershipDto
```

## Example 9: Create an IP pool without specifying networks

The first of our IP pool examples does not specify networks.

```
#Retrieve the management server
$managementServers = Get-ManagementServers
$msid = $managementServers.ManagementServerCollection.ManagementServers | Where-Object
{ $_.name -match "manta" } |
Select -ExpandProperty "id"

#Retrieve the datacenter
$datacenters = Get-Datacenters -msid $msid
$dc = $datacenters.DatacenterCollection.Datacenters | Where-Object { $_.name -match
"Engineering" }

#Create an IP Pool template
$ipPoolDto = New-DTOTemplateObject -DTOTagName "IpPool"
$ipPoolDto.IpPool.name = "Pool #2"
$ipPoolDto.IpPool.freeIpWarningThreshold = 3

#Set up the datacenter
$ipPoolDto.IpPool.datacenter.id = $dc.id
$ipPoolDto.IpPool.datacenter.displayName = $dc.name #Not required

#Set up the network configuration
$ipPoolDto.IpPool.networkConfig.defaultGateway = "10.10.10.251"
$ipPoolDto.IpPool.networkConfig.subnetMask = "255.255.255.0"
$ipPoolDto.IpPool.networkConfig.dnsSuffix = "embotics.com"
$ipPoolDto.IpPool.networkConfig.primaryDns = "10.10.10.1"
$ipPoolDto.IpPool.networkConfig.secondaryDns = "10.10.10.2"

#Set up IP range
$ipSlot1 = New-DTOTemplateObject -DTOTagName "Ipv4Range"
$ipSlot1.Ipv4Range.from = "10.10.10.3"
$ipSlot1.Ipv4Range.to = "10.10.10.10"

$ipSlot2 = New-DTOTemplateObject -DTOTagName "Ipv4Range"
$ipSlot2.Ipv4Range.from = "10.10.10.15"
$ipSlot2.Ipv4Range.to = "10.10.10.20"

$ipSlot3 = New-DTOTemplateObject -DTOTagName "Ipv4Range"
$ipSlot3.Ipv4Range.from = "10.10.10.30"
$ipSlot3.Ipv4Range.to = "10.10.10.50"

#Assign the IP slots to the range structure
$ipPoolDto.IpPool.range = @($ipSlot1.Ipv4Range, $ipSlot2.Ipv4Range,
$ipSlot3.Ipv4Range)

#Create the IP pool
$taskInfo = New-IPPool -ipPoolDto $ipPoolDto
```

## Example 10: Create an IP pool and specify networks

Our second IP pool example includes network configuration.

```
#Retrieve the management server
$managementServers = Get-ManagementServers
$msid = $managementServers.ManagementServerCollection.ManagementServers | Where-Object
{ $_.name -match "manta" } | Select -ExpandProperty "id"

#Retrieve the datacenter
```

```

$datacenters = Get-Datacenters -msId $msid
$dc = $datacenters.DatacenterCollection.Datacenters | where-Object { $_.name -match "Engineering"}

#Create an IP Pool template
$ipPoolDto = New-DTOTemplateObject -DTOTagName "IpPool"
$ipPoolDto.IpPool.name = "Pool #2"
$ipPoolDto.IpPool.freeIpWarningThreshold = 3

#Set up the datacenter
$ipPoolDto.IpPool.datacenter.id = $dc.id
$ipPoolDto.IpPool.datacenter.displayName = $dc.name #Not required

#Set up the network configuration
$ipPoolDto.IpPool.networkConfig.defaultGateway = "10.10.10.251"
$ipPoolDto.IpPool.networkConfig.subnetMask = "255.255.255.0"
$ipPoolDto.IpPool.networkConfig.dnsSuffix = "embotics.com"
$ipPoolDto.IpPool.networkConfig.primaryDns = "10.10.10.1"
$ipPoolDto.IpPool.networkConfig.secondaryDns = "10.10.10.2"

#Set up IP range
$ipSlot1 = New-DTOTemplateObject -DTOTagName "Ipv4Range"
$ipSlot1.Ipv4Range.from = "10.10.10.3"
$ipSlot1.Ipv4Range.to = "10.10.10.10"

$ipSlot2 = New-DTOTemplateObject -DTOTagName "Ipv4Range"
$ipSlot2.Ipv4Range.from = "10.10.10.15"
$ipSlot2.Ipv4Range.to = "10.10.10.20"

$ipSlot3 = New-DTOTemplateObject -DTOTagName "Ipv4Range"
$ipSlot3.Ipv4Range.from = "10.10.10.30"
$ipSlot3.Ipv4Range.to = "10.10.10.50"

#Assign the IP slots to the range structure
$ipPoolDto.IpPool.range = @($ipSlot1.Ipv4Range, $ipSlot2.Ipv4Range,
$ipSlot3.Ipv4Range)

$networks = Get-AvailableNetworks -dcid $dc.id
$network1 = $networks.ManagedObjectReferenceCollection.ManagedObjectReferences |
where-Object { $_.displayName -match "Dev Network"}
$network2 = $networks.ManagedObjectReferenceCollection.ManagedObjectReferences |
where-Object { $_.displayName -match "PV Network"}

$poolNetworks = @($network1,$network2)
Add-Member -InputObject $ipPoolDto.IpPool -MemberType NoteProperty -Name "networks" -
value $poolNetworks -Force

#Create the IP pool
$taskInfo = New-IPPool -ipPoolDto $ipPoolDto

```

## Example 11: Create a deployment destination for vCenter

This section contains several examples showing how to create a vCenter deployment destination.

In the first example, a globally available deployment destination is created on a vCenter managed system. VMs are deployed to a cluster and connected to a specified network.

```

#Retrieve management server
$mobjectCollection = Get-ManagedObjectByName -name "manta" -type "MANAGEMENTSERVER"
$managementServer = $mobjectCollection.ManagedObjectCollection.managedObjects[0]

#Retrieve the folder to deploy the VMs to
$folderCollection = Get-FoldersByName -name "Dev Infrastructure"
$folder = $folderCollection.FolderCollection.Folders[0]

#Retrieve the target (cluster)
$cluster = Get-ClusterByName -msId $managementServer.id -clusterName "Engineering
Cluster"

```



```

#To retrieve the resource pool
#$rpCollection = Get-ManagedObject -name "Dev Infrastructure" -type "RESOURCEPOOL"
#$rp = $rpCollection.ManagedObjectCollection.managedObjects[0]

#To retrieve the host
#$hostCollection = Get-ManagedObject -name "paragon.embotics.com" -type
"RUNTIMESERVER"
#$targetHost = $hostCollection.ManagedObjectCollection.managedObjects[0]

#Retrieve the network to connect the VM to
$networkCollection1 = Get-ManagedObject -name "Dev Network" -type "NETWORK"
$networkCollection2 = Get-ManagedObject -name "PV Network" -type "NETWORK"

$network1 = $networkCollection1.ManagedObjectCollection.managedObjects[0]
$network2 = $networkCollection2.ManagedObjectCollection.managedObjects[0]

#Retrieve the datastores
$dsCollection1 = Get-ManagedObject -name "Paragon" -type "DATASTORE"
$dsCollection2 = Get-ManagedObject -name "Renegade" -type "DATASTORE"

$ds1 = $dsCollection1.ManagedObjectCollection.managedObjects[0]
$ds2 = $dsCollection2.ManagedObjectCollection.managedObjects[0]

#Create a VMware deployment destination DTO
$ddDTO = New-DTOTemplateObject -DToTagName "VMwareDeploymentDestination"
$ddDTO.VMwareDeploymentDestination.name = "Default #1"
$ddDTO.VMwareDeploymentDestination.assignIpFromPool = $false
$ddDTO.VMwareDeploymentDestination.diskFormat = "THICK" #Possible values are THICK,
THIN, and SAME_AS_SOURCE. See DiskFormat DTO.
$ddDTO.VMwareDeploymentDestination.peakCapacity = $true #to use average capacity, set
to $false

#Specify the management server
$ddDTO.VMwareDeploymentDestination.managementServer = $managementServer

#Specify the folder
$ddDTO.VMwareDeploymentDestination.folder = $folder

#Specify the target
$ddDTO.VMwareDeploymentDestination.target = $cluster.Cluster

#Specify the network
$ddDTO.VMwareDeploymentDestination.PSObject.Properties.Remove('network')
Add-Member -InputObject $ddDTO.VMwareDeploymentDestination -MemberType NoteProperty -
Name "networks" -Value @($network1, $network2) -Force

#Specify the datastores
$ddDTO.VMwareDeploymentDestination.datastores = @($ds1, $ds2)

#Create the deployment destination
$createDD = New-VMwareDeploymentDestination -ddDTO $ddDTO

```

In this example, a globally available deployment destination is created on a vCenter managed system. VMs are deployed to a datacenter and to a specific host, and are connected to the same network as the source image.

```

#Retrieve management server
$mobjectCollection = Get-ManagedObjectByName -name "manta" -type "MANAGEMENTSERVER"
$managementServer = $mobjectCollection.ManagedObjectCollection.managedObjects[0]

#Retrieve the datacenter
$datacenterCollection = Get-ManagedObject -name "Engineering" -type "DATACENTER"
$dc = $datacenterCollection.ManagedObjectCollection.managedObjects[0]

#Retrieve the host
$hostCollection = Get-ManagedObject -name "paragon.embotics.com" -type "RUNTIMESERVER"
$targetHost = $hostCollection.ManagedObjectCollection.managedObjects[0]

```

```

#Retrieve the datastores
$dsCollection1 = Get-ManagedObject -name "Paragon" -type "DATASTORE"
$ds1 = $dsCollection1.ManagedObjectCollection.managedObjects[0]

#Create a vmware deployment destination DTO
$ddDTO = New-DTOTemplateObject -DOTOTagName "VMwareDeploymentDestination"
$ddDTO.VMwareDeploymentDestination.name = "Default #2"
$ddDTO.VMwareDeploymentDestination.assignIpFromPool = $false
$ddDTO.VMwareDeploymentDestination.diskFormat = "THICK"
$ddDTO.VMwareDeploymentDestination.peakCapacity = $true

#Specify the management server
$ddDTO.VMwareDeploymentDestination.managementServer = $managementServer

#Specify the datacenter
$ddDTO.VMwareDeploymentDestination.PSObject.Properties.Remove("folder")
Add-Member -InputObject $ddDTO.VMwareDeploymentDestination -MemberType NoteProperty -
Name "datacenter" -Value $dc -Force

#Specify the target
$ddDTO.VMwareDeploymentDestination.target = $targetHost

#Network is same as source
$ddDTO.VMwareDeploymentDestination.PSObject.Properties.Remove("network")

#Specify the datastores
$ddDTO.VMwareDeploymentDestination.datastores = @($ds1)

#Create the deployment destination
$createdDD = New-VMwareDeploymentDestination -ddDTO $ddDTO

```

In this example, a vCenter deployment destination is assigned to a specific organization and to specific users. VMs are placed in a folder, deployed to a specified cluster, and are connected to a specific network.

```

#Retrieve management server
$mobjectCollection = Get-ManagedObjectByName -name "manta" -type "MANAGEMENTSERVER"
$managementServer = $mobjectCollection.ManagedObjectCollection.managedObjects[0]

#Retrieve the folder to deploy the VMs to
$folderCollection = Get-FoldersByName -name "Dev Infrastructure"
$folder = $folderCollection.FolderCollection.Folders[0]

#Retrieve the target (cluster)
$cluster = Get-ClusterByName -msId $managementServer.id -clusterName "Engineering
Cluster"

#Retrieve the network to connect the VM to
$networkCollection1 = Get-ManagedObject -name "Dev Network" -type "NETWORK"
$networkCollection2 = Get-ManagedObject -name "PV Network" -type "NETWORK"

$network1 = $networkCollection1.ManagedObjectCollection.managedObjects[0]
$network2 = $networkCollection2.ManagedObjectCollection.managedObjects[0]

#Retrieve the datastores
$dsCollection1 = Get-ManagedObject -name "Paragon" -type "DATASTORE"
$dsCollection2 = Get-ManagedObject -name "Renegade" -type "DATASTORE"

$ds1 = $dsCollection1.ManagedObjectCollection.managedObjects[0]
$ds2 = $dsCollection2.ManagedObjectCollection.managedObjects[0]

#Create a vmware deployment destination DTO
$ddDTO = New-DTOTemplateObject -DOTOTagName "VMwareDeploymentDestination"
$ddDTO.VMwareDeploymentDestination.name = "Default #3"
$ddDTO.VMwareDeploymentDestination.assignIpFromPool = $false
$ddDTO.VMwareDeploymentDestination.diskFormat = "THICK"
$ddDTO.VMwareDeploymentDestination.peakCapacity = $true

```



```

#Specify the management server
$ddDTO.VMwareDeploymentDestination.managementServer = $managementServer

#Specify the folder
$ddDTO.VMwareDeploymentDestination.folder = $folder

#Specify the target
$ddDTO.VMwareDeploymentDestination.target = $cluster.Cluster

#Specify the network
$ddDTO.VMwareDeploymentDestination.PSObject.Properties.Remove('network')
Add-Member -InputObject $ddDTO.VMwareDeploymentDestination -MemberType NoteProperty -
Name "networks" -Value @($network1, $network2) -Force

#Specify the datastores
$ddDTO.VMwareDeploymentDestination.datastores = @($ds1, $ds2)

#Assigning user(s)
#Create a user; the only information we need is the loginId
$userDTO = New-DTOTemplateObject -DTOTagName "OwnerInfo"
$userDTO.OwnerInfo.id = -1
$userDTO.OwnerInfo.loginId = "superuser"
$userDTO.OwnerInfo.itContact = $false
$userDTO.OwnerInfo.primary = $false
$userDTO.OwnerInfo.email = $null
$userDTO.OwnerInfo.displayName = $null

$userDTO2 = New-DTOTemplateObject -DTOTagName "OwnerInfo"
$userDTO2.OwnerInfo.id = -1
$userDTO2.OwnerInfo.loginId = "manager"
$userDTO2.OwnerInfo.itContact = $false
$userDTO2.OwnerInfo.primary = $false
$userDTO2.OwnerInfo.email = $null
$userDTO2.OwnerInfo.displayName = $null

$users = @($userDTO.OwnerInfo, $userDTO2.OwnerInfo)
Add-Member -InputObject $ddDTO.VMwareDeploymentDestination -MemberType NoteProperty -
Name "users" -Value $users -Force

```

In the following example, a globally available deployment destination is created on a vCenter managed system. VMs are deployed to a resource pool and are connected to the same network as the source image, with fencing based on a distributed switch.

```

#Retrieve management server
$mobjectCollection = Get-ManagedObjectByName -name "manta" -type "MANAGEMENTSERVER"
$managementServer = $mobjectCollection.ManagedObjectCollection.managedObjects[0]

#Retrieve the folder to deploy the VMs to
$folderCollection = Get-FoldersByName -name "Dev Infrastructure"
$folder = $folderCollection.FolderCollection.Folders[0]

#Retrieve the target
$rpCollection = Get-ManagedObject -name "Dev Infrastructure" -type "RESOURCEPOOL"
$rp = $rpCollection.ManagedObjectCollection.managedObjects[0]

#Retrieve the datastores
$dsCollection1 = Get-ManagedObject -name "Paragon" -type "DATASTORE"
$dsCollection2 = Get-ManagedObject -name "Renegade" -type "DATASTORE"

$ds1 = $dsCollection1.ManagedObjectCollection.managedObjects[0]
$ds2 = $dsCollection2.ManagedObjectCollection.managedObjects[0]

#Create a vmware deployment destination DTO
$ddDTO = New-DTOTemplateObject -DTOTagName "VMwareDeploymentDestination"
$ddDTO.VMwareDeploymentDestination.name = "Default #4"
$ddDTO.VMwareDeploymentDestination.assignIpFromPool = $false
$ddDTO.VMwareDeploymentDestination.diskFormat = "SAME_AS_SOURCE"
$ddDTO.VMwareDeploymentDestination.peakCapacity = $false

```

```

#Specify the management server
$ddDTO.VMwareDeploymentDestination.managementServer = $managementServer

#Specify the folder
$ddDTO.VMwareDeploymentDestination.folder = $folder

#Specify the target
$ddDTO.VMwareDeploymentDestination.target = $rp

#Network is same as source
$ddDTO.VMwareDeploymentDestination.PSObject.Properties.Remove("network")

#Specify the datastores
$ddDTO.VMwareDeploymentDestination.datastores = @($ds1, $ds2)

#Retrieve the network
$networkCollection = Get-ManagedObject -name "PV Network" -type "NETWORK"
$network = $networkCollection.ManagedObjectCollection.managedObjects[0]

#Retrieve distributed switch
$distributedSwitches = Get-DistributedSwitches -rpId $rp.id -switchname "dvsPvNetwork"
$distributedSwitch =
$distributedSwitches.ManagedObjectReferenceCollection.ManagedObjectReferences[0]

#Create fenced network config
#Note: The external network must be configured with an IP pool
$fencedNetworkConfig = New-DTOTemplateObject -DTOTagName "FencedNetworkConfig"
$fencedNetworkConfig.FencedNetworkConfig.externalNetwork = $network
$fencedNetworkConfig.FencedNetworkConfig.vlanIds = "1"
$fencedNetworkConfig.FencedNetworkConfig.distributedSwitch = $distributedSwitch

```

In the following example, a globally available deployment destination is created on a vCenter managed system. VMs are placed in a folder and are connected to the same network as the source image, with fencing based on standard switches.

```

#Retrieve management server
$mobjectCollection = Get-ManagedObjectByName -name "autumnrain" -type
"MANAGEMENTSERVER"
$managementServer = $mobjectCollection.ManagedObjectCollection.managedObjects[0]

#Retrieve hosts
$hostCollection = Get-RuntimeServers -msId $managementServer.id
$panoramixHost = $hostCollection.RuntimeServerCollection.RuntimeServers | Where-Object
{$_.displayName -match "panoramix.embotics.com"}

#Retrieve the folder to deploy the VMs to
$folderCollection = Get-FoldersByName -name "Sanity"
$folder = $folderCollection.FolderCollection.Folders[0]

#Retrieve the datastores
$dsCollection1 = Get-ManagedObject -name "Panoramix" -type "DATASTORE"
$ds1 = $dsCollection1.ManagedObjectCollection.managedObjects[0]

#Create a vmware deployment destination DTO
$ddDTO = New-DTOTemplateObject -DTOTagName "VMwareDeploymentDestination"
$ddDTO.VMwareDeploymentDestination.name = "Default #5"
$ddDTO.VMwareDeploymentDestination.assignIpFromPool = $false
$ddDTO.VMwareDeploymentDestination.diskFormat = "SAME_AS_SOURCE"
$ddDTO.VMwareDeploymentDestination.peakCapacity = $false

#Specify the management server
$ddDTO.VMwareDeploymentDestination.managementServer = $managementServer

#Specify the folder
$ddDTO.VMwareDeploymentDestination.folder = $folder

#Specify the target

```

```

$ddDTO.VMwareDeploymentDestination.target = $panoramixHost

#Network is same as source
$ddDTO.VMwareDeploymentDestination.PSObject.Properties.Remove("network")

#Specify the datastores
$ddDTO.VMwareDeploymentDestination.datastores = @($ds1)

#Retrieve the network
$networkCollection = Get-ManagedObject -name "Irina VM Network" -type "NETWORK"
$network = $networkCollection.ManagedObjectCollection.managedObjects[0]

#Retrieve standard switches
$panoramixHostSwitches = Get-HostStandardSwitches -hostId $panoramixHost.id

#Just to check the switch exists
$panoramixHostSwitch0 =
$panoramixHostSwitches.ManagedObjectReferenceCollection.ManagedObjectReferences |
where-Object {$_.displayName -match "vSwitch0"}

#Create fenced network config
#Note: The external network must be configured with an IP Pool
$fencedNetworkConfig = New-DTOTemplateObject -DTOTagName "FencedNetworkConfig"
$fencedNetworkConfig.FencedNetworkConfig.externalNetwork = $network
$fencedNetworkConfig.FencedNetworkConfig.vlanIds = "1"

#remove the distributed switches
$fencedNetworkConfig.FencedNetworkConfig.PSObject.Properties.Remove("distributedSwitch
")

#add standard switches
$switchBindingDTO = New-DTOTemplateObject -DTOTagName "SwitchBinding"
$switchBindingDTO.SwitchBinding.host = $panoramixHost
$switchBindingDTO.SwitchBinding.switchName = $panoramixHostSwitch0.displayName

$standardSwitchBindings = @($switchBindingDTO.SwitchBinding)
Add-Member -InputObject $fencedNetworkConfig.FencedNetworkConfig -MemberType
NoteProperty -Name "standardSwitches" -Value $standardSwitchBindings -Force

#Add the network fencing module
Add-Member -InputObject $ddDTO.VMwareDeploymentDestination -MemberType NoteProperty -
Name "fencedNetworkConfig" -Value $fencedNetworkConfig.FencedNetworkConfig -Force

#Create the deployment destination
$createdDD = New-VMwareDeploymentDestination -ddDTO $ddDTO

```

## Example 12: Create a deployment destination for SCVMM

This section contains several examples showing how to create an SCVMM deployment destination.

In the first example, a globally available deployment destination is created on an SCVMM managed system. VMs are deployed to a cluster and are connected to a specified network.

```

#Retrieve management server
$mobjectCollection = Get-ManagedObjectByName -name "Blaze" -type "MANAGEMENTSERVER"
$managementServer = $mobjectCollection.ManagedObjectCollection.managedObjects[0]

#Retrieve the target (cluster)
$cluster = Get-ClusterByName -msId $managementServer.id -clusterName
"FuegoCluster.embotics.com"

# Logical network
$logicaNetworkCollection1 = Get-ManagedObject -name "PVNet20 Logical Network" -type
"NETWORK"
$logicaNetworkCollection2 = Get-ManagedObject -name "PVNet21 Logical Network" -type
"NETWORK"

```

```

$logicalNetwork1 =
$logicalNetworkCollection1.ManagedObjectCollection.managedObjects[0]
$logicalNetwork2 =
$logicalNetworkCollection2.ManagedObjectCollection.managedObjects[0]

#Retrieve the datastores
$dsCollection1 = Get-ManagedObject -name "Jazz.embotics.com\D:\\" -type "DATASTORE"
$ds1 = $dsCollection1.ManagedObjectCollection.managedObjects[0]

#Create a SCVMM deployment destination DTO
$ddDTO = New-DTOTemplateObject -DTOTagName "SCVMMDeploymentDestination"
$ddDTO.SCVMMDeploymentDestination.name = "Default #1"
$ddDTO.SCVMMDeploymentDestination.diskFormat = "FIXED" #possible values are FIXED,
DYNAMICALLY_EXPANDING, and SAME_AS_SOURCE. See DiskFormat DTO for options.
$ddDTO.SCVMMDeploymentDestination.peakCapacity = $true #to use average capacity, set
to $false"

#Specify the management server
$ddDTO.SCVMMDeploymentDestination.managementServer = $managementServer

#Specify the target
$ddDTO.SCVMMDeploymentDestination.target = $cluster.Cluster

#Specify the network
$ddDTO.SCVMMDeploymentDestination.PSObject.Properties.Remove('network')
$ddDTO.SCVMMDeploymentDestination.PSObject.Properties.Remove('logicalNetwork')
Add-Member -InputObject $ddDTO.SCVMMDeploymentDestination -MemberType NoteProperty -
Name "logicalNetworks" -Value @($logicalNetwork1, $logicalNetwork2) -Force

#Specify the datastore
$ddDTO.SCVMMDeploymentDestination.datastores = @($ds1)

#Create the deployment destination
$createDD = New-SCVMMDeploymentDestination -ddDTO $ddDTO

```

In this example, an SCVMM deployment destination is assigned to a specific organization and to specific users. VMs are deployed to a specified host and are connected to the same network as the source image.

```

#Retrieve management server
$moObjectCollection = Get-ManagedObjectByName -name "Blaze" -type "MANAGEMENTSERVER"
$managementServer = $moObjectCollection.ManagedObjectCollection.managedObjects[0]

#Retrieve the target
$hostCollection = Get-ManagedObject -name "Jazz.embotics.com" -type "RUNTIMESERVER"
$targetHost = $hostCollection.ManagedObjectCollection.managedObjects[0]

#Retrieve the datastores
$dsCollection1 = Get-ManagedObject -name "Jazz.embotics.com\D:\\" -type "DATASTORE"
$ds1 = $dsCollection1.ManagedObjectCollection.managedObjects[0]

#Create a SCVMM deployment destination DTO
$ddDTO = New-DTOTemplateObject -DTOTagName "SCVMMDeploymentDestination"
$ddDTO.SCVMMDeploymentDestination.name = "Default #2"
$ddDTO.SCVMMDeploymentDestination.diskFormat = "SAME_AS_SOURCE"
$ddDTO.SCVMMDeploymentDestination.peakCapacity = $false

#Specify the management server
$ddDTO.SCVMMDeploymentDestination.managementServer = $managementServer

#Specify the target
$ddDTO.SCVMMDeploymentDestination.target = $targetHost

#Clear the network
$ddDTO.SCVMMDeploymentDestination.PSObject.Properties.Remove("network")
$ddDTO.SCVMMDeploymentDestination.PSObject.Properties.Remove("logicalNetwork")

#Specify the datastore

```

```

$ddDTO.SCVMMDeploymentDestination.datastores = @($ds1)

#Assigning user(s)
#Create a user; the only information we need is the loginId
$userDTO = New-DTOTemplateObject -DTOTagName "OwnerInfo"
$userDTO.OwnerInfo.id = -1
$userDTO.OwnerInfo.loginId = "superuser"
$userDTO.OwnerInfo.itContact = $false
$userDTO.OwnerInfo.primary = $false
$userDTO.OwnerInfo.email = $null
$userDTO.OwnerInfo.displayName = $null

$userDTO2 = New-DTOTemplateObject -DTOTagName "OwnerInfo"
$userDTO2.OwnerInfo.id = -1
$userDTO2.OwnerInfo.loginId = "manager"
$userDTO2.OwnerInfo.itContact = $false
$userDTO2.OwnerInfo.primary = $false
$userDTO2.OwnerInfo.email = $null
$userDTO2.OwnerInfo.displayName = $null

$users = @($userDTO.OwnerInfo, $userDTO2.OwnerInfo)
Add-Member -InputObject $ddDTO.SCVMMDeploymentDestination -MemberType NoteProperty -
Name "users" -Value $users -Force

#Assigning organization(s)
#Grab the organization
$org = Get-OrganizationByName -name "Default Organization"

$orgs = @($org.Organization)
Add-Member -InputObject $ddDTO.SCVMMDeploymentDestination -MemberType NoteProperty -
Name "organizations" -Value $orgs -Force

#Create the deployment destination
$createDD = New-SCVMMDeploymentDestination -ddDTO $ddDTO

```

## Example 13: Create a deployment destination for Amazon EC2

This section contains several examples showing how to create an Amazon EC2 deployment destination.

In the first example, an EC2 deployment destination is assigned to a specific organization and to specific users. VMs are deployed to a virtual cloud.

```

#Retrieve management server
$objectCollection = Get-ManagedObjectByName -name "it@embotics.com" -type
"MANAGEMENTSERVER"
$managementServer = $objectCollection.ManagedObjectCollection.managedObjects[0]

#Retrieve the target
$regionCollection = Get-ManagedObject -name "us-west-2" -type "REGION"
$targetRegion = $regionCollection.ManagedObjectCollection.managedObjects[0]
$virtualCloud = $targetRegion.VirtualClouds | where-Object {$_.displayName -match
"vpc-3b793253"}

#Subnet
$subnets = Get-VirtualCloudSubnets -regionId $targetRegion.id -vcid $virtualCloud.id
$subnet = $subnets.ManagedObjectReferenceCollection.ManagedObjectReferences | where-
Object {$_.displayName -match "192.168.100.0/24"}

#Security group
$securityGroups = Get-VirtualCloudSecurityGroups -regionId $targetRegion.id -vcid
$virtualCloud.id
$securityGroup =
$securityGroups.ManagedObjectReferenceCollection.ManagedObjectReferences | where-
Object {$_.displayName -match "default"}

#Create a deployment destination DTO
$ddDTO = New-DTOTemplateObject -DTOTagName "AWSDeploymentDestination"
$ddDTO.AWSDeploymentDestination.name="Default #3"

```

```

$ddDTO.AWSDeploymentDestination.keyPair="Embotics"

#Specify the management server
$ddDTO.AWSDeploymentDestination.managementServer = $managementServer

#Specify the target
$ddDTO.AWSDeploymentDestination.target = $virtualCloud

#Add subnet
Add-Member -InputObject $ddDTO.AWSDeploymentDestination -MemberType NoteProperty -Name
"subnet" -Value $subnet -Force

#Setup security group
$ddDTO.AWSDeploymentDestination.securityGroups = @($securityGroup)

#Assigning user(s)
#Create a user; the only information we need is the loginId
$userDTO = New-DTOTemplateObject -DTOTagName "OwnerInfo"
$userDTO.OwnerInfo.id = -1
$userDTO.OwnerInfo.loginId = "superuser"
$userDTO.OwnerInfo.itContact = $false
$userDTO.OwnerInfo.primary = $false
$userDTO.OwnerInfo.email = $null
$userDTO.OwnerInfo.displayName = $null

$userDTO2 = New-DTOTemplateObject -DTOTagName "OwnerInfo"
$userDTO2.OwnerInfo.id = -1
$userDTO2.OwnerInfo.loginId = "manager"
$userDTO2.OwnerInfo.itContact = $false
$userDTO2.OwnerInfo.primary = $false
$userDTO2.OwnerInfo.email = $null
$userDTO2.OwnerInfo.displayName = $null

$users = @($userDTO.OwnerInfo, $userDTO2.OwnerInfo)
Add-Member -InputObject $ddDTO.AWSDeploymentDestination -MemberType NoteProperty -Name
"users" -Value $users -Force

#Assigning organization(s)
#Grab the organization
$org = Get-OrganizationByName -name "Default Organization"

$orgs = @($org.Organization)
Add-Member -InputObject $ddDTO.AWSDeploymentDestination -MemberType NoteProperty -Name
"organizations" -Value $orgs -Force

#Create the deployment destination
$createdDD = New-AWSDeploymentDestination -ddDTO $ddDTO

```

## Example 14: Create a Compliance Policy

This example shows how to create a Compliance policy that targets a datacenter. In case of non-compliance, a workflow is executed.

```

#Retrieve management server
$dcCollection = Get-ManagedObjectByName -name "EmptyDC" -type "DATACENTER"
$dc = $dcCollection.ManagedObjectCollection.managedObjects[0]

$dto = New-DTOTemplateObject -DTOTagName "CompliancePolicy"
$dto.CompliancePolicy.name = "Policy #7"
$dto.CompliancePolicy.description = "Policy #7 description"
$dto.CompliancePolicy.enabled = $true
$dto.CompliancePolicy.alertViaSNMP = $true
$dto.CompliancePolicy.allowsOverride = $true
$dto.CompliancePolicy.generateVCenterAlerts = $true
$dto.CompliancePolicy.gracePeriodInHrs = 7 # 1-24 hrs, or multiple of 24 hrs up to 168
hrs (7 days)
$dto.CompliancePolicy.treeViewType = "OPERATIONAL" #Or VMS_AND_TEMPLATES

```



```

#Specify target
$dto.CompliancePolicy.targets =@($dc)

#See New-EndOfLifePolicy API for example(s) on how to create policy workflow action
$dto.CompliancePolicy.action.type = "NOTIFY_ONLY"

#Retrieve a number of attributes
$costCenterManagedObjectCollection = Get-ManagedObjectByName -type "ATTRIBUTE" -name
"Cost Center"
$costCenterReference =
$costCenterManagedObjectCollection.ManagedObjectCollection.managedObjects[0]

$slaManagedObjectCollection = Get-ManagedObjectByName -type "ATTRIBUTE" -name "SLA"
$slaReference = $slaManagedObjectCollection.ManagedObjectCollection.managedObjects[0]

#Set up a compliance requirement
$complianceRequirementDTO = New-DTOTemplateObject -DTOTagName "ComplianceRequirement"
$complianceRequirementDTO.ComplianceRequirement.expiryDateRequired = $true
$complianceRequirementDTO.ComplianceRequirement.primaryOwnerRequired = $false
$complianceRequirementDTO.ComplianceRequirement.attributes =
@($costCenterReference,$slaReference)
$dto.CompliancePolicy.complianceRequirement =
$complianceRequirementDTO.ComplianceRequirement

$createdPolicy = New-CompliancePolicy -dto $dto

```

## Example 15: Create a Default Attributes Policy

This section provides examples for creating a Default Attributes policy.

The first example shows how to create a Default Attributes policy that targets a datacenter. In case of non-compliance, a workflow is executed.

```

#Retrieve management server
$dcCollection = Get-ManagedObjectByName -name "EmptyDC" -type "DATACENTER"
$dc = $dcCollection.ManagedObjectCollection.managedObjects[0]

$dto = New-DTOTemplateObject -DTOTagName "DefaultAttributesPolicy"
$dto.DefaultAttributesPolicy.name = "Policy #6"
$dto.DefaultAttributesPolicy.description = "Policy #6 description"
$dto.DefaultAttributesPolicy.enabled = $true
$dto.DefaultAttributesPolicy.allowsOverride = $false

#Set to -1 for never expires
$dto.DefaultAttributesPolicy.daysToExpire = 25

#Specify target
$dto.DefaultAttributesPolicy.targets =@($dc)

#See New-EndOfLifePolicy API for example(s) on how to create policy workflow action
$dto.DefaultAttributesPolicy.action.type = "NOTIFY_ONLY"

#Set up groups
$scanGroup = Get-GroupByName -groupType "VM_SCAN_GROUP" -name "Default Guest OS Scan
Group"
$powerScheduleGroup = Get-GroupByName -groupType "POWER_SCHEDULE_GROUP" -name "Default
Power Schedule Group"
$expiryGroup = Get-GroupByName -groupType "EXPIRY_GROUP" -name "Default Expiry Group"

$dto.DefaultAttributesPolicy.scanGroup = $scanGroup.Group
$dto.DefaultAttributesPolicy.powerScheduleGroup = $powerScheduleGroup.Group
$dto.DefaultAttributesPolicy.expiryGroup = $expiryGroup.Group

$createdPolicy = New-AttributesPolicy -dto $dto

```

This example shows how to create a Default Attributes policy that targets a datacenter. In case of non-compliance, a workflow is executed. Compliance checks are included for Guest OS Scan group membership, Power Schedule group membership, and Expiry group membership.

```
#Retrieve management server
$dcCollection = Get-ManagedObjectByName -name "EmptyDC" -type "DATACENTER"
$dc = $dcCollection.ManagedObjectCollection.managedObjects[0]

$dto = New-DTOTemplateObject -DTOTagName "DefaultAttributesPolicy"
$dto.DefaultAttributesPolicy.name = "Policy #6"
$dto.DefaultAttributesPolicy.description = "Policy #6 description"
$dto.DefaultAttributesPolicy.enabled = $true
$dto.DefaultAttributesPolicy.allowsOverride = $false

#Set to -1 for never expires
$dto.DefaultAttributesPolicy.daysToExpire = 25

#Specify target
$dto.DefaultAttributesPolicy.targets = @($dc)

#See New-EndOfLifePolicy API for example(s) on how to create policy workflow action
$dto.DefaultAttributesPolicy.action.type = "NOTIFY_ONLY"

#Using defaults
$dto.DefaultAttributesPolicy.PSObject.Properties.Remove("scanGroup")
$dto.DefaultAttributesPolicy.PSObject.Properties.Remove("powerScheduleGroup")
$dto.DefaultAttributesPolicy.PSObject.Properties.Remove("expiryGroup")

$createdPolicy = New-AttributesPolicy -dto $dto
```

## Example 16: Create an Expired VM Policy

This section provides examples for creating an Expired VM policy.

This example shows how to create an Expired VM policy that targets a datacenter.

```
#Retrieve management server
$dcCollection = Get-ManagedObjectByName -name "EmptyDC" -type "DATACENTER"
$dc = $dcCollection.ManagedObjectCollection.managedObjects[0]

$dto = New-DTOTemplateObject -DTOTagName "ExpiryPolicy"
$dto.ExpiryPolicy.name = "Policy #5"
$dto.ExpiryPolicy.description = "Policy #5 description"
$dto.ExpiryPolicy.enabled = $true
$dto.ExpiryPolicy.targets = @($dc)

#Set action - soon to expire
$soonToExpireAction = $dto.ExpiryPolicy.intervalActions | where-Object {$_.interval -
match "SOON_TO_EXPIRE"}
$soonToExpireAction.action.type = "NOTIFY_ONLY"

#Set action - expired
$expiredAction = $dto.ExpiryPolicy.intervalActions | where-Object {$_.interval -eq
"EXPIRED"}
$expiredAction.action.type = "SET_EOL"

#Set action - post expired
#Find the correct workflow definition to use
$workflowDefinitions = Get-WorkflowDefinitionsByType -workflowType "VM"
$workflow = $workflowDefinitions.workflowDefinitionCollection.workflowDefinitions |
where-Object {$_.name -match "Command workflow #1"}

#Create a policy workflow action
#Use the workflow definition we just pulled
$policyworkflowActionObject = New-DTOTemplateObject -DTOTagName "PolicyworkflowAction"
$policyworkflowActionObject.PolicyworkflowAction.workflowDefinition = $workflow
```



```

$postExpiredAction = $dto.ExpiryPolicy.intervalActions | where-Object {$_.interval -eq
"POST_EXPIRED"}
$postExpiredAction.action = $policyworkflowActionObject.PolicyworkflowAction

#Set expiry groups
$expiryGroup = Get-GroupByName -groupType "EXPIRY_GROUP" -name "Default Expiry Group"
$dto.ExpiryPolicy.expiryGroup = $expiryGroup.Group
$dto.ExpiryPolicy.preExpiryLength = 8
$dto.ExpiryPolicy.postExpiryLength = 31

#Expiry extension
$dto.ExpiryPolicy.expiryExtension.extensionDays = 27
$dto.ExpiryPolicy.expiryExtension.vmStateToUnapproved = $false

#Expiry notifications
$dto.ExpiryPolicy.expiryNotifications.alertViaSNMP = $true
$dto.ExpiryPolicy.expiryNotifications.emailITContact = $true
$dto.ExpiryPolicy.expiryNotifications.emailPrimaryOwner = $true
$dto.ExpiryPolicy.expiryNotifications.emailAllOwners = $false
$dto.ExpiryPolicy.expiryNotifications.emailSubject = "Email subject"
$dto.ExpiryPolicy.expiryNotifications.emailBody = "Email body"

$createdPolicy = New-ExpiryPolicy -dto $dto

```

The next example shows how to create an expired VM policy that targets a datacenter; expiry extension and notification options are disabled.

```

#Retrieve management server
$dcCollection = Get-ManagedObjectByName -name "EmptyDC" -type "DATACENTER"
$dc = $dcCollection.ManagedObjectCollection.managedObjects[0]

$dto = New-DTOTemplateObject -DTOTagName "ExpiryPolicy"
$dto.ExpiryPolicy.name = "Policy #5"
$dto.ExpiryPolicy.description = "Policy #5 description"
$dto.ExpiryPolicy.enabled = $true
$dto.ExpiryPolicy.targets = @($dc)

#Set action - soon to expire
$soonToExpireAction = $dto.ExpiryPolicy.intervalActions | where-Object {$_.interval -
match "SOON_TO_EXPIRE"}
$soonToExpireAction.action.type = "NOTIFY_ONLY"

#Set action - expired
$expiredAction = $dto.ExpiryPolicy.intervalActions | where-Object {$_.interval -eq
"EXPIRED"}
$expiredAction.action.type = "SET_EOL"

#Set action - post expired
#Find the correct workflow definition to use
$workflowDefinitions = Get-WorkflowDefinitionsByType -workflowType "VM"
$workflow = $workflowDefinitions.workflowDefinitionCollection.workflowDefinitions |
where-Object {$_.name -match "Command workflow #1"}

#Create a policy workflow action
#Use the workflow definition we just pulled
$policyworkflowActionObject = New-DTOTemplateObject -DTOTagName "PolicyworkflowAction"
$policyworkflowActionObject.PolicyworkflowAction.workflowDefinition = $workflow

$postExpiredAction = $dto.ExpiryPolicy.intervalActions | where-Object {$_.interval -eq
"POST_EXPIRED"}
$postExpiredAction.action = $policyworkflowActionObject.PolicyworkflowAction

#Set expiry groups
$expiryGroup = Get-GroupByName -groupType "EXPIRY_GROUP" -name "Default Expiry Group"
$dto.ExpiryPolicy.expiryGroup = $expiryGroup.Group
$dto.ExpiryPolicy.preExpiryLength = 8
$dto.ExpiryPolicy.postExpiryLength = 31

#Disable expiry extension
$dto.ExpiryPolicy.PSObject.Properties.Remove("expiryExtension")

```

```
#Disable expiry notifications
$dto.ExpiryPolicy.PSObject.Properties.Remove("expiryNotifications")

$createdPolicy = New-ExpiryPolicy -dto $dto
```

## Example 17: Create a Default Ownership Policy

This section provides examples for creating a Default Ownership policy.

The first example shows how to create a default ownership policy that targets a datacenter. In case of non-compliance, a workflow is executed.

```
#Retrieve datacenter
$dcCollection = Get-ManagedObjectByName -name "EmptyDC" -type "DATACENTER"
$dc = $dcCollection.ManagedObjectCollection.managedObjects[0]

#Find the correct workflow definition to use
$workflowDefinitions = Get-WorkflowDefinitionsByType -workflowType "VM"
$workflow = $workflowDefinitions.WorkflowDefinitionCollection.WorkflowDefinitions |
Where-Object {$_.name -match "Command workflow #1"}

#Create a policy workflow action
#Use the workflow definition we just pulled
$policyworkflowActionObject = New-DTOTemplateObject -DTOTagName "PolicyworkflowAction"
$policyworkflowActionObject.PolicyworkflowAction.workflowDefinition = $workflow

$dto = New-DTOTemplateObject -DTOTagName "DefaultOwnershipPolicy"
$dto.DefaultOwnershipPolicy.name = "Policy #4"
$dto.DefaultOwnershipPolicy.description = "Policy #4 description"
$dto.DefaultOwnershipPolicy.enabled = $true
$dto.DefaultOwnershipPolicy.treeViewType = "OPERATIONAL" #Or VMS_AND_TEMPLATES
$dto.DefaultOwnershipPolicy.allowsOverride = $false

#Override simple policy action with a workflow action
$dto.DefaultOwnershipPolicy.action = $policyworkflowActionObject.PolicyworkflowAction

#Specify target
$dto.DefaultOwnershipPolicy.targets = @($dc)

#Create a user; the only information we need is the loginId
$user1DTO = New-DTOTemplateObject -DTOTagName "OwnerInfo"
$user1DTO.OwnerInfo.loginId = "superuser"
$user1DTO.OwnerInfo.itContact = $false
$user1DTO.OwnerInfo.primary = $true

#Create a user; the only information we need is the loginId
$user2DTO = New-DTOTemplateObject -DTOTagName "OwnerInfo"
$user2DTO.OwnerInfo.loginId = "manager"
$user2DTO.OwnerInfo.itContact = $true
$user2DTO.OwnerInfo.primary = $false

#Create a user; the only information we need is the loginId
$user3DTO = New-DTOTemplateObject -DTOTagName "OwnerInfo"
$user3DTO.OwnerInfo.loginId = "bob@embotecs.com"
$user3DTO.OwnerInfo.itContact = $false
$user3DTO.OwnerInfo.primary = $false

#Set the owners
$dto.DefaultOwnershipPolicy.owners = @($user1DTO.OwnerInfo, $user2DTO.OwnerInfo,
$user3DTO.OwnerInfo)

#Grab the organization
$org = Get-OrganizationByName -name "Default Organization"
$dto.DefaultOwnershipPolicy.organization = $org.Organization

$createdPolicy = New-OwnershipPolicy -dto $dto
```

This example shows how to create a default ownership that targets a folder using the VMs and Templates view. Notify Only is selected as the resulting action for non-compliance.

```
#Retrieve folder
$folderCollections = Get-FoldersByName -name "VMDeploymentFolder"
$folder = $folderCollections.FolderCollection.Folders[0]

$dto = New-DTOTemplateObject -DTOTagName "DefaultOwnershipPolicy"
$dto.DefaultOwnershipPolicy.name = "Policy #4"
$dto.DefaultOwnershipPolicy.description = "Policy #4 description"
$dto.DefaultOwnershipPolicy.enabled = $true
$dto.DefaultOwnershipPolicy.treeViewType = "VMS_AND_TEMPLATES" #Or OPERATIONAL
$dto.DefaultOwnershipPolicy.allowsOverride = $false
$dto.DefaultOwnershipPolicy.action.type = "NOTIFY_ONLY"

#Specify target
$dto.DefaultOwnershipPolicy.targets = @($folder)

#Create a user; the only information we need is the loginId
$user1DTO = New-DTOTemplateObject -DTOTagName "OwnerInfo"
$user1DTO.OwnerInfo.loginId = "superuser"
$user1DTO.OwnerInfo.itContact = $false
$user1DTO.OwnerInfo.primary = $true

#Create a user; the only information we need is the loginId
$user2DTO = New-DTOTemplateObject -DTOTagName "OwnerInfo"
$user2DTO.OwnerInfo.loginId = "manager"
$user2DTO.OwnerInfo.itContact = $true
$user2DTO.OwnerInfo.primary = $false

#Create a user; the only information we need is the loginId
$user3DTO = New-DTOTemplateObject -DTOTagName "OwnerInfo"
$user3DTO.OwnerInfo.loginId = "bob@embotics.com"
$user3DTO.OwnerInfo.itContact = $false
$user3DTO.OwnerInfo.primary = $false

#Set the owners
$dto.DefaultOwnershipPolicy.owners = @($user1DTO.OwnerInfo, $user2DTO.OwnerInfo,
$user3DTO.OwnerInfo)

#Grab the organization
$org = Get-OrganizationByName -name "Default Organization"
$dto.DefaultOwnershipPolicy.organization = $org.Organization

$createdPolicy = New-OwnershipPolicy -dto $dto
```

## Example 18: Create a cost model

This example shows how to create a cost model which sets the CPU, memory and storage costs.

```
$costModel = New-DTOTemplateObject -DTOTagName "CostModel"
$costModel.CostModel.name = "Cost Model #1"

# Retrieve the target
Add-Member -InputObject $costModel.CostModel -MemberType NoteProperty -Name "targets"
-Value "PowerShell Rest client" -Force

$mobjectCollection
= Get-ManagedObjectByName -name "manta" -type "MANAGEMENTSERVER"

$managementServer =
$mobjectCollection.ManagedObjectCollection.ManagedObjectReferences[0]

$costModel.CostModel.targets = @( $managementServer )

# Valid values for view: OPERATIONAL | VMS_AND_TEMPLATES
$costModel.CostModel.view = "OPERATIONAL"

# Resource Costs
```

```

$costModel.CostModel.reservedMemoryCost = 500.00
$costModel.CostModel.allocatedMemoryCost = 400.00
$costModel.CostModel.reservedCpuCost = 300.00
$costModel.CostModel.allocatedCpuCost = 200.00

# Storage Costs
$tier1 = New-DTOTemplateObject -DTOTagName "StorageTierCost"
$tier1.StorageTierCost.name = "Storage Tier 1"
$tier1.StorageTierCost.cost = 100.00

$tier2 = New-DTOTemplateObject -DTOTagName "StorageTierCost"
$tier2.StorageTierCost.name = "Storage Tier 2"
$tier2.StorageTierCost.cost = 80.00

$tier3 = New-DTOTemplateObject -DTOTagName "StorageTierCost"
$tier3.StorageTierCost.name = "Storage Tier 3"
$tier3.StorageTierCost.cost = 60.00

$costModel.CostModel.defaultStorageTier = "Storage Tier 2"

$costModel.CostModel.storageTierCosts = @( $tier1.StorageTierCost,
$tier2.StorageTierCost, $tier3.StorageTierCost )

# Valid values for storageCostCalculation: ACTUAL_SIZE | PROVISIONED_SIZE
$costModel.CostModel.storageCostCalculation = "ACTUAL_SIZE"

# Create the Cost Model
$taskId = New-CostModel -costModelDto $costModel

```

## Example 19: Create an approval workflow

This example shows how to create an approval workflow definition which has an email step and approver step.

```

#Create a DTO
$approvalWorkflowDTO = New-DTOTemplateObject -DTOTagName
"NewApprovalWorkflowDefinition"

#Set appropriate properties
$approvalWorkflowDTO.NewApprovalWorkflowDefinition.name = "My New Request Approval"
$approvalWorkflowDTO.NewApprovalWorkflowDefinition.global = $false
$approvalWorkflowDTO.NewApprovalWorkflowDefinition.addOwnerAsAdmin = $true
$approvalWorkflowDTO.NewApprovalWorkflowDefinition.autoDeploy = $true

#Create an email step
$emailStepDefinitionDTO = New-DTOTemplateObject -DTOTagName
"workflowEmailStepDefinition"
$emailStepDefinitionDTO.workflowEmailStepDefinition.name="Send Email To ABC"
$emailStepDefinitionDTO.workflowEmailStepDefinition.addresses="bob@embotics.com;sally@
embotics.com"
$emailStepDefinitionDTO.workflowEmailStepDefinition.emailBody = "This is email body"
$emailStepDefinitionDTO.workflowEmailStepDefinition.emailSubject = "This is email
subject"
$emailStepDefinitionDTO.workflowEmailStepDefinition.includeRequestDetails = $true
$emailStepDefinitionDTO.workflowEmailStepDefinition.stepNumber = 0

#Create an approver step
$approverStepDefinitionDTO = New-DTOTemplateObject -DTOTagName
"workflowApproverStepDefinition"
$approverStepDefinitionDTO.workflowApproverStepDefinition.name="To Manager for
approval"
$approverStepDefinitionDTO.workflowApproverStepDefinition.addresses="managers@embotics
.com"
$approverStepDefinitionDTO.workflowApproverStepDefinition.emailBody = "This is the
content of the approver email body"
$approverStepDefinitionDTO.workflowApproverStepDefinition.emailSubject = "This is the
subject of the approver email"
$approverStepDefinitionDTO.workflowApproverStepDefinition.stepNumber = 1

```

```

#Clear the existing step(s); add new ones we just created
$approvalworkflowDTO.NewApprovalWorkflowDefinition.Steps = @()
$approvalworkflowDTO.NewApprovalWorkflowDefinition.Steps +=
$emailStepDefinitionDTO.WorkflowEmailStepDefinition
$approvalworkflowDTO.NewApprovalWorkflowDefinition.Steps +=
$approverStepDefinitionDTO.WorkflowApproverStepDefinition

#Retrieve the organization by name [This will fail if you don't have an organization
with the name 'Org 1']
$org = Get-OrganizationByName -name "Org 1"

#Create a reference to this organization. We really just need the id and type.
$orgReferencedDTO = New-DTOTemplateObject -DTOTagName "ManagedObjectReference"
$orgReferencedDTO.ManagedObjectReference.id = $org.Organization.id
$orgReferencedDTO.ManagedObjectReference.type = "ORGANIZATION"

#Clear out organization(s); add new one we just created
$approvalworkflowDTO.NewApprovalWorkflowDefinition.Organizations = @()
$approvalworkflowDTO.NewApprovalWorkflowDefinition.Organizations +=
$orgReferencedDTO.ManagedObjectReference

#Create a user; the only information we need is the loginId
$userDTO = New-DTOTemplateObject -DTOTagName "OwnerInfo"
$userDTO.OwnerInfo.id = -1
$userDTO.OwnerInfo.loginId = "superuser"
$userDTO.OwnerInfo.itContact = $false
$userDTO.OwnerInfo.primary = $false
$userDTO.OwnerInfo.email = $null
$userDTO.OwnerInfo.displayName = $null

#Clear out user(s); add new one we just created
$approvalworkflowDTO.NewApprovalWorkflowDefinition.Users = @()
$approvalworkflowDTO.NewApprovalWorkflowDefinition.Users += $userDTO.OwnerInfo

#Submit the DTO for creation
$createdDefinition = New-WorkflowDefinition -definitionDto $approvalworkflowDTO

```

## Example 20: Create a completion workflow

This example shows how to create a completion workflow definition which has a Set Custom Attribute step and an Execute SSH Command step.

```

#Create a DTO
$vmCompletionworkflowDTO = New-DTOTemplateObject -DTOTagName
"ComponentCompletionWorkflowDefinition"

#Set appropriate properties
$vmCompletionworkflowDTO.ComponentCompletionWorkflowDefinition.name = "My VM
Completion workflow"
$vmCompletionworkflowDTO.ComponentCompletionWorkflowDefinition.global = $false

#Retrieve a publish service and get one of its components
#This assumes that you have a published service with the name 'AutomatedTest_SC_VM'
$ps = Get-PublishedServiceByName -name "AutomatedTest_SC_VM"
$psComponentId = $ps.PublishedService.serviceComponents[0].id

#Create an object to represent this component
$objectRepresentationDTO = New-DTOTemplateObject -DTOTagName "ObjectRepresentation"
$objectRepresentationDTO.ObjectRepresentation.originalObjectId = $psComponentId

#Clear out all the component representations; add the one we just created
$vmCompletionworkflowDTO.ComponentCompletionWorkflowDefinition.ComponentRepresentation
s = @()
$vmCompletionworkflowDTO.ComponentCompletionWorkflowDefinition.ComponentRepresentation
s += $objectRepresentationDTO.ObjectRepresentation

#Create a custom attribute DTO
$attributeDTO = New-DTOTemplateObject -DTOTagName "CustomAttribute"
$attributeDTO.CustomAttribute.allowedValues = @() #not important

```



```

$attributeDTO.CustomAttribute.description = $null #not important
$attributeDTO.CustomAttribute.targetManagedObjectTypes = @() #not important
$attributeDTO.CustomAttribute.name= "SLA"
$attributeDTO.CustomAttribute.value = "Gold"

#Create a custom attribute step
$caStepDefinitionDTO = New-DTOTemplateObject -DTOTagName
"workflowCustomAttributeStepDefinition"
$caStepDefinitionDTO.workflowCustomAttributeStepDefinition.name="Set Custom Attribute"
$caStepDefinitionDTO.workflowCustomAttributeStepDefinition.CustomAttributes = @()
$caStepDefinitionDTO.workflowCustomAttributeStepDefinition.CustomAttributes +=
$attributeDTO.CustomAttribute
$caStepDefinitionDTO.workflowCustomAttributeStepDefinition.stepNumber = 0

#Create a SSH command step definition
$sshCommandStepDefinitionDTO = New-DTOTemplateObject -DTOTagName
"workflowSSHCommandStepDefinition"
$sshCommandStepDefinitionDTO.workflowSSHCommandStepDefinition.name="Execute SSH
Command"
$sshCommandStepDefinitionDTO.workflowSSHCommandStepDefinition.timeoutInSeconds = 300
$sshCommandStepDefinitionDTO.workflowSSHCommandStepDefinition.proceedToNextStepOnFailu
re = $true
$sshCommandStepDefinitionDTO.workflowSSHCommandStepDefinition.captureOutput = $true
$sshCommandStepDefinitionDTO.workflowSSHCommandStepDefinition.commandLine = "ls"
$sshCommandStepDefinitionDTO.workflowSSHCommandStepDefinition.hostname = "example-
host"
$sshCommandStepDefinitionDTO.workflowSSHCommandStepDefinition.guestOSCredential.displa
yName = $null #doesn't matter
$sshCommandStepDefinitionDTO.workflowSSHCommandStepDefinition.guestOSCredential.id =
$guestOScredId

#Clear the existing step(s); add new ones we just created
$vmCompletionWorkflowDTO.ComponentCompletionWorkflowDefinition.Steps = @()
$vmCompletionWorkflowDTO.ComponentCompletionWorkflowDefinition.Steps +=
$caStepDefinitionDTO.workflowCustomAttributeStepDefinition
$vmCompletionWorkflowDTO.ComponentCompletionWorkflowDefinition.Steps +=
$sshCommandStepDefinitionDTO.workflowSSHCommandStepDefinition

#Submit the DTO for creation
$createdDefinition = New-WorkflowDefinition -definitionDto $vmCompletionWorkflowDTO

```

## Example 21: Create a command workflow

This example shows how to create a Command Workflow definition which has a Decommission Networking step.

```

#Create a DTO
$vmWorkflowDTO = New-DTOTemplateObject -DTOTagName "ServiceWorkflowDefinition"

#Set appropriate properties
$vmWorkflowDTO.ServiceWorkflowDefinition.name = "workflow with Decommission Networking
step"
$vmWorkflowDTO.ServiceWorkflowDefinition.global = $true
$vmWorkflowDTO.ServiceWorkflowDefinition.promptOnRun = $true
$vmWorkflowDTO.ServiceWorkflowDefinition.workflowAsCommand = $true

#Create a Decommission Networking step
$decommissionGuestNetworkDto = New-DTOTemplateObject -DTOTagName
"workflowDecommissionGuestNetworkStepDefinition"
$decommissionGuestNetworkDto.workflowDecommissionGuestNetworkStepDefinition.name =
"Decommission Networking"
$decommissionGuestNetworkDto.workflowDecommissionGuestNetworkStepDefinition.ipSource =
"BLUE_CAT"

#Clear the existing step(s); add new ones we just created
$vmWorkflowDTO.ServiceWorkflowDefinition.Steps = @()
$vmWorkflowDTO.ServiceWorkflowDefinition.Steps +=
$decommissionGuestNetworkDto.workflowDecommissionGuestNetworkStepDefinition

```

```
#Clear out organization(s); add new one we just created
$vmWorkflowDTO.ServiceWorkflowDefinition.Organizations = @()
#Clear out user(s); add new one we just created
$vmWorkflowDTO.ServiceWorkflowDefinition.Users = @()

#Submit the DTO for creation
$createdDefinition = New-WorkflowDefinition -definitionDto $vmWorkflowDTO
```

## Example 22: Connect a media file to a VM's CD-ROM drive

This example shows how to mount a media file to a VM's CD-ROM drive.

```
#get the VM
$vms = Get-VMByRemoteId -vmRemoteId 'vm-2566'
$vm = $vms.VirtualMachineCollection.VirtualMachines[0]

# find the CD-ROM
foreach ($disk in $vm.disks) {
    if ($disk.diskType -eq "CDROM") {
        $diskkey = $disk.key
        $mediaFileMountDto = New-DTOTemplateObject -DTOTagName "MediaFileMountSpec"
        $mediaFileMountDto.MediaFileMountSpec.mediaFilePath = "[DataStoreName]
Folder/SubFolder/mediaFile.iso"

        $taskInfo = Mount-MediaFile -vmid $vm.id -diskkey $diskkey -mediaFileDto
$mediaFileMountDto
        break
    }
}
```

## Example 23: Disconnect a media file from a VM's CD-ROM drive

This example shows how to dismount a media file from a VM's CD-ROM drive.

```
#get the VM
$vms = Get-VMByRemoteId -vmRemoteId 'vm-2566'
$vm = $vms.VirtualMachineCollection.VirtualMachines[0]

# find the CD-ROM
foreach ($disk in $vm.disks) {
    if ($disk.diskType -eq "CDROM") {
        $diskkey = $disk.key

        $taskInfo = Dismount-MediaFile -vmid $vm.id -diskkey $diskkey
        break
    }
}
```

## Example 24: Create a global media folder

This example shows how to create a media folder that is globally accessible.

```
#Create a media folder
$mfDto = New-DTOTemplateObject -DTOTagName "MediaFolder"

#Specify name and description
$mfDto.MediaFolder.name="GlobalMediaFolder"
$mfDto.MediaFolder.description="description example"

#Specify datastore and path
$moObjectCollection = Get-ManagedObjectByName -name "myDatastore" -type "DATASTORE"
$datastoreDto = $moObjectCollection.ManagedObjectCollection.managedObjects[0]
$mfDto.MediaFolder.datastore = $datastoreDto
$mfDto.MediaFolder.path = "ISOFolder"
```

```
#Specify that is global
$mfDto.MediaFolder.global = $true
#Clear organizations
$mfDto.MediaFolder.organizations = @()

$taskInfo = New-MediaFolder -mfDto $mfDto
```

## Example 25: Create an organization-specific media folder

This example shows how to create a media folder that is accessible only to an organization.

```
#Create a media folder
$mfDto = New-DTTemplateObject -DTOTagName "MediaFolder"

#Specify name and description
$mfDto.MediaFolder.name="Org1MediaFolder"
$mfDto.MediaFolder.description="description example"

#Specify datastore and path
$objectCollection = Get-ManagedObjectByName -name "myDatastore" -type "DATASTORE"
$datastoreDto = $objectCollection.ManagedObjectCollection.managedObjects[0]
$mfDto.MediaFolder.datastore = $datastoreDto
$mfDto.MediaFolder.path = "ISOFolderOrg1"

#Specify organization
$org1 = Get-OrganizationByName -name "Org1"

#Create a reference to this organization. We really just need the ID and type.
$mfDto.MediaFolder.global = $false
$mfDto.MediaFolder.organizations = @()
$mfDto.MediaFolder.organizations += $org1.Organization

$taskInfo = New-MediaFolder -mfDto $mfDto
```

## Example 26: Create relationships between custom attributes

This example shows how to create relationships between list custom attributes, so that the selection of a value for one attribute limits the allowable values for a second attribute.

```
#Create a parent list attribute, a child sublist attribute, and a grandchild sublist attribute

### List custom attribute that is the parent of the Cities sublist attribute

# Create a custom attribute DTO
$customAttribute = New-DTTemplateObject -DTOTagName "CustomAttribute"

# Specify the values for the list attribute properties
$customAttribute.CustomAttribute.name = "Provinces"
$customAttribute.CustomAttribute.description = "Canadian Provinces"
$customAttribute.CustomAttribute.id = -1
$customAttribute.CustomAttribute.targetManagedObjectTypes = @("VIRTUALMACHINE")
$customAttribute.CustomAttribute.allowedValues = @("Ontario", "Quebec", "Alberta")
$createdAttribute = New-CustomAttribute -customAttributeDTO $customAttribute

### Sublist attribute that is a child of the Provinces list attribute and parent of the Mayors sublist attribute

# Create a sublist attribute DTO
$customAttribute = New-DTTemplateObject -DTOTagName "SubListCustomAttribute"

# Specify the values for the sublist attribute properties
$customAttribute.SubListCustomAttribute.name = "Cities"
$customAttribute.SubListCustomAttribute.description = "Canadian Cities by Province"
$customAttribute.SubListCustomAttribute.id = -1
```



```
# The targetManagedObjectTypes property does not need to be specified, as the value is
inherited from the parent list attribute
$customAttribute.SubListCustomAttribute.sublistOfAttribute = "Provinces"
$customAttribute.SubListCustomAttribute.allowedValues = @{"Alberta" = @("Calgary",
"Edmonton"); "Ontario" = @("Aylmer", "Toronto", "Ottawa"); "Quebec" = @("Quebec City",
"Aylmer", "Montreal")}]
$createdAttribute = New-CustomAttribute -customAttributedTo $customAttribute

### Sublist attribute that is a child of the Cities sublist attribute

# Create a sublist attribute DTO
$customAttribute = New-DTOTemplateObject -DTOTagName "SubListCustomAttribute"

# Specify the values for the sublist attribute properties
$customAttribute.SubListCustomAttribute.name = "Mayors"
$customAttribute.SubListCustomAttribute.description = "Mayors of Canadian Cities"
$customAttribute.SubListCustomAttribute.id = -1

# The targetManagedObjectTypes property does not need to be specified, as the value is
inherited from the parent
list custom attribute
$customAttribute.SubListCustomAttribute.sublistOfAttribute = "Cities"

# Because the city of Aylmer exists in both Ontario and Quebec, the city name needs to
be prefixed by the province followed by a semicolon
$customAttribute.SubListCustomAttribute.allowedValues = @{"Calgary" = @("Naheed
Nenshi"); "Edmonton" = @("Don Iveson"); "Ontario;Aylmer" = @("Greg Currie"); "Toronto"
= @("John Tory"); "Ottawa" = @("Jim Watson"); "Quebec City" = @("Regis Labeaume");
"Montreal" = @("Denis Coderre"); "Quebec;Aylmer" = @("Marc Croteau")}]
$createdAttribute = New-CustomAttribute -customAttributedTo $customAttribute
```

## Example 27: Update a custom attribute that has a relationship with a sublist attribute

This example shows how to update list custom attributes that have sublist attributes.

```
# Update a parent list attribute, a child sublist attribute, and a grandchild sublist
attribute

### List attribute that is the parent of the Cities sublist attribute

# Get a custom attribute and add a new allowed value
$customAttribute = Get-CustomAttributeByName -name "Provinces"
$allowedValues = $customAttribute.CustomAttribute.allowedValues
$customAttribute.CustomAttribute.allowedValues = @()
$customAttribute.CustomAttribute.allowedValues += $allowedValues
$customAttribute.CustomAttribute.allowedValues += "Nova Scotia"

Update-CustomAttribute -id $customAttribute.CustomAttribute.id -customAttributedTo
$customAttribute

### Sublist attribute that is a child of the Provinces list attribute and parent of
the Mayors sublist attribute

# Get a sublist attribute and add new allowed values for a specific parent value
$customAttribute = Get-CustomAttributeByName -name "Cities"
$customAttribute.SubListCustomAttribute.allowedValues.Add("Nova Scotia", @("Halifax",
"Sydney"))

Update-CustomAttribute -id $customAttribute.SubListCustomAttribute.id -
customAttributedTo $customAttribute

### Sublist attribute that is a child of the Cities sublist attribute

# Get a custom attribute and add new allowed values for a specific parent value
$customAttribute = Get-CustomAttributeByName -name "Mayors"
$customAttribute.SubListCustomAttribute.allowedValues.Add("Halifax", "ha")
$customAttribute.SubListCustomAttribute.allowedValues.Add("Sydney", "sy")
```

```
Update-CustomAttribute -id $customAttribute.SublistCustomAttribute.id -
customAttributeDTO $customAttribute
```

## Example 28: Add a VM to the service catalog and configure the service

This example shows how to add a VM to the service catalog, assign a customization spec, assign it to an organization, configure the blueprint, and assign a completion workflow.

```
# Locate a VM by name (returns a list)
$vm = Get-VMReferences -vmName "Base Template for Puppet Deployments"
# Take the first VM in the list
$vm = $vm.ManagedObjectReferenceCollection.ManagedObjectReferences[0]

# Create a published service component for the VM
$psc = New-DTOTemplateObject -DTOTagName "PublishedServiceComponent"
$psc.PublishedServiceComponent.ref.id = $vm.id
$psc.PublishedServiceComponent.ref.type = $vm.type
$psc.PublishedServiceComponent.name = "My Component"
$psc.PublishedServiceComponent.description = "Component Description"
$psc.PublishedServiceComponent.linkedClone = $true

# Set the customization spec
$psc.PublishedServiceComponent.customizationSpecName = "Windows 2008r2"

# Create the new Published Service with the VM component
$ps = New-DTOTemplateObject -DTOTagName "PublishedService"
$ps.PublishedService.name = "My New Service"
$ps.PublishedService.description = "Service Description"
$ps.PublishedService.serviceComponents = @($psc.PublishedServiceComponent)

# Publish it to a specific organization
$ps.PublishedService.publishState = "PUBLISHED_SPECIFIC_USERS"
$ps.PublishedService.serviceUsers = @()
$org = Get-OrganizationByName -name "Development"
Add-Member -InputObject $ps.PublishedService -MemberType NoteProperty -Name
"organizations" -Value @($org.Organization)

# Use blueprint forms
$ps.PublishedService.useStaticComponentForms = $false

# Create the new service
$newService = New-PublishedService -psDto $ps

# Get the newly created component so we can assign it to a workflow
$newComponent = $newService.PublishedService.serviceComponents[0]

# Get the desired completion workflow
$wf = Get-WorkflowDefinition -name "Sample workflow" -type "NEW_VM_POST_DEPLOY"

# Link the workflow to the component
$update = Join-WorkflowDefinitionToServiceComponent -definitionId
$wf.ComponentCompletionWorkflowDefinition.id -componentId $newComponent.id
```

## Example 29: Request a service

This example shows how to request a service from the catalog.

```
# Locate the Published Service and retrieve the form elements
$ps = Get-PublishedServiceByName -name "Sample Service"
$requestParams = Get-PSRequestParams -psId $ps.PublishedService.id

# Populate the required elements on the service form
$element =
$requestParams.PSDeployParamCollection.serviceformElements.RequestFormElements |
Where-Object {$_.label -eq "Project Code"}
$element.value = "Project 123"
```

```

$element =
$requestParams.PSDeployParamCollection.serviceFormElements.RequestFormElements |
where-Object {$_.label -eq "Expiration Date"}
$element.value = "2016/04/07"

# Populate the required elements on the component form
$componentParams = $requestParams.PSDeployParamCollection.componentParams | where-
Object { $_.componentName -eq "Component One" }
$element = $componentParams.formElements.RequestFormElements | where-Object {$_.label -
eq "Build Revision"}
$element.value = "1111"

#Submit the request
$requestDTO = New-ServiceRequest -psId $ps.PublishedService.id -requestParams
$requestParams

```

### Example 30: Retrieve available quota for an organization or organization member

This example shows how to retrieve quota statistics, including used and available quota, for an organization and its members. Note that a Service Portal account must be used to retrieve quota information. To retrieve information for all members of an organization, the Service Portal role must have the Manage Organizations permission.

```

# Get references to all visible organizations
$org = Get-OrganizationReferences
foreach ($mor in $org.ManagedObjectReferenceCollection.ManagedObjectReferences) {
    write-host $mor.displayName

    $stats = Get-QuotaStatistics -orgId $mor.id

    # stats for a specific user
    $stats = Get-QuotaStatistics -orgId $mor.id -member "superuser"

    write-host "Quota stats for $($stats.QuotaStatistics.name)"

    if ($stats.QuotaStatistics.resourceQuota -ne $null) {
        write-host "Available Resource Quota"
        $stats.QuotaStatistics.resourceQuota | Format-List *
    }

    if ($stats.QuotaStatistics.costQuota -ne $null) {
        write-host "Available Cost Quota"
        $stats.QuotaStatistics.costQuota | Format-List *
    }
}

```

### Example 31: Run a command workflow

This example shows how to run a command workflow on a VM.

```

# Locate a VM by name
$vms = Get-VMReferences -vmName "My VM"
$vm = $vms.ManagedObjectReferenceCollection.ManagedObjectReferences[0]

# Locate a command workflow
$wf = Get-WorkflowDefinition -name "Send Email" -type "VM"

$taskInfo = Start-CommandWorkflow -vmId $vm.id -workflowDefinitionId
$wf.ServiceWorkflowDefinition.id

```

### Example 32: Share a VM

This example shows how to share a VM with an organization or with specific users.

```

#Define how the VM is shared
$shareOptions = New-DTOTemplateObject -DTOTagName "ShareOptions"
$shareOptions.ShareOptions.daysUntilExpired = 10
$shareOptions.ShareOptions.emailComments = "Please investigate automated tests failure."
$shareOptions.ShareOptions.emailSubject = "Automated tests failure"
$shareOptions.ShareOptions.keepSourceOwnership = $false
$shareOptions.ShareOptions.notifyRecipients = $true
$shareOptions.ShareOptions.serviceDescription = "VM containing installer and tests."
$shareOptions.ShareOptions.serviceName = "W2K8 Service"

#Make it visible to an organization or specific users

#Uncomment the next two lines to share with an organization
#$org = Get-OrganizationByName "Default Organization"
#$shareOptions.ShareOptions.sharedOrganization = $org.Organization

#Uncomment this line if organization visibility is not used
$shareOptions.ShareOptions.PSObject.Properties.Remove("sharedOrganization")

#Uncomment the next three lines to share with a specific user
$userDTO = New-DTOTemplateObject -DTOTagName "OwnerInfo"
$userDTO.OwnerInfo.loginId = "superuser"
$shareOptions.ShareOptions.sharedUsers = @($userDTO.OwnerInfo)

#Uncomment this line if specific user visibility is not used
#$shareOptions.ShareOptions.PSObject.Properties.Remove("sharedUsers")

# Locate a VM by name
$vm = Get-VMReferences -vmName "My VM"
$vm = $vm.ManagedObjectReferenceCollection.ManagedObjectReferences[0]

# Share the VM with the desired options
$shareResults = New-VMShare -vmId $vm.id -shareOptions $shareOptions

```

### Example 33: Create a completion workflow for shared VMs

This example creates a component-level completion workflow for new requests that targets only shared VMs.

```

#Create a DTO
$sharedVMCompletionWorkflowDTO = New-DTOTemplateObject -DTOTagName
"SharedVMCompletionWorkflowDefinition"

#Set appropriate properties
$sharedVMCompletionWorkflowDTO.SharedVMCompletionWorkflowDefinition.name = "My shared
VM Completion workflow"
$sharedVMCompletionWorkflowDTO.SharedVMCompletionWorkflowDefinition.global = $true

#Create an Email step definition
$emailStepDefinitionDTO = New-DTOTemplateObject -DTOTagName
"WorkflowEmailStepDefinition"

#Clear the existing step(s); add new ones we just created
$sharedVMCompletionWorkflowDTO.SharedVMCompletionWorkflowDefinition.Steps = @()
$sharedVMCompletionWorkflowDTO.SharedVMCompletionWorkflowDefinition.Steps +=
$emailStepDefinitionDTO.WorkflowEmailStepDefinition

#Submit the DTO for creation
$createdDefinition = New-WorkflowDefinition -definitionDto
$sharedVMCompletionWorkflowDTO

```

### Examples for limiting results using query syntax

These examples show how to restrict the data returned using query syntax. See [Using a search query to restrict the data returned](#) above.

```

# Find the first 500 requests with a workflow that has a current workflow step that
failed.
$q = 'workflow.currentStep.fail -eq true'
$requestCollection = Get-ServiceRequests -query $q -size 500

# Find the first 500 requests with a workflow that has a current workflow step that
did not fail.
$q = 'workflow.currentStep.fail -eq false'
$requestCollection = Get-ServiceRequests -query $q -size 500

# Find the first 500 requests in the Pending Completion state.
$q = 'state -eq "Pending Completion"'
$requestCollection = Get-ServiceRequests -query $q -size 500

# Find the first 500 new service requests.
$q = 'type -eq "New Service"'
$requestCollection = Get-ServiceRequests -query $q -size 500

# Find the first 500 workflows where the current step name is Send Acknowledgement
Email.
$q = 'workflow.currentStep.name -eq "Send Acknowledgement Email"'
$requestCollection = Get-ServiceRequests -query $q -size 500

# Find the first 500 workflows where the current step name contains Approval Email.
$q = 'workflow.currentStep.name -contains "Approval Email"'
$requestCollection = Get-ServiceRequests -query $q -size 500

# Find the first 500 workflows submitted on or before December 30, 2015.
$q = 'requestDate -le "2015/12/30 00:00:00"'
$requestCollection = Get-ServiceRequests -query $q -size 500

# Find requests 10-19.
$requestCollection = Get-ServiceRequests -query $q -size 10 -offset 10

# Find requests 20-29.
$requestCollection = Get-ServiceRequests -query $q -size 10 -offset 20

# Find the number of requests in the Pending Completion state.
$q = 'state -eq "Pending Completion"'
$totalCount = Get-ServiceRequestCount -query $q

# Find the number of requests in a state earlier than or equal to Pending Completion.
$q = 'state -le "Pending Completion"'
$totalCount = Get-ServiceRequestCount -query $q

# Find the first 500 new service requests in the Pending Completion state where the
workflow name contains Test1 and the current workflow step name equals ExScript and
the workflow step has failed, plus change requests in any state
$q = '((state -eq "Pending Completion") -and (type -eq "New Service") -and
(workflow.name -contains "Test1") -and (workflow.currentStep.name -eq "ExScript") -and
(workflow.currentStep.fail -eq true)) -or (type -eq "Change Service")'
$requestCollection = Get-ServiceRequests -query $q -size 500 -offset 0

```

## Troubleshooting

To get debug or verbose log statements, add the -Debug or -Verbose switch to any function call. For example:

```
Get-PublishedServiceByName "Published Service #1" -Debug
```

## “Client requires security protocol TLSv1.2 or TLSv1.1 to communicate with server” error

The REST API supports TLSv1.1 and TLSv1.2, which are included with the Microsoft .NET Framework v4.5 and higher. If .NET Framework v2.5 or higher is not installed, the following error is displayed when attempting to use the REST API PowerShell client:

```
Security protocols Used      : Ssl3, Tls
Installed .Net Framework versions : 2.0.50727.5420
                                   3.0.30729.5420
                                   3.0.4506.5420
                                   3.0.6920.5011
                                   3.5.30729.5420
                                   4.0.30319

=====
To begin, call the function Connect-Client.

PS C:\Users\Administrator> Connect-Client
The current installed .NET Framework doesn't support TLSv1.2 or TLSv1.1. Client requires security protocol TLSv1.2
or TLSv1.1 to communicate with server. Security protocols supported on this system: Ssl3 Tls
At C:\Windows\system32\WindowsPowerShell\v1.0\Modules\VCommander\RESTClient\VCommanderRestClient.psml:268 char:3
+ throw "The current installed .NET Framework doesn't support TLSv1.2 or TLSv1.1 ..."
+ ~~~~~
+ CategoryInfo          : OperationStopped: (The current ins...system: Ssl3 Tls:String) [], RuntimeException
+ FullyQualifiedErrorId : The current installed .NET Framework doesn't support TLSv1.2 or TLSv1.1. Client requ
ires security protocol TLSv1.2 or TLSv1.1 to communicate with server. Security protocols supported on this sys
tem: Ssl3 Tls

PS C:\Users\Administrator> |
```

Note the following:

- SSL 2.0 and TLSv1.2 are not compatible on Windows 7 and later operating systems. SSL 2.0 should be disabled as described here: <https://support.microsoft.com/en-us/kb/2851628>
- Not every Windows OS is equipped with TLSv1.2 by default. See the following article for details: <http://blogs.msdn.com/b/kaushal/archive/2011/10/02/support-for-ssl-tls-protocols-on-windows.aspx>
- On some Windows operating systems, TLSv1.2 is supported, but is not enabled by default. The following article explains how to enable it: <https://support.microsoft.com/en-us/kb/245030>

## “Could not load file or assembly” error

### Problem

If you see the following error when trying to run the command `Import-module VCommander`:

```
Import-Module : Could not load file or assembly
'file:///C:\Windows\system32\WindowsPowerShell\v1.0\Modules\VCommander\VCommander.dll'
or one of its dependencies.
Operation is not supported. (Exception from HRESULT: 0x80131515)
At line:1 char:1
+ Import-Module -name VCommander
+ ~~~~~
+ CategoryInfo          : NotSpecified: (:) [Import-Module], FileLoadException
+ FullyQualifiedErrorId :
System.IO.FileLoadException,Microsoft.PowerShell.Commands.ImportModuleCommand
```

You may also see this error for the PowerShellLogging module when initializing the configuration.

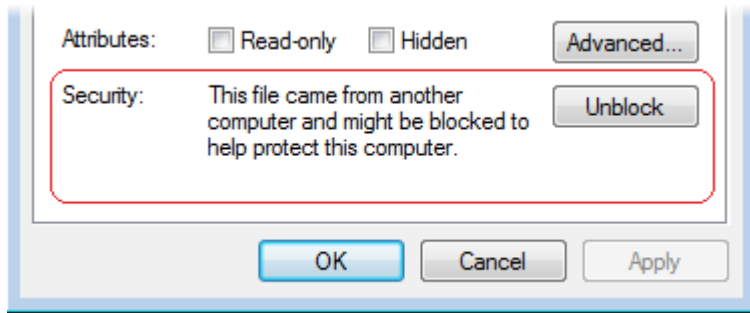
## Resolution

Windows is blocking the use of files from external sources.

Check the permissions on the following file by right-clicking it and selecting **Properties**:

`C:\Windows\system32\WindowsPowerShell\v1.0\Modules\VCommander\VCommander.dll`

On the General tab of the Properties dialog, the following message indicates that the file is blocked:



To unblock the file, you need to unblock the original .zip file and re-extract the files. Clicking **Unblock** for file level will have no effect.

1. Exit PowerShell.
2. Delete the VCommander and VCommanderRestClient folders (and the PowerShellLogging folder, if you saw the error for PowerShellLogging).
3. Right-click the original .zip file you downloaded, select **Properties**, and click **Unblock**.
4. Unzip the files to the same location you used earlier.

Now you should be able to import the module successfully (or, in the case of the PowerShellLogging module, you should be able to initialize the configuration successfully).

## “Module file not found ” error when attempting to run Import-Module

### Problem

If you see the following error when trying to run the command `Import-Module <module>`:

```
Import-Module : The specified module 'XXX' was not loaded because no valid module file was found in any module directory.
```

### Resolution

Install all modules in the appropriate location. See the [Installation section](#) for details. Be aware that you must place each module in a folder with the same name as the module. For example:

```
.../WindowsPowerShell/v1.0/Modules/XXX/XXX.psml
.../WindowsPowerShell/v1.0/Modules/XXX/XXX.psd1
```

Also, verify that you’ve installed your modules to a system location and not your personal location. Enter the following variable in your PowerShell console to see the module paths:



```
$env:PSModulePath  
(or $PSHOME)
```

This variable will show the current module paths. There should be at least two: one will be your user directory, and the other will be \$PSHOME\Modules.

## “Untrusted certificate” error

If you see an untrusted certificate error, you can call the following function to ignore the error:

**VCommander\Set-IgnoreSslErrors**

You can also use the `--ssltrustall` option with the `Connect-Client2` command. See [Using the REST API with a Service Portal account](#) above.

## “Content is not allowed in prolog ” error when attempting to call an API

### Problem

If you see the following error when trying to call an API:

```
Invoke-WebRequest : JAXBException occurred : Content is not allowed in prolog.
```

### Resolution

This often happens when passing incorrect DTOs, specifically a DTO without a root container, to an API.

For example:

**New-Account -accountDto**



The above API expects an Account DTO object. If we create a DTO using the New-DTOTemplateObject API, we would see the following:

```
PS H:\> $accountTemplate = New-DTOTemplateObject -DTOTagName "Account"
Creating template DTO object for Account

PS H:\> $accountTemplate

Account
-----
System.Object

PS H:\> $accountTemplate.Account

id                : 0
emailAddress      : bob@embotics.com
enabled           : true
firstName         : 
lastName          : 
password          : 
primaryPhoneNo    : 
role              : System.Object
secondaryPhoneNo  : 
securitySourceType : USER_DIRECTORY_USER
userid            : bob@embotics.com
class             : wsAccount
```

The variable \$accountTemplate has an “Account” object, while its sub-property \$accountTemplate.Account contains a number of properties. If we convert the variable \$accountTemplate into XML, we would see the following:

```
PS H:\> $xmlOfAccountTemplate = Convert-ObjectToXml $accountTemplate

PS H:\> $xmlOfAccountTemplate
<Account xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="wsAccount">
  <id>0</id>
  <emailAddress>bob@embotics.com</emailAddress>
  <enabled>true</enabled>
  <firstName></firstName>
  <lastName></lastName>
  <password></password>
  <primaryPhoneNo></primaryPhoneNo>
  <role>
    <id>-1</id>
    <key>RoleKey</key>
  </role>
  <secondaryPhoneNo></secondaryPhoneNo>
  <securitySourceType>USER_DIRECTORY_USER</securitySourceType>
  <userid>bob@embotics.com</userid>
</Account>
```

Notice the root container “Account” for the XML. This is properly formed XML.

If we convert the sub-property `$accountTemplate.Account` into XML, we would see:

```
PS H:\> $xmlOfAccountTemplateAccount = Convert-ObjectToXml $accountTemplate.Account

PS H:\> $xmlOfAccountTemplateAccount
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="wsAccount"<id>0</id>
<emailAddress>bob@embotics.com</emailAddress>
<enabled>true</enabled>
<firstName></firstName>
<lastName></lastName>
<password></password>
<primaryPhoneNo></primaryPhoneNo>
<role>
  <id>-1</id>
  <key>RoleKey</key>
</role>
<secondaryPhoneNo></secondaryPhoneNo>
<securitySourceType>USER_DIRECTORY_USER</securitySourceType>
<userid>bob@embotics.com</userid>
```

Notice how the XML is ill-formed; there is no root container. The correct DTO to pass to the vCommander REST service is:

```
$updatedAccount = New-Account -accountDto $accountTemplate
```

And not:

```
$updatedAccount = New-Account -accountDto $accountTemplate.Account
```

The former has a properly formed XML, while the latter does not.

## Error: JAXBException occurred : The reference to entity "{0}" must end with the ';' delimiter

The REST API uses XML for message exchanges between the REST client and the vCommander server. If the XML contains data having characters with special meaning, then they must be escaped.

For example:

Division that the VM is allocated to (e.g. R&D, Marketing, Customer Service, etc.)

The & character must be escaped as:

R&amp;D

## Appendix: PowerShell script credential encryption

The VCommander module uses two-stage encryption to protect credentials used in scripts. The goal is to prevent exposure of plaintext passwords through workflow steps during script execution. In the first stage, passwords are RSA-encrypted with a machine key. The encryption process produces an encrypted credential key file which can be stored on disk. During script executions, the VCommander module is used to decrypt this key and re-encrypt the password in a form that can be stored in memory. The second stage uses a standard Windows function to receive the password securely without storing it in memory as clear text.

The encrypted accounts can be either domain or local users.

### To encrypt your credentials

1. Open a PowerShell console/ISE as administrator.
2. Import the VCommander module into PowerShell:
3. If the user execution policy is set to Restricted, set it to Remote Signed with the following command:
4. Confirm this change by clicking **Yes**.
5. Encrypt your credentials into a file for storage on disk:

```
Import-Module VCommander
```

```
Set-ExecutionPolicy RemoteSigned
```

```
New-EncryptCredential -destFilePath "<file-path>"
```

Make sure the destination directory exists. If the destination directory doesn't exist, an error will be thrown. If the destination file exists, it will be overwritten.

It's best practice to give this file an .xml extension.

For example:

```
New-EncryptCredential -destFilePath "C:\\Scripts\\vcmdr_credential_key.xml"
```

6. Enter the username and password for the account.

The credential is encrypted, and the resulting encrypted object is serialized into the specified file. Repeat the above steps for other credentials.

Regenerate these files whenever the password changes.

### To decrypt your credentials

1. Open a PowerShell console/ISE or your scripts.
2. Import the VCommander module into PowerShell:

```
Import-Module VCommander
```

3. If the user execution policy is set to Restricted, set it to Remote Signed with the following command:

```
Set-ExecutionPolicy RemoteSigned
```

4. Decrypt your credentials from file :

```
New-DecryptCredential -keyFilePath "<file-path>"
```

5. An error will be thrown if the file path is incorrect. For example:

```
$credential = New-DecryptCredential -keyFilePath  
"C:\Scripts\vcmdr_credential_key.xml"
```

The \$credential variable is a PSCredential object containing your decrypted credentials.

6. Repeat the above steps for other credentials.

## The encryption process

1. The credentials are acquired from the user via prompt as a SecureString (System.Security.SecureString).
2. Using the PowerShell function ConvertFrom-SecureString, the password portion of this secure string is encrypted with AES using a 256 bit key. This key is contained in Security.psm1 and can be modified.
3. An RSA machine key container is created, using the cryptographic service provider (CSP).
4. The username and password from step 2 are encrypted and stored in the machine keystore, using the key created in step 3 with the Microsoft RSA crypto provider.
5. The encrypted bytes are serialized to disk as XML.

### Notes

- The machine key is persisted in the computer's key store instead of the user profile store.
- This machine-level RSA key container is available to all users that can log in to a computer. NTFS access control lists (ACLs) on the server can be used to restrict access to the machine key.
- The machine key is stored at the following location:  
...\\Application Data\\Microsoft\\Crypto\\RSA\\
- Because the key is stored on the machine that generated the encrypted credential file, encryption and decryption must be done on that same machine. Importing the RSA key container to a different machine is not covered in this guide.

## Appendix: vCommander REST API security and authentication reference

### Client authentication

vCommander uses SSL and session-based authentication to secure the REST API service.

All REST API calls:

- must take place over HTTPS
- must have a security token (or access token) in the http header of each API call. This token is provided when a client authenticates with the vCommander service.

### Login

**Operation:** POST /sessions

**Description:** To authenticate to the API service, POST a request to its login URL. The request body must contain a MIME Base64 encoding of the client credentials in the form:

user@domain:password (domain credentials)

or

user:password (for a local user)

or

user:password:orgId (if connecting with a local account that has a Service Portal role)

**Output:** An HTTP response is returned, along with a security token in the header. The security token is in the following format (see also the example below):

securityToken: <token>

**Note:** This security token is required in all subsequent API requests to the service. The token must be transmitted to the service via the HTTP header. The token has an expiry date. As long as requests are made before the expiry date, the expiry date of this token will be extended. An expired security token requires the client to re-authenticate.

### Example:

#### Request

HTTP 1.1

POST / webservices/services/rest/v2/sessions

Body:

PZQkW0HUSFQVeNyZO5uu/TzQie6YYXQdEifBhdL05A

Response

HTTP/1.1 200 OK

## Headers:

securitytoken : DCFyZO0HUSF5uuS0HUSFDGSGif0HUSFBhD=!bob

**Logout**

**Operation:** DELETE / webservices/services/rest/v2/sessions/<securitytoken>

**Description:** To log out and terminate the vCommander REST API session, delete the session you created when you logged in.

**Output:** An HTTP response (200 OK) if successful.

**Notes:** This request, like all other authenticated requests, must include the security token in the HTTP header.

**Example:**Request

HTTP 1.1

DELETE / webservices/services/rest/v2/sessions/  
DCFyZO0HUSF5uuS0HUSFDGSGif0HUSFBhD=!bob

Response

HTTP/1.1 200 OK

## Session overview

