VMware vSphere Automation SDK for Python Programming Guide

Modified on 18 JUN 2018
vSphere Automation SDK for Python 6.5
vCenter Server 6.5
VMware ESXi 6.5

vmware®

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

https://docs.vmware.com/

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

Contents

About VMware vSphere Automation SDK for Python Programming Guide

VMware vSphere Automation SDK for Python Programming Guide describes how to use the VMware vSphere Automation SDK for Python. VMware provides different APIs and SDKs for different applications and goals. The vSphere Automation SDK for Python supports the development of clients that use the vSphere Automation SDK for infrastructure support tasks .

Intended Audience

This information is intended for anyone who will develop applications using the vSphere Automation SDK for Python. Some programming background in Python is required.

Updated Information

This *VMware vSphere Automation SDK for Python Programming Guide* is updated with each release of the product or when necessary.

This table provides the update history of the *VMware vSphere Automation SDK for Python Programming Guide*.

Revision	Description
18 JUN 2018	Updated some code snippets.
15 NOV 2016	Initial release.

Introduction to the vSphere Automation SDKs

1

The vSphere Automation SDKs bundle client libraries for accessing new vSphere Automation features like Content Library and existing features like Tagging. The vSphere Automation SDKs contain sample applications and API reference documentation for the Content Library and Tagging services. The vSphere Automation SDKs also provide sample code that retrieves the endpoints of vSphere Automation and vSphere services and establishes a secure connection with the vSphere Automation endpoint.

vSphere Automation supports six languages for accessing the vSphere Automation API services and provides six SDKs for developing client applications for managing components in your virtual environment.

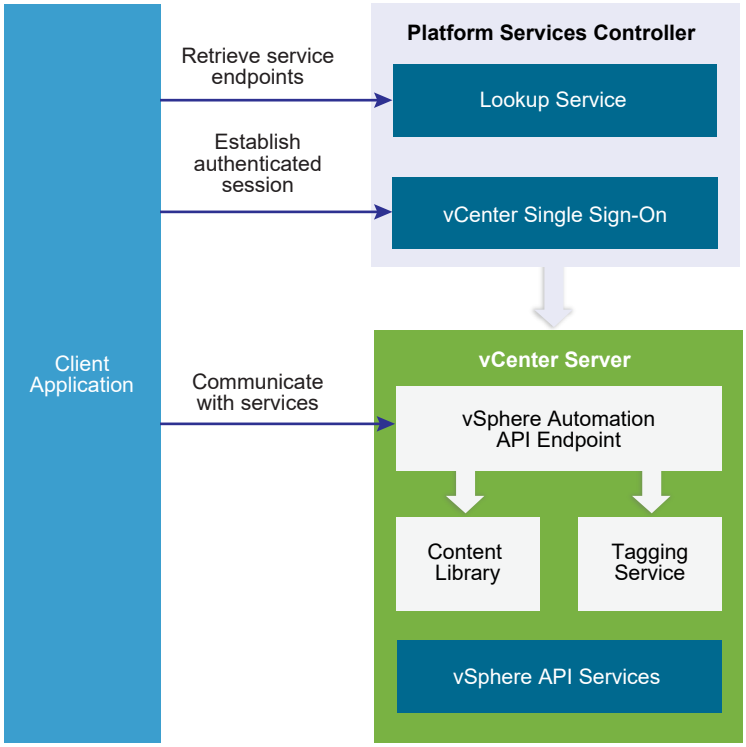
This chapter includes the following topics:

vSphere Automation SDKs Overview

The vSphere Automation API provides a unified programming interface to vSphere Automation services that you can use through SDKs provided in six programming languages. The vSphere Automation API provides a service-oriented architecture for accessing resources in the virtual environment by issuing requests to the vSphere Automation Endpoint.

vSphere Automation API client applications communicate with services on the Platform Services Controller and vCenter Server.

Figure 1-1. Communication Model of vSphere Automation API Client Applications



vSphere Automation API client applications use the Lookup Service to retrieve the vCenter Single Sign-On endpoint, the vSphere Automation Endpoint, and the endpoints of services that are exposed through the vSphere API. To access vSphere Automation services such as Content Library and Tagging, client applications issue requests to the vSphere Automation Endpoint. By using the vCenter Single Sign-On service, client applications can either establish an authenticated vSphere Automation session, or authenticate individual requests to the vSphere Automation Endpoint.

Client applications can access services that are exposed through the vSphere API by using the vSphere Management SDK.

Depending on the vSphere deployment model, client applications can communicate with vSphere Automation services on a single vCenter Server instance or multiple vCenter Server instances. For more information about the vSphere deployment models, see Chapter 2 Components of the vSphere Automation Virtualization Layer

SDK Developer Setup

To start developing a vSphere Automation API client application, you must download the software and set up a development environment. You can find instructions for setting up a development environment in the README for each vSphere Automation SDK.

SDK Samples

The vSphere Automation SDKs provide sample applications that you can extend to implement client applications that serve your needs. The code examples in the vSphere Automation SDKs documentation are based on these sample applications.

Supported Programming Languages

The vSphere Automation SDKs are packed in six different programming languages that let you build client applications on your preferred programming language.

- vSphere Automation SDK for Java
- vSphere Automation SDK for Python
- vSphere Automation SDK for .NET
- vSphere Automation SDK for Perl
- vSphere Automation SDK for Ruby
- vSphere Automation SDK for REST

Components of the vSphere Automation Virtualization Layer

2

At the core of vSphere Automation is vSphere, which provides the virtualization layer of the software-defined data center. You can use vSphere deployment options for vCenter Server and ESXi hosts to build virtual environments of different scales.

This chapter includes the following topics:

Components and Services of a Virtual Environment

Starting with vSphere 6.0, the deployment of the virtual environment consists of two major components that provide different sets of services, the VMware Platform Services Controller and vCenter Server. You can deploy vCenter Server with an embedded or external Platform Services Controller.

Services Installed with Platform Services Controller

The Platform Services Controller group of infrastructure services contains vCenter Single Sign-On, License Service, Lookup Service, and VMware Certificate Authority. The services installed with the Platform Services Controller are common to the entire virtual environment. A Platform Services Controller can be connected to one or more vCenter Server instances. In a deployment that consists of more than one Platform Services Controller, the data of each service is replicated across all Platform Services Controller instances.

In vSphere Automation API client applications, you use the vCenter Single Sign-On and the Lookup Service on the Platform Services Controller to provide a range of functionality.

Authentication and Session Management

You use the vCenter Single Sign-On service to establish an authenticated session with the vSphere Automation Endpoint. You send credentials to the vCenter Single Sign-On service and receive a SAML token that you use to obtain a session ID from the vSphere Automation Endpoint. Alternatively, you can access the vSphere Automation APIs in a sessionless manner

by including the SAML token in every request that you issue to the vSphere Automation Endpoint.

Service Discovery

You use the Lookup Service to discover the endpoint URL for the vCenter Single Sign-On service on the Platform Services Controller, the location of the vCenter Server instances, and the vSphere Automation Endpoint.

Services Installed with vCenter Server

vCenter Server is a central administration point for ESXi hosts. The vCenter Server group of services contains vCenter Server, vSphere Web Client, Inventory Service, vSphere® Auto Deploy™, vSphere® ESXi™ Dump Collector, VMware vSphere® Syslog Collector on Windows and VMware vSphere Syslog Service for the vCenter Server Appliance.

vCenter Server also provides services that you can access through the vSphere Automation Endpoint.

Content Library Service

You can use the Content Library Service to share VM templates, vApp templates, and other files across the software-defined data center. You can create, share, and subscribe to content libraries on the same vCenter Server instance or on a remote instance. This promotes consistency, compliance, efficiency, and automation in deploying workloads at scale. By using content libraries, you can also create OVF packages from virtual machines and virtual appliances in hosts, resource pools, and clusters. You can then use the OVF packages to provision new virtual machines in hosts, resource pools, and clusters.

Tagging Service

This service supports the definition of tags that you can associate with vSphere objects or vSphere Automation resources. The vSphere Automation SDKs provide the capability to manage tags programmatically.

vSphere Deployment Configurations

vSphere Automation client applications communicate with services on the Platform Services Controller and vCenter Server components of the virtual environment. vCenter Server can be deployed with an embedded or external Platform Services Controller.

vCenter Server with an Embedded Platform Services Controller

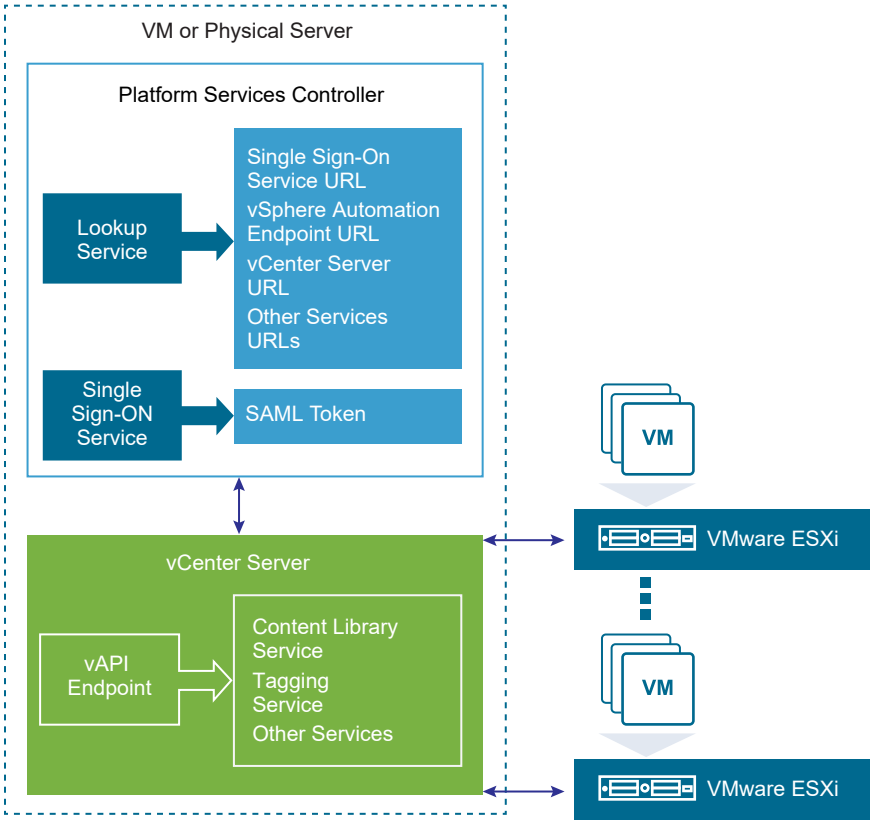
vCenter Server and Platform Services Controller reside on the same virtual machine or physical server. This deployment is most suitable for small environments such as development or test beds.

You can use the Platform Services Controller in two ways to establish secure, authenticated sessions for your client application, by making requests to the Lookup Service and the vCenter Single Sign-On Service.

One way to use the Platform Services Controller is to request an authentication token that can be used to authenticate requests across services. The client connects to the Lookup Service and retrieves the vCenter Single Sign-On Service endpoint and the vSphere Automation API endpoint. The client then uses the vCenter Single Sign-On endpoint to authenticate with user credentials and receive a token that securely verifies the client's credentials. This allows the client to authenticate with a number of service endpoints without sending user credentials over the network repeatedly.

Alternatively, if the client connects directly to the vSphere Automation API endpoint, there is no need for the client to interact with the vCenter Single Sign-On Service. The client sends user credentials to the vSphere Automation API endpoint, which creates a session identifier that persists across requests.

Figure 2-1. vCenter Server with Embedded Platform Services Controller

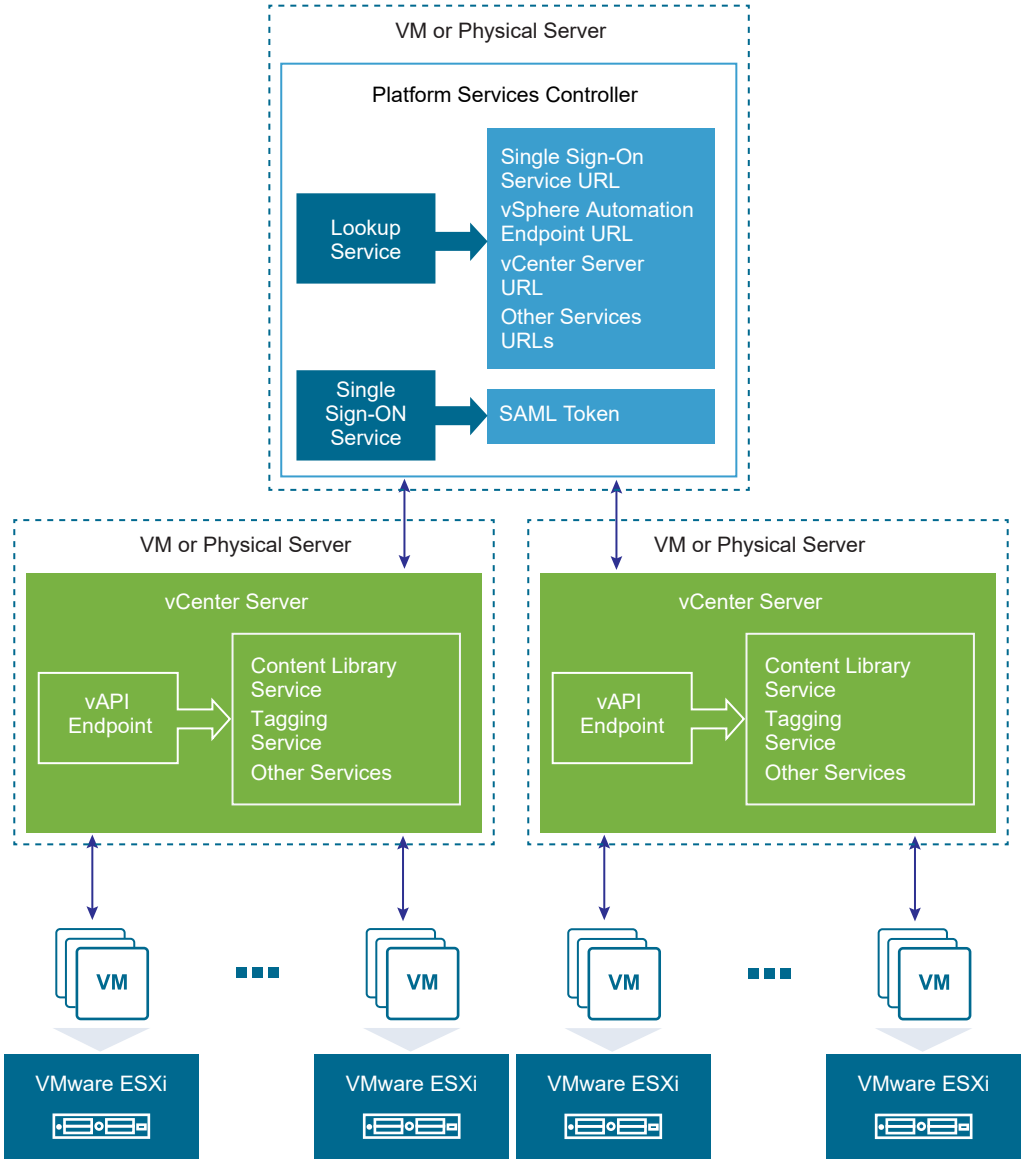


vCenter Server with an External Platform Services Controller

In the case of an external Platform Services Controller, the vCenter Server and the Platform Services Controller are deployed on separate virtual machines or physical servers. The Platform Services Controller can be shared across several vCenter Server instances. For larger deployments or to provide better availability, more than one Platform Services Controller can be deployed. When configured as replication partners within a single vCenter Single Sign-On domain, Platform Services Controller instances replicate all user and system data within the cluster.

A client application functions in a similar way as in a Platform Services Controller with embedded vCenter Server deployment. The only difference is that the client application can access services on multiple vCenter Server instances, or services only on a particular vCenter Server instance.

Figure 2-2. vCenter Server with External Platform Services Controller



Retrieving Service Endpoints

3

To access services and resources in the virtual environment, vSphere Automation API client applications must know the endpoints of vSphere Automation and vSphere services. Client applications retrieve service endpoints from the Lookup Service that runs on the Platform Services Controller.

The Lookup Service provides service registration and discovery by using a Web services API. By using the Lookup Service, you can retrieve endpoints of services on the Platform Services Controller and vCenter Server. The following endpoints are available from the Lookup Service.

- The vCenter Single Sign-On endpoint on the Platform Services Controller. You use the vCenter Single Sign-On service to get a SAML token and establish an authenticated session with a vSphere Automation endpoint or a vCenter Server endpoint.
- The vSphere Automation Endpoint on vCenter Server. Through the vSphere Automation Endpoint, you can make requests to vSphere Automation API services such as virtual machine management, Content Library, and Tagging.
- The vCenter Server endpoint. In an environment with external Platform Services Controller instances, you can use the vCenter Server endpoint to get the node ID of a particular vCenter Server instance. By using the node ID , you can retrieve service endpoints on that vCenter Server instance.
- The vSphere API endpoint and endpoints of other vSphere services that run on vCenter Server.

Workflow for Retrieving Service Endpoints

The workflow that you use to retrieve service endpoints from the Lookup Service might vary depending on the endpoints that you need and their number. Follow this general workflow for retrieving service endpoints.

1 Connect to the Lookup Service on the Platform Services Controller and service registration object so that you can query for registered services.

2 Create a service registration filter for the endpoints that you want to retrieve.

3 Use the filter to retrieve registration information for services from the Lookup Service.

4 Extract one or more endpoint URLs from the array of registration information that you receive from the Lookup Service.

This chapter includes the following topics:

Filtering for Predefined Service Endpoints

The Lookup Service maintains a registration list of vSphere services. You can use the Lookup Service to retrieve registration information for any service by setting a registration filter that you pass to the `List()` function on the Lookup Service. The functions and objects that you can use with the Lookup Service are defined in the `lookup.wsdl` file that is part of the SDK.

Lookup Service Registration Filters

You can query for service endpoints through a service registration object that you obtain from the Lookup Service. You invoke the `List()` function on the Lookup Service to list the endpoints that you need by passing `LookupServiceRegistrationFilter`. `LookupServiceRegistrationFilter` identifies the service and the endpoint type that you can retrieve.

Optionally, you can include the node ID parameter in the filter to identify the vCenter Server instance where the endpoint is hosted. When the node ID is omitted, the `List()` function returns the set of endpoint URLs for all instances of the service that are hosted on different vCenter Server instances in the environment.

For example, a `LookupServiceRegistrationFilter` for querying the vSphere Automation service has these service endpoint elements.

Table 3-1. Service Registration Filter Parameters

Filter Types	Value	Description
`LookupServiceRegistrationServiceType`	`product= "com.vmware.cis"`	vSphere Automation namespace.
	`type="cs.vapi"`	Identifies the vSphere Automation service.
`LookupServiceRegistrationEndpointType`	`type="com.vmware.vapi.endpoint"`	Specifies the endpoint path for the service.
	`protocol= "vapi.json.https.public"`	Identifies the protocol that will be used for communication with the endpoint .

For information about the filter parameter of the available predefined service endpoints, see Filter Parameters for Predefined Service Endpoints.

Filter Parameters for Predefined Service Endpoints

Depending on the service endpoint that you want to retrieve, you provide different parameters to the `LookupServiceRegistrationFilter` that you pass to the `List()` function on the Lookup Service. To search for services on a particular vCenter Server instance, set the node ID parameter to the filter.

Table 3-2. Input Data for URL Retrieval for the Lookup Service Registration Filter

Service	Input Data	Value
vCenter Single Sign-On	product namespace	`com.vmware.cis`
	service type	`cs.identity`
	protocol	`wsTrust`
	endpoint type	`com.vmware.cis.cs.identity.sso`
vSphere Automation Endpoint	product namespace	`com.vmware.cis`
	service type	`cs.vapi`
	protocol	`vapi.json.https.public`
	endpoint type	`com.vmware.vapi.endpoint`
vCenter Server	product namespace	`com.vmware.cis`
	service type	`vcenterserver`
	protocol	`vmomi`
	endpoint type	`com.vmware.vim`
vCenter Storage Monitoring Service	product namespace	`com.vmware.vim.sms`
	service type	`sms`
	protocol	`https`
	endpoint type	`com.vmware.vim.sms`
vCenter Storage Policy-Based Management	product namespace	`com.vmware.vim.sms`
	service type	`sms`
	protocol	`https`
	endpoint type	`com.vmware.vim.pbm`
vSphere ESX Agent Manager	product namespace	`com.vmware.vim.sms`
	service type	`cs.eam`

Table 3-2. Input Data for URL Retrieval for the Lookup Service Registration Filter (continued)

Service	Input Data	Value
	protocol	`vmomi`
	endpoint type	`com.vmware.cis.cs.eam.sdk`

Connect to the Lookup Service and Retrieve the Service Registration Object

You must connect to the Lookup Service to gain access to its operations. After you connect to the Lookup Service, you must retrieve the service registration object to make registration queries.

- Python Example of Connecting to the Lookup Service and Retrieving a Service Registration Object

  The example is based on the code from the `lookup_service_helper.py` sample file.

Procedure

1. Connect to the Lookup Service.
   a. Configure a connection stub for the Lookup Service endpoint, which uses SOAP bindings, by using the HTTPS protocol.
   b. Create a connection object to communicate with the Lookup Service.
2. Retrieve the Service Registration Object.
   a. Create a managed object reference to the Service Instance.
   b. Invoke the `RetrieveServiceContent()` method to retrieve the `ServiceContent` data object.
   c. Save the managed object reference to the service registration object.

      With the service registration object, you can make registration queries.

Python Example of Connecting to the Lookup Service and Retrieving a Service Registration Object

The example is based on the code from the `lookup_service_helper.py` sample file.

Note For a complete and up-to-date version of the sample code, see the vSphere Automation SDK Python samples at GitHub.

```
...

# 1 - Create SOAP client object to communicate with the Lookup Service.
my_ls_stub = Client(url=wsdl_url, location=ls_url)

# 2 - Configure service & port type for client transaction.
```

```
my_ls_stub.set_options(service='LsService', port='LsPort')

# 3 – Manufacture a managed object reference.
managed_object_ref = \
   my_ls_stub.factory.create('ns0:ManagedObjectReference')
managed_object_ref._type = 'LookupServiceInstance'
managed_object_ref.value = 'ServiceInstance'

# 4 – Retrieve the ServiceContent object.
ls_service_content = \
my_ls_stub.service.RetrieveServiceContent(managed_object_ref)

# 5 – Retrieve the service registration object.
service_registration = ls_service_content.serviceRegistration
```

Retrieve Service Endpoints on vCenter Server Instances

You can create a function that obtains the endpoint URLs of a service on all vCenter Server instances in the environment. You can modify that function to obtain the endpoint URL of a service on a particular vCenter Server instance.

Prerequisites

- Establish a connection with the Lookup Service.
- Retrieve a service registration object.

Procedure

1. Create a registration filter object, which contains the following parts:
   - A filter criterion for service information
   - A filter criterion for endpoint information

Option	Description
Omit the node ID parameter	Retrieves the endpoint URLs of the service on all vCenter Server instances.
Include the node ID parameter	Retrieves the endpoint URL of the service on a particular vCenter Server instance.

2. Retrieve the specified service information by using the `List()` function.

Results

Depending on whether you included the node ID parameter, the `List()` function returns one of the following results:

- A list of endpoint URLs for a service that is hosted on all vCenter Server instances in the environment.
- An endpoint URL of a service that runs on a particular vCenter Server instance.

What to do next

Call the function that you implemented to retrieve service endpoints. You can pass different filter parameters depending on the service endpoints that you need. For more information, see Filter Parameters for Predefined Service Endpoints.

To retrieve a service endpoint on a particular vCenter Server instance, you must retrieve the node ID of that instance and pass it to the function. For information about how to retrieve the ID of a vCenter Server instance, see Retrieve a vCenter Server ID by Using the Lookup Service.

Python Example of Retrieving a Service Endpoint on a vCenter Server Instance

This example provides a common pattern for filtering Lookup Service registration data. This example is based on the code in the `lookup_service_helper.py` sample file.

Note For a complete and up-to-date version of the sample code, see the vSphere Automation SDK Python samples at GitHub.

```
def lookup_service_infos(prod, svc_type, proto, ep_type, node_id) :

   # 1 – Create a filter criterion for service info.
   filter_service_type = \
my_ls_stub.factory.create('ns0:LookupServiceRegistrationServiceType')
   filter_service_type.product = prod
   filter_service_type.type = svc_type

   # 2 – Create a filter criterion for endpoint info.
   filter_endpoint_type = \
my_ls_stub.factory.create('ns0:LookupServiceRegistrationEndpointType')
   filter_endpoint_type.protocol = proto
   filter_endpoint_type.type = ep_type

   # 3 – Create the registration filter object.
   filter_criteria = \
my_ls_stub.factory.create('ns0:LookupServiceRegistrationFilter')
   filter_criteria.serviceType = filter_service_type
   filter_criteria.endpointType = filter_endpoint_type
   filter_criteria.nodeId = node_id

   # 4 – Retrieve specified service info with the List() method.
   service_infos = my_ls_stub.service.List(service_registration,
filter_criteria)
   return service_infos
```

Retrieve a vCenter Server ID by Using the Lookup Service

You use the node ID of a vCenter Server instance to retrieve the endpoint URL of a service on that vCenter Server instance. You specify the node ID in the service registration filter that you pass to the `List()` function on the Lookup Service.

Managed services are registered with the instance name of the vCenter Server instance where they run. The instance name maps to a unique vCenter Server ID. The instance name of a vCenter Server system is specified during installation and might be an FQDN or an IP address.

Prerequisites

- Establish a connection with the Lookup Service.
- Retrieve a service registration object.

Procedure

1. List the vCenter Server instances.
2. Find the matching node name of the vCenter Server instance and save the ID.

Results

Use the node ID of the vCenter Server instance to filter subsequent endpoint requests. You can use the node ID in a function that retrieves the endpoint URL of a service on a vCenter Server instance. For information about implementing such a function, see Retrieve Service Endpoints on vCenter Server Instances.

Python Example of Retrieving a vCenter Server ID by Using the Lookup Service

This example provides a common pattern for filtering Lookup Service registration data. This example is based on the code in the `lookup_service_helper.py` sample file.

Note For a complete and up-to-date version of the sample code, see the vSphere Automation SDK Python samples at GitHub.

```
def get_mgmt_node_id(node_instance_name) :

   # 1 – List the vCenter Server instances.
   mgmt_node_infos = lookup_service_infos(prod='com.vmware.cis',
                                          svc_type='vcenterserver',
                                          proto='vmomi', ep_type='com.vmware.vim',
                                          node_id='*')

   # 2 – Find the matching node name and save the ID.
   for node in mgmt_node_infos :
     for attribute in node.serviceAttributes :
       if attribute.key == 'com.vmware.vim.vcenter.instanceName' :
         if attribute.value == node_instance_name :
           return node.nodeId
```

Retrieve a vSphere Automation Endpoint on a vCenter Server Instance

Through the vSphere Automation Endpoint, you can access other vSphere Automation services that run on vCenter Server, such as Content Library and Tagging. To use a vSphere Automation service, you must retrieve the vSphere Automation Endpoint.

Prerequisites

- Establish a connection with the Lookup Service.
- Retrieve a service registration object.
- Determine the node ID of the vCenter Server instance where the vSphere Automation service runs.
- Implement a function that retrieves the endpoint URL of a service on a vCenter Server instance.

Procedure

1. Invoke the function for retrieving the endpoint URL of a service on a vCenter Server instance by passing filter strings that are specific to the vSphere Automation endpoint.
2. Save the URL from the resulting single-element list.

Python Example of Retrieving a vSphere Automation Endpoint on a vCenter Server Instance

This example provides a common pattern for filtering Lookup Service registration data. This example is based on the code in the `lookup_service_helper.py` sample file.

Note For a complete and up-to-date version of the sample code, see the vSphere Automation SDK Python samples at GitHub.

```
service_infos = lookup_service_infos(prod='com.vmware.cis',
                                     svc_type='cs.vapi',
                                     proto='vapi.json.https.public',
                                     ep_type='com.vmware.vapi.endpoint',
                                     node_id=my_mgmt_node_id)
my_vapi_url = service_infos[0].serviceEndpoints[0].url
```

Authentication Mechanisms

4

To perform operations on services in the vSphere environment, you must create an authenticated connection to the services that you want to use. With the vSphere Automation SDK for Python you can authenticate and access vSphere Automation services.

Client applications can choose from two supported authentication patterns for accessing services in the virtual environment.

For better security, client applications can request a security token to authenticate connections with the vSphere Automation services.

To invoke operations on services, client applications must create a security context. The security context represents the client authentication. You can achieve authentication by using one of the following mechanisms.

Password-Based Authentication

To authenticate with user name and password, you connect to the vSphere Automation Endpoint with vCenter Single Sign-On user credentials and obtain a session identifier (ID). The user account credentials are validated by the vSphere Automation Endpoint, and must be present in the vCenter Single Sign-On identity store. The session ID is valid only for the service endpoint that you want to access and that issues the session ID.

Token-Based Authentication

Client applications can authenticate by using the vCenter Single Sign-On component on thePlatform Services Controller. vCenter Single Sign-On includes the Security Token Service (STS) that issues security tokens. The token must comply with the Security Assertion Markup Language (SAML) specification, which defines an XML-based encoding for communicating authentication data.

vCenter Single Sign-On supports two types of security tokens: bearer token and Holder-of-Key (HoK) token. To acquire a SAML token, client applications must issue a token request to vCenter Single Sign-On.

Client applications can present a SAML token to the vSphere Automation Endpoint in exchange for a session identifier with which they can perform a series of authenticated operations.

To retrieve a session ID for the vSphere Web Services endpoint, you provide the SAML token to the vSphere Web services endpoint. For more information about creating an authenticated session to access the vSphere Web Services, see the *vSphere Web Services SDK Programming Guide* documentation.

This chapter includes the following topics:

Retrieve a SAML Token

The vCenter Single Sign-On service provides authentication mechanisms for securing the operations that your client application performs in the virtual environment. Client applications use SAML security tokens for authentication.

Client applications use the vCenter Single Sign-On service to retrieve SAML tokens. For more information about how to acquire a SAML security token, see the *vCenter Single Sign-On Programming Guide* documentation.

The vSphere Automation SDK for Python provides a utility class to simplify the task of requesting a SAML token from the vCenter Single Sign-On service. The utility provides a wrapper around the complexity of handling token requests. For more information about the utility, see the `sso.py` sample file. The source file is in the vSphere Automation SDK for Python directory: `client/samples/src/com/vmware/vcloud/suite/sample/common/sso.py`.

Prerequisites

Verify that you have the vCenter Single Sign-On URL. You can use the Lookup Service on the Platform Services Controller to obtain the endpoint URL. For information about retrieving service endpoints, see Chapter 3 Retrieving Service Endpoints.

Procedure

1. Create a connection object to communicate with the vCenter Single Sign-On service.

   Pass the vCenter Single Sign-On endpoint URL, which you can get from the Lookup Service.

2. Issue a security token request by sending valid user credentials to the vCenter Single Sign-On service on the Platform Services Controller.

Results

The vCenter Single Sign-On service returns a SAML token.

What to do next

You can present the SAML token to the vSphere Automation API Endpoint or other endpoints, such as the vSphere Web Services Endpoint. The endpoint returns a session ID and establishes a persistent session with that endpoint. Each endpoint that you connect to uses your SAML token to create its own session.

Python Example of Retrieving a SAML Token

This example is based on the code in the `external_psc_sso_workflow.py` sample file.

This example uses the following global variables.

- *my_vapi_hostname*
- *my_sso_username*
- *my_sso_password*

Note For a complete and up-to-date version of the sample code, see the vSphere Automation SDK Python samples at GitHub.

```
from vsphere.samples.common import sso

# Use the SsoAuthenticator utility class to retrieve
# a bearer SAML token from the vCenter Single Sign-On service.
sso_url = 'https://' + my_vapi_hostname + ':7444/ims/STSService'
authenticator = sso.SsoAuthenticator(sso_url)
saml_token = authenticator.get_bearer_saml_assertion(my_sso_username,
                                                     my_sso_password,
                                                     delegatable=True)
```

Create a vSphere Automation Session with a SAML Token

To establish a vSphere Automation session, you create a connection to the vSphere Automation API Endpoint and then you authenticate with a SAML token to create a session for the connection.

Prerequisites

- Retrieve the vSphere Automation Endpoint URL from the Lookup Service.
- Obtain a SAML token from the vCenter Single Sign-On service.

Procedure

1. Create a connection by specifying the vSphere Automation API Endpoint URL and the message protocol to be used for the connection.

   Note To transmit your requests securely, use **`https`** for the vSphere Automation API Endpoint URL.

2 Create the request options or stub configuration and set the security context to be used.

   The security context object contains the SAML token retrieved from the vCenter Single Sign-On service. Optionally, the security context might contain the private key of the client application.

3 Create an interface stub or a REST path that uses the stub configuration instance.

   The interface stub corresponds to the interface containing the method to be invoked.

4 Invoke the session `create` method.

   The service creates an authenticated session and returns a session identification cookie to the client.

5 Create a security context instance and add the session ID to it.

6 Update the stub configuration instance with the session security context.

What to do next

Use the updated stub configuration with the session ID to create a stub for the interface that you want to use. Method calls on the new stub use the session ID to authenticate.

Python Example of Creating a vSphere Automation API Session with a SAML Token

This example is based on code in the `external_psc_sso_workflow.py` sample file.

This example uses the following global variables.

- *my_vapi_hostname*
- *my_stub_config*
- *saml_token*

The example assumes that you previously obtained a vSphere Automation API URL from the Lookup Service, and a SAML token from the vCenter Single Sign-On Service.

Note For a complete and up-to-date version of the sample code, see the vSphere Automation SDK Python samples at GitHub.

```
...

# Create a session object in the client.
session = requests.Session()

# For development environment only, suppress server certificate checking.
session.verify = False
from requests.packages.urllib3 import disable_warnings
from requests.packages.urllib3.exceptions import InsecureRequestWarning
disable_warnings(InsecureRequestWarning)

# Create a connection for the session.
```

```
vapi_url = 'https://' + my_vapi_hostname + '/api'
connector = get_requests_connector(session=session, url=vapi_url)

# Add SAML token security context to the connector.
saml_token_context = create_saml_bearer_security_context(saml_token)
connector.set_security_context(saml_token_context)

# Create a stub configuration by using the SAML token security context.
my_stub_config = StubConfigurationFactory.new_std_configuration(connector)

# Create a Session stub with SAML token security context.
session_stub = Session(my_stub_config)

# Use the create operation to create an authenticated session.
session_id = session_stub.create()

# Create a session ID security context.
session_id_context = create_session_security_context(session_id)

# Update the stub configuration with the session ID security context.
my_stub_config.connector.set_security_context(session_id_context)
```

Create a vSphere Automation Session with User Credentials

With the vSphere Automation SDK for Python, you can create authenticated sessions by using only user credentials.

You connect to the vSphere Automation Endpoint by using a user name and password known to the vCenter Single Sign-On service. The vSphere Automation uses your credentials to authenticate with the vCenter Single Sign-On Service and obtain a SAML token.

Prerequisites

- Retrieve the vSphere Automation Endpoint URL from the Lookup Service.
- Verify that you have valid user credentials for the vCenter Single Sign-On identity store.

Procedure

1. Create a connection stub by specifying the vSphere Automation Endpoint URL and the message protocol to be used for the connection.
2. Create a stub configuration instance and set the specific security context to be used.

   The security context object uses the user name and password that are used for authenticating to the vCenter Single Sign-On service.
3. Create a `Session` stub that uses the stub configuration instance.
4. Call the create operation on the `Session` stub to create an authenticated session to the vSphere Automation Endpoint.

   The operation returns a session identifier.

5 Create a security context instance and add the session ID to it.

6 Update the stub configuration instance with the session security context.

What to do next

You can use the authenticated session to access vSphere Automation services. For more information about creating stubs to the vSphere Automation services, see Chapter 5 Accessing vSphere Automation Services.

Create a Web Services Session

To develop a complex workflow, you might need to send requests to vSphere Web Services running in your virtual environment. To achieve this, you access the vSphere Web Services API by using the Web Services endpoint.

The vSphere Web Services API also supports session-based access. To establish an authenticated session, you can send the SAML token retrieved from the vCenter Single Sign-On service to a vSphere Web Service. In return you receive a session identifier that you can use to access the service. For more information about accessing Web Services, see the *vSphere Web Services SDK Programming Guide* documentation.

The vSphere Automation SDK for Python supports a simplified way of creating connections to the Web Services API by using the `pyVim` library.

Prerequisites

- Retrieve the vSphere Web Services endpoint URL from the Lookup Service.
- Obtain a SAML token from the vCenter Single Sign-On service.

Procedure

1 Connect to the vSphere Web Services endpoint.

2 Send the SAML token to a specific Web service to create an authenticated session.

3 Add the retrieved session ID to the service content object.

   The Service Content object gives you access to several server-side managed objects that represent vSphere services and components.

Python Example of Creating a Web Services Session

This example is based on code in the `service_manager.py` sample file.

This example uses the following global variables.

- *my_ws_url*
- *my_sso_username*
- *my_sso_password*

The *my_ws_url* variable represents the URL of the vCenter Server Web Services API endpoint. You can retrieve the endpoint URL from the Lookup Service.

Note For a complete and up-to-date version of the sample code, see the vSphere Automation SDK Python samples at GitHub.

```
...

# Extract the hostname from the endpoint URL.
url_scheme, url_host, url_path, url_params, url_query, url_fragment = \
      urlparse(my_ws_url)
pattern = '(?P<host>[^:/ ]+).?(?P<port>[0-9]*)'
match = re.search(pattern, url_host)
host_name = match.group('host')

# Invoke the SmartConnect() method by supplying
# the host name, user name, and password.
service_instance_stub = SmartConnect(host=host_name,
                                     user=my_sso_username,
                                     pwd=my_sso_password)

# Retrieve the service content.
service_content = service_instance_stub.RetrieveContent()
```

Accessing vSphere Automation Services

5

vSphere Automation SDK provides mechanisms for creating remote stubs to give clients access to vSphere Automation services.

The sequence of tasks you must follow to create a remote stub starts with creating a `ProtocolFactory`. You use the protocol factory object to create a `ProtocolConnection`. Connection objects provide the basis for creating stub interfaces tovSphere Automation services.

When you establish a connection to the vSphere Automation Endpoint, you can create a `StubFactory` object and a `StubConfiguration` object. With these objects, you can create the remote stub for the vSphere Automation service that you want to access.

The complete connection sequence also includes SSL truststore support and a temporary `StubConfiguration` that you use for SAML token authentication and session creation.

For greater security, use an external utility to create a certificate store:

```
keytool –import –noprompt –trustcacerts \
–alias <alias name> \
–file <certificate file> \
–keystore <truststore filename> \
–storepass <truststore password>
```

This chapter includes the following topics:

- Access a vSphere Automation Service

Access a vSphere Automation Service

To access a vSphere Automation service, you must have a valid session connection. The sequence for accessing a vSphere Automation service includes creating a protocol connection object and using it to create the service stub.

Prerequisites

Establish a connection to the vSphere Automation Endpoint URL. For more information about the authentication mechanisms that you can use, see Chapter 4 Authentication Mechanisms.

Procedure

1. Create a protocol factory object.
2. Create a protocol connection object to access an API provider.

   The vSphere Automation API clients use `ApiProvider` instances to invoke operations on services running in the virtual environment. To invoke an operation, you must specify the target service and operation, input parameters, and execution context.
3. Create a `StubFactory` object by using the `ApiProvider` instance.
4. Create a `StubConfiguration` instance and set the security context to be used for the service stub.
5. Create the stub for the vSphere Automation service interface by calling the create method of the `StubFactory` instance. Pass the service class and the `StubConfiguration` instance as arguments.

Content Library Service

6

The Content Library Service provides means for managing content libraries in the context of a single or multiple vCenter Server instances deployed in your virtual environment. You can use the vSphere Automation APIs to access the Content Library Service through the vSphere Automation Endpoint.

Administrators can use content libraries to share VM templates, vApp templates, and other types of files across vCenter Server instances in the virtual environment. Sharing templates across your virtual environment promotes consistency, compliance, efficiency, and automation in deploying workloads at scale.

- Content Library Overview

  A content library instance represents a container for a set of library items. A content library item instance represents the logical object stored in the content library, which might be one or more usable files.

- Querying Content Libraries

  You can create queries to find libraries that match your criteria. You can also retrieve a list of all libraries or only the libraries of a specific type.

- Content Libraries

  The Content Library API provides services that allow you to create and manage content libraries programmatically. You can create a local library and publish it for the entire virtual environment. You can also subscribe to use the contents of a local library and enable automatic synchronization to ensure that you have the latest content.

- Library Items

  A library item groups multiple files within one logical unit. You can perform various tasks with the items in a content library.

Content Library Overview

A content library instance represents a container for a set of library items. A content library item instance represents the logical object stored in the content library, which might be one or more usable files.

- Content Library Types

  You can create two types of libraries, local and subscribed.

- Content Library Items

  Library items are VM templates, vApp templates, or other VMware objects that can be contained in a content library. VMs and vApps have several files, such as log files, disk files, memory files, and snapshot files that are part of a single library item. You can create library items in a specific local library or remove items from a local library. You can also upload files to an item in a local library so that the libraries subscribed to it can download the files to their NFS or SMB server, or datastore.

- Content Library Storage

  When you create a local library, you can store its contents on a datastore managed by the vCenter Server instance or on a remote file system.

Content Library Types

You can create two types of libraries, local and subscribed.

- You can create a local library as the source for content you want to save or share. Create the local library on a single vCenter Server instance. You can add items to a local library or remove them. You can publish a local library and as a result this content library service endpoint can be accessed by other vCenter Server instances in your virtual environment. When you publish a library, you can configure the authentication method, which a subscribed library must use to authenticate to it.
- You can create a subscribed library and populate its content by synchronizing to a local library. A subscribed library contains copies of the local library files or just the metadata of the library items. The local library can be located on the same vCenter Server instance as the subscribed library, or the subscribed library can reference a local library on a different vCenter Server instance. You cannot add library items to a subscribed library. You can only add items to the source library. After synchronization, both libraries will contain the same items.

Content Library Items

Library items are VM templates, vApp templates, or other VMware objects that can be contained in a content library. VMs and vApps have several files, such as log files, disk files, memory files, and snapshot files that are part of a single library item. You can create library items in a specific local library or remove items from a local library. You can also upload files to an item in a local

library so that the libraries subscribed to it can download the files to their NFS or SMB server, or datastore.

For information about the tasks that you can perform by using the content library service, see Content Libraries.

Content Library Storage

When you create a local library, you can store its contents on a datastore managed by the vCenter Server instance or on a remote file system.

Depending on the type of storage that you have, you can use Virtual Machine File System (VMFS) or Network File System (NFS) for storing content on a datastore.

For storing content on a remote file system, you can enter the path to the NFS storage that is mounted on the Linux file system of the vCenter Server Appliance. For example, you can use the following URI formats: `nfs://<server>/<path>?version=4` and `nfs://<server>/<path>`. If you have a vCenter Server instance that runs on a Windows machine, you can specify the Server Massage Block (SMB) URI to the Windows shared folders that store the library content. For example, you can use the following URI format: `smb://<unc-server>/<path>`.

Python Example of Storing Library Content on a Datastore

This example is based on the code in the `library_crud.py` sample file.

Note For a complete and up-to-date version of the sample code, see the vSphere Automation SDK Python samples at GitHub.

```
...

# Create a StorageBacking instance of datastore type.
library_backing = library_client.StorageBacking()
library_backing.type = library_client.StorageBacking.Type.DATASTORE

# Pass the value of the datastore managed object reference.
# The  vSphere
             Automation SDK for Python contains
# the GetDatastoreByName class, which sample resource is located in the
# /client/samples/src/com/vmware/vcloud/suite/sample/vim/helpers/ directory.
# You can use the utility to retrieve the managed object reference of the datastore entity.
library_backing.datastore_id = 'datastore-123'

# Create a LibraryModel that represents a local library backed on a datastore.
library_model = content_client.LibraryModel()
library_model.name = 'AcmeLibrary'
library_model.storage_backings = [library_backing]
```

Querying Content Libraries

You can create queries to find libraries that match your criteria. You can also retrieve a list of all libraries or only the libraries of a specific type.

- Listing All Content Libraries

  You can retrieve a list of all content library IDs in your virtual environment, regardless of their type, by using the `Library` service.

- Listing Content Libraries of a Specific Type

  You can use the vSphere Automation API to retrieve content libraries of a specific type. For example, you can list only the local libraries in your virtual environment.

- List Content Libraries by Using Specific Search Criteria

  You can filter the list of content libraries and retrieve only the libraries that match your specific criteria. For example, you might want to publish all local libraries with a specific name.

Listing All Content Libraries

You can retrieve a list of all content library IDs in your virtual environment, regardless of their type, by using the `Library` service.

You can use the `list` function to retrieve all local and subscribed libraries in your system.

Python Example of Retrieving a List of All Content Libraries

This example is based on the code in the `library_crud.py` sample file.

Note For a complete and up-to-date version of the sample code, see the vSphere Automation SDK Python samples at GitHub.

```
...

library_stub = content_client.Library(my_stub_config)
libraries = library_stub.list()
print('List of all library identifiers:')
for library_id in library_ids :
  library = library_stub.get(library_id)
  print('Library ID {}: {}'.format(library_id, library.name))
```

Listing Content Libraries of a Specific Type

You can use the vSphere Automation API to retrieve content libraries of a specific type. For example, you can list only the local libraries in your virtual environment.

If you want to retrieve only a list of the local libraries, you must retrieve the `LocalLibrary` service and use the `list` function on the `LocalLibrary` service. To list only subscribed libraries, you must retrieve the `SubscribedLibrary` service and call the `list` function on the `SubscribedLibrary` service.

List Content Libraries by Using Specific Search Criteria

You can filter the list of content libraries and retrieve only the libraries that match your specific criteria. For example, you might want to publish all local libraries with a specific name.

Prerequisites

Verify that you have access to the `Library` service.

Procedure

1. Create a `FindSpec` instance and specify your search criteria.
2. Call the `find` function on the `Library` service.

   All content libraries that match your search criteria are listed.

Python Example of Retrieving a List of All Local Libraries with a Specific Name

This example retrieves a list of all local libraries with the name **`AcmeLibrary`** that exist in your virtual environment.

Note For related code samples, see the vSphere Automation SDK Python samples at GitHub.

```
...

# Create a FindSpec object to specify your search criteria.
find_spec = content_client.Library.FindSpec()
find_spec.name = 'AcmeLibrary'
find_spec.type = content_client.LibraryModel.LibraryType.LOCAL

# Invoke the find() function by using the FindSpec instance.
library_stub = content_client.Library(my_stub_config)
library_ids = library_stub.find(find_spec)
```

Content Libraries

The Content Library API provides services that allow you to create and manage content libraries programmatically. You can create a local library and publish it for the entire virtual environment. You can also subscribe to use the contents of a local library and enable automatic synchronization to ensure that you have the latest content.

- Create a Local Content Library

  You can create a local content library programmatically by using the vSphere Automation API. The API allows you to populate the content library with VM and vApp templates. You can use these templates to deploy virtual machines or vApps in your virtual environment.

- Publish an Existing Content Library

  To make the library content available for other vCenter Server instances across the vSphere Automation environment, you must publish the library. Depending on your workflow, select a method for publishing the local library. You can publish a local library that already exists in your vSphere Automation environment.

- Publish a Library at the Time of Creation

  You can publish a local library at the time of creation to enable other libraries to subscribe and use the library content.

- Subscribe to a Content Library

  You can subscribe to public content libraries. The source objects for a public content library can be: a library created on a vCenter Server 6.0 instance, a catalog created on a vCloud Director 5.5 instance, or a third-party library. When you subscribe to a library, you must specify the backing storage for the library content. You must also provide the correct user name and password if the library requires basic authentication.

- Synchronize a Subscribed Content Library

  When you subscribe to a published library, you can configure the settings for downloading and updating the library content.

- Editing the Settings of a Content Library

  You can update the settings of content library types in your virtual environment by using the vSphere Automation API.

- Removing the Content of a Subscribed Library

  You can free storage space in your virtual environment by removing the subscribed library content that you no longer need.

- Delete a Content Library

  When you no longer need a content library, you can invoke the `delete` method on either the `LocalLibrary` or the `SubscribedLibrary` service depending on the library type.

Create a Local Content Library

You can create a local content library programmatically by using the vSphere Automation API. The API allows you to populate the content library with VM and vApp templates. You can use these templates to deploy virtual machines or vApps in your virtual environment.

Procedure

1. Create a `StorageBacking` instance and define the storage location.
2. Create a `LibraryModel` instance and set the properties of the new local library.
3. Access the `LocalLibrary` object which is part of the vSphere Automation API service interfaces.
4. Call the create function on the `LocalLibrary` object and pass the `LibraryModel` as a parameter.

Python Example of Creating a Local Content Library

This example creates a local library with name `AcmeLibrary` , which is stored on the local file system where vCenter Server runs.

Note For related code samples, see the vSphere Automation SDK Python samples at GitHub.

```
...

# 1 – Create a storage backing instance on a local file system.
library_backing = library_client.StorageBacking()
library_backing.type = library_client.StorageBacking.Type.OTHER
library_backing.storage_uri = 'file:///tmp'

# 2 – Create a Library model to specify properties of the new library.
library_model = content_client.LibraryModel()
library_model.type = content_client.LibraryModel.LibraryType.LOCAL
library_model.name = 'AcmeLibrary'
library_model.storage_backings = [library_backing]

# 3 – Call the create() method, passing the library model as a parameter.
idem_token = str(uuid.uuid4())
local_library_stub = content_client.LocalLibrary(my_stub_config)
library_id = local_library_stub.create(create_spec=library_model,
                                       client_token=idem_token)
```

Publish an Existing Content Library

To make the library content available for other vCenter Server instances across the vSphere Automation environment, you must publish the library. Depending on your workflow, select a method for publishing the local library. You can publish a local library that already exists in your vSphere Automation environment.

Procedure

1. Retrieve a reference to the `LocalLibrary` service.
2. Retrieve an existing local library by using the library ID.
3. Create a `PublishInfo` instance to define how the library is published.
4. Specify the authentication method to be used by a subscribed library to authenticate to the local library. You can enable either basic authentication or no authentication. Basic authentication requires a user name and password.
5. (Optional) If you publish the library with basic authentication, you must specify a user name and password for the `PublishInfo` instance, which must be used for authentication.

   Important Use the predefined user name **vcsp** or leave the user name undefined. You must set only a password to protect the library.
6. Specify that the library is published.

7 Use the retrieved local library to configure it with the `PublishInfo` instance.

8 Update the properties of the `LibraryModel` object returned for the local library.

Python Example of Publishing an Existing Content Library

This example is based on the code in the `library_publish_subscribe.py` sample file.

Note For a complete and up-to-date version of the sample code, see the vSphere Automation SDK Python samples at GitHub.

```
...

# Retrieve an existing local library.
local_library_stub = content_client.LocalLibrary(my_stub_config)
local_library = local_library_stub.get(my_library_id)

# Specify how the local library is published, using BASIC authentication.
publish_info = library_client.PublishInfo()
publish_info.user_name = 'vcsp' # Can omit this value; if specified, it must be 'vcsp'.
publish_info.password = 'password'
publish_info.authentication_method = library_client.PublishInfo.AuthenticationMethod.BASIC
publish_info.published = True

# Update the LibraryModel object retieved in step 1
# and configure it with the PublishInfo object.
local_library.publish_info = publish_info

# Use the LibraryModel object to update the library instance.
local_library_stub.update(library_id=my_library_id,
update_spec=local_library)
```

Publish a Library at the Time of Creation

You can publish a local library at the time of creation to enable other libraries to subscribe and use the library content.

Procedure

1 Retrieve the `LocalLibrary` service.

2 Create a `PublishInfo` instance to define how the library is published.

3 Specify the authentication method to be used by a subscribed library to authenticate to the local library.

You can enable either basic authentication or no authentication on the library. Basic authentication requires a user name and password.

4 (Optional) If you publish the library with basic authentication, you must specify a user name and password for the `PublishInfo` instance, which must be used for authentication.

Important Use the predefined user name **vcsp** or leave the user name undefined. You must set only a password to protect the library.

5 Create a `LibraryModel` instance and configure the instance.

6 Set the library type to local and use the configured `PublishInfo` instance to set the library to published.

7 Define where the content of the local library is stored by using the `StorageBacking` class.

8 Create a published local library.

Subscribe to a Content Library

You can subscribe to public content libraries. The source objects for a public content library can be: a library created on a vCenter Server 6.0 instance, a catalog created on a vCloud Director 5.5 instance, or a third-party library. When you subscribe to a library, you must specify the backing storage for the library content. You must also provide the correct user name and password if the library requires basic authentication.

Note If you subscribe to libraries created with basic authentication on a vCenter Serverinstance, make sure that you pass **vcsp** as an argument for the user name.

Procedure

1 Create a `StorageBacking` instance to define the location where the content of the subscribed library is stored.

2 Create a `SubscriptionInfo` instance to define the subscription behavior of the library.

   a Provide the mechanism to be used by the subscribed library to authenticate to the published library.

   You can choose between no authentication and basic authentication depending on the settings of the published library you subscribe to. If the library is published with basic authentication, you must set basic authentication in the `SubscriptionInfo` instance. Set the user name and the password of the `SubscriptionInfo` instance to match the credentials of the published library.

   b Provide the URL to the endpoint where the metadata of the published library is stored.

c Define the synchronization mechanism of the subscribed library.

You can choose between two synchronization modes: automatic and on demand. If you enable automatic synchronization for a subscribed library, both the content and the metadata are synchronized with the published library. To save storage space, you can synchronize the subscribed library on demand and update only the metadata for the published library content.

d Set the thumbprint that is used for validating the certificate of the published library.

3 Create a `LibraryModel` instance and set the library type to subscribed (`LibraryModel.LibraryType.SUBSCRIBED`).

4 Access the `SubscribedLibrary` service and create the subscribed library instance.

Python Example of Subscribing to a Published Library

This example is based on the code in the `library_publish_subscribe.py` sample file.

Note For a complete and up-to-date version of the sample code, see the vSphere Automation SDK Python samples at GitHub.

```
...

# Create a StorageBacking instance on a local file system.
library_backing = library_client.StorageBacking()
library_backing.type = library_client.StorageBacking.Type.OTHER
library_backing.storage_uri = '/mnt/nfs/cls-root'

# Create a new SubscriptionInfo object to describe the subscription behavior.
subscription_info = library_client.SubscriptionInfo()
subscription_info.authentication_method = library_client.SubscriptionInfo.AuthenticationMethod.BASIC
subscription_info.user_name = 'libraryUser'
subscription_info.password = 'password'
subscription_info.subscription_url = 'https://www.acmecompary.com/library_inventory/lib.json'
subscription_info.automatic_sync_enabled = True
subscription_info.ssl_thumbprint = '9B:00:3F:C4:4E:B1:F3:F9:0D:70:47:48:E7:0B:D1:A7:0E:DE:60:A5'

# Create a new LibraryModel object for the subscribed library.
library_model = content_client.LibraryModel()
library_model.type = content_client.LibraryModel.LibraryType.SUBSCRIBED
library_model.name = 'subscrLibrary'

# Attach the storage backing and the subscription info to the library model.
library_model.storage_backings = [library_backing]
library_model.subscription_info = subscription_info

# Create the new library instance.
idem_token = str(uuid.uuid4())
local_library_stub = content_client.LocalLibrary(my_stub_config)
library_id = local_library_stub.create(create_spec=library_model,client_token=idem_token)
```

Synchronize a Subscribed Content Library

When you subscribe to a published library, you can configure the settings for downloading and updating the library content.

- You can enable automatic synchronization of the subscribed library and download a copy of the content of the local library immediately.
- You can save storage space and download only the metadata for the items that are part of the local library.

To ensure that your subscribed library contains the most recent published library content, you can force a synchronization task.

Procedure

1. Access the `SubscribedLibrary` vSphere Automation service.
2. Retrieve the subscribed library ID from the `SubscribedLibrary` service.
3. Force the synchronization of the subscribed library.

Results

The synchronization operation depends on the update settings of the subscribed library. If the subscribed library is configured to update only on demand, only the metadata of the library items will be synchronized.

Editing the Settings of a Content Library

You can update the settings of content library types in your virtual environment by using the vSphere Automation API.

Table 6-1. Options for Updating a Content Library

Content Library Types	Option

Local content library	You can change the settings of a local library before calling the update function on the `LocalLibrary` object:
	■ Before a library is published, you can edit the following settings:
	  ■ The name of a local library that is retrieved by using the `LocalLibrary` object
	  ■ The human-readable description of a local library retrieved by using the `LocalLibrary` object
	■ After a library is published, you must first retrieve the `PublishInfo` instance of the published library you want. You can use the instance to configure the following settings.
	  ■ Unpublish the local library.
	  ■ Change the authentication method of the library.
	  ■ Change the password that must be used for authentication.

Subscribed content library	You can edit the settings of a subscribed library if you retrieve the `SubscriptionInfo` instance associated with it. To apply the changes, you must update the library by using the `SubscribedLibrary` object. You can configure the following settings: ■ The authentication method required by the local library ■ The user name and password of the subscribed library for authentication to the local library ■ The method for synchronizing the metadata and the content of the subscribed library ■ The thumbprint used for validating the SSL certificate of the local library

Removing the Content of a Subscribed Library

You can free storage space in your virtual environment by removing the subscribed library content that you no longer need.

You can create a subscribed library with the option to download the library content on demand. As a result, only the metadata for the library items is stored in the associated with the subscribed library storage. When you want to deploy a virtual machine from a VM temple in the subscribed library, you must synchronize the subscribed library to download the entire published library content. When you no longer need the VM template, you can call the `evict` function on the `SubscribedLibrary` service. You must provide the subscribed library ID to this function. As a result, the subscribed library content that is cached on the backing storage is deleted.

If the subscribed library is not configured to synchronize on demand, an exception is thrown. In this case the subscribed library always attempts to have the most recent published library content.

Delete a Content Library

When you no longer need a content library, you can invoke the `delete` method on either the `LocalLibrary` or the `SubscribedLibrary` service depending on the library type.

Procedure

1. Access the `SubscribedLibrary` or the `LocalLibrary` service by using the vSphere Automation Endpoint.
2. Retrieve the library ID you want to delete.
3. Call the `delete` function on the library service and pass the library ID as argument.

   All library items cashed on the storage backing are removed asynchronously. If this operation fails, you must manually remove the content of the library.

Library Items

A library item groups multiple files within one logical unit. You can perform various tasks with the items in a content library.

You can upload files to a library item in a local library and update existing items. You can download the content of a library item from a subscribed library and use the item, for example, to deploy a virtual machine. You can remove the content of a library item from a subscribed library to free storage space and keep only the metadata of the library item. When you no longer need local library items, you can delete them and they are removed from the subscribed library when a synchronization task is completed.

You can create a library item from a specific item type, for example **`.ovf`**. The Content Library service must support the library item type to handle the item correctly. If no support is provided for a specified type, the Content Library service handles the library item in the default way, without adding metadata to the library item or guiding the upload process.

- Create an Empty Library Item

  You can create as many library items as needed and associate them with a local content library.
- Querying Library Items

  You can perform numerous query operations on library items.
- Edit the Settings of a Library Item

  You can edit the name, description, and type of a library item.
- Upload a File from a Local System to a Library Item

  You can upload different types of files from a local system to a library item that you want to use in the vSphere Automation environment.

- Upload a File from a URL to a Library Item

  You can upload different types of files from a local system to a library item that you want to use in the vSphere Automationenvironment.

- Download Files to a Local System from a Library Item

  You might want to download files to a local system from a library item and then make changes to the files before you use them.

- Synchronizing a Library Item in a Subscribed Content Library

  The items in a subscribed library have features that are distinct from the items in a local library. Synchronizing the content and the metadata of an item in a subscribed library depends on the synchronization mechanism of the subscribed library.

- Removing the Content of a Library Item

  You can remove the content from a library item to free space on your storage.

- Deleting a Library Item

  You can remove a library item from a local library when you no longer need it.

Create an Empty Library Item

You can create as many library items as needed and associate them with a local content library.

Procedure

1. Access the `Item` service by using the vSphere Automation endpoint.
2. Instantiate the `ItemModel` class.
3. Define the settings of the new library item.
4. Associate the library item with an existing local library.
5. Invoke the `create` function on the `Item` object to pass the library item specification and the unique client token.

What to do next

Upload content to the new library item. See Upload a File from a Local System to a Library Item and Upload a File from a URL to a Library Item.

Python Example of Creating a Library Item

This example shows how to create an empty library item that stores an ISO image file.

Note For related code samples, see the vSphere Automation SDK Python samples at GitHub.

```
...

# 1 – Create an instance of the ItemModel class and specify the item settings.
item_model = library_client.ItemModel()
item_model.name = 'ESXi ISO image'
```

```
item_model.description = 'ISO image with the latest security patches for ESXi'
item_model.type = 'iso'

# 2 - Associate the new item with an existing library.
item_model.library_id = my_library_id

# 3 - Create the new instance of the Item class, using the specified model.
idem_token = str(uuid.uuid4())
item_stub = library_client.Item(my_stub_config)
item_id = item_stub.create(create_spec=item_model, client_token=idem_token)
```

Querying Library Items

You can perform numerous query operations on library items.

You can retrieve a list of all items in a library, retrieve a library item that has a specific type or name, and find a library item that is not cached on the disk. You can then update the library item content from the subscribed library.

List Library Items

You can use the `list` method of the `Item` object to retrieve a list of all items in a particular library.

Prerequisites

Verify that you have access to the `Item` service.

Procedure

1. Retrieve the ID of the content library whose items you want to list.
2. List the items of the specific library.
3. Retrieve a list of the files that belong to a library item.

Example

You can see an example query operation in the code example for Edit the Settings of a Library Item. The beginning of the example lists the items of a published library and prints a list with the names and size of each file in the listed items.

List Library Items That Match Specific Criteria

You can filter the items contained in a library and retrieve only the items matching specific criteria. For example, you might want to remove or update only specific items in a library.

Prerequisites

Verify that you have access to the `Item` service.

Procedure

1. Create an instance in the `FindSpec` class.
2. Specify the filter properties by using the `FindSpec` instance.

3 List the items matching the specified filter.

Results

A list of items matching the filter criteria is created as a result.

Edit the Settings of a Library Item

You can edit the name, description, and type of a library item.

Prerequisites

Verify that you have access to the `Item` service.

Procedure

1 Retrieve the item that you want to update.

2 Create an `ItemModel` instance.

3 Change the human-readable name and description of the library item.

4 Update the library item with the configured item model.

Python Example of Changing the Settings for a Library Item

This example shows how to find an item by using the item name and then how to change the name and description of the retrieved item.

Note For related code samples, see the vSphere Automation SDK Python samples at GitHub.

```
...

# 1 – List the items in a published library.
item_stub = library_client.Item(my_stub_config)
item_ids = item_stub.list(my_library_id)

# 2 – List the files uploaded to each library item and print their names and sizes.
file_stub = item_client.File(my_stub_config)
for item_id in item_ids :
  item = item_stub.get(item_id)
  file_infos = file_stub.list(item_id)
  for file_info in file_infos :
    print('Library item {} has file {} with size {}'.format(item.name, file_info.name,
file_info.size))

# 3 – For a library item with a specified name,
#     create an ItemModel to change the name and description of the library item.
if item.name == 'simpleVmTemplate' :
  print('Library item {} with description {}'.format(item.name, item.description))
  item_model = library_client.ItemModel()
  item_model.name = 'newItemName'
  item_model.description = 'Description of the newItemName'
  item_stub.update(library_item_id=item_id,
                   update_spec=item_model)
```

```
print('has been changed to:')
print('library item {} with description {}'.format(item_model.name, item_model.description))
```

Upload a File from a Local System to a Library Item

You can upload different types of files from a local system to a library item that you want to use in the vSphere Automation environment.

Prerequisites

- Create an empty library item. See Create an Empty Library Item.
- Verify that you have access to the `UpdateSession` and `File` services.

Procedure

1. Create an `UpdateSessionModel` instance to track the changes that you make to the library item.
2. Create an update session by using the `UpdateSession` service.
3. Create an `AddSpec` instance to describe the upload method and other properties of the file to be uploaded.
4. Create the request for changing the item by using the `File` service.
5. Upload the file that is on the local system.
6. Complete and delete the update session to apply the changes to the library item.

Python Example of Uploading Files to a Library Item from a Local System

This example shows how to upload an ISO image file from the local system to a library item.

Note For related code samples, see the vSphere Automation SDK Python samples at GitHub.

```
...

# 1 – Create an instance of the ItemModel class and specify the item settings.
item_model = library_client.ItemModel()
item_model.name = 'ESXi patches'
item_model.description = 'ESXi security patches'
item_model.type = 'iso'
item_model.library_id = my_library_id
idem_token = str(uuid.uuid4())
item_stub = library_client.Item(my_stub_config)
item_id = item_stub.create(create_spec=item_model,
client_token=idem_token)

# 2 – Create an UpdateSessionModel instance to track the changes you make to the item.
update_session_model = item_client.UpdateSessionModel()
update_session_model.library_item_id = item_id

# 3 – Create an update session from the model.
idem_token = str(uuid.uuid4())
update_session_stub = update_session_client.UpdateSession(my_stub_config)
```

```
session_id = update_session_stub.create(create_spec=update_session_model
client_token=idem_token)

try :
  # 4 – Create a new AddSpec instance to describe the properties of the file to be uploaded.
  file_spec = update_session_client.AddSpec()
  file_spec.name = 'ESXi patch'
  file_spec.source_type = update_session_client.File.SourceType.PUSH

  # 5 – Link the ISO file spec to the update session.
  update_file_stub = update_session_stub.File(my_stub_config)
  file_info = update_file_stub.File.add(update_session_id=session_id,
                                        file_spec=file_spec)
  # 6 – Use HTTP library to upload the file to the library item.
  upload_uri = file_info.upload_endpoint.uri
  file_name = "/updates/esxi/esxi_patch.iso"
  host = urlparse.urlsplit(upload_uri)
  connection = httplib.HTTPConnection(host.netloc)
  with open(file_name, "rb") as f :
  connection.request("PUT", upload_uri, f)

  # 7 – Commit the updates.
  library_item_service.UpdateSession.complete(session_id)

  finally :
  # 8 – Delete the session.
  library_item_service.UpdateSession.delete(session_id)
```

Upload a File from a URL to a Library Item

You can upload different types of files from a local system to a library item that you want to use in the vSphere Automationenvironment.

Prerequisites

- Create an empty library item. See Create an Empty Library Item.
- Verify that you have access to the `UpdateSession` and `File` services.

Procedure

1. Create an `UpdateSessionModel` instance to track the changes that you make to the library item.
2. Create an update session by using the `UpdateSession` service.
3. Create a file specification to describe the upload method and other properties of the file to be uploaded.
4. Specify the location of the file to be uploaded by creating a `TransferEndpoint` instance.
5. Add the file source endpoint to the file specification.
6. Create a request for changing the item by using the configured file specification.
7. Complete the update session to apply the changes to the library item.

Python Example of Uploading a File from a URL to a Library Item

This example shows how to upload a file from a URL to a library item. The example is based on the code in the `cls_api_helper.py` sample file.

Note For a complete and up-to-date version of the sample code, see the vSphere Automation SDK Python samples at GitHub.

```
...

# 1 – Create a new library item to hold the uploaded file.
item_model = library_client.ItemModel()
item_model.name = 'ESXi patches'
item_model.description = 'ESXi security patches'
item_model.type = 'iso'
item_model.library_id = my_library_id
idem_token = str(uuid.uuid4())
item_stub = library_client.Item(my_stub_config)
item_id = item_stub.create(create_spec=item_model, client_token=idem_token)

# 2 – Create an UpdateSessionModel instance to track the changes you make to the item.
update_session_model = item_client.UpdateSessionModel()
update_session_model.library_item_id = item_id

# 3 – Create an update session from the model.
idem_token = str(uuid.uuid4())
update_session_stub = update_session_client.UpdateSession(my_stub_config)
session_id = update_session_stub.create(create_spec=update_session_model, client_token=idem_token)

try :
  # 4 – Create a new AddSpec instance to describe the properties of the file to be uploaded.
  file_spec = update_session_client.AddSpec()
  file_spec.name = 'ESXi patch'
  file_spec.source_type = update_session_client.File.SourceType.PULL

  # 5 – Specify the location from which the file is to be uploadod to the library item.
  endpoint = item_client.TransferEndpoint()
  endpoint.uri = 'http://www.example.com/patches_ESXi65/ESXi_patch.iso'
  file_spec.source_endpoint = endpoint

  # 6 – Link the file specification to the update session.
  update_file_stub = update_session_client.File(my_stub_config)
  update_file_stub.File.add(update_session_id=session_id, file_spec=file_spec)

  # 7 – Mark session as completed, to initiate the asynchronous transfer.
  update_session_stub.complete(session_id)
```

Download Files to a Local System from a Library Item

You might want to download files to a local system from a library item and then make changes to the files before you use them.

Procedure

1. Create a download session model to specify the item, which contains the file that you want to download.
2. Access the `File` service and retrieve the file that you want to export to your system within the new download session.
3. Prepare the files that you want to download and wait until the files are in the prepared state.
4. Retrieve the download endpoint URI of the files.
5. Download the files with an HTTP GET request.
6. Delete the download session after all files are downloaded.

Python Example of Downloading Files from a Library Item to Your Local System

This example uses the code in the `cls_api_helper.py` sample file.

Note For a complete and up-to-date version of the sample code, see the vSphere Automation SDK Python samples at GitHub.

```
...

# 1 – Create a new download session model.
download_session_model = item_client.DownloadSessionModel()
download_session_model.library_item_id = my_library_item_id
download_session_stub = item_client.DownloadSession(my_stub_config)
idem_token = str(uuid.uuid4())
download_session_id = download_session_stub.create(create_spec=download_session_model,
client_token=idem_token)

# 2 – Access the File service and retrieve the files you want to export.
download_session_file_stub = download_session_client.File(my_stub_config)
file_infos = download_session_file_stub.list(download_session_id)
for file_info in file_infos :
  download_session_file_stub.prepare(download_session_id, file_info.name)

# 3 – Wait until the file is in the prepared state before downloading.
  download_info = download_session_file_stub.get(download_session_id, file_info.name)
  while (DownloadSessionFile.PrepareStatus.PREPARED != download_info.status) :
  time.sleep(30)

# 4 – Download the file with an HTTP GET request.
  response = urllib2.urlopen(download_info.download_endpoint.uri)
  file_path = os.path.join(my_directory, file_info.name)
  with open(file_path, 'wb') as f :
  f.write(response.read())

# 5 – Delete the download session after all files are downloaded.
download_session_stub.delete(download_session_id)
```

Synchronizing a Library Item in a Subscribed Content Library

The items in a subscribed library have features that are distinct from the items in a local library. Synchronizing the content and the metadata of an item in a subscribed library depends on the synchronization mechanism of the subscribed library.

Table 6-2. Options for Synchronizing a Library Item

Synchronization Type of the Subscribed Library	Description
Synchronized on demand	If the subscribed library is synchronized on demand, you can use the `sync` method on the `SubscribedItem` service and pass as arguments the library item ID and `true`. When you perform the task, both the item metadata and the content are synchronized. To synchronize only the metadata of the item, pass the library ID and `false` as arguments to the method.
Not synchronized on demand	If the subscribed library is not synchronized on demand, you can use the `sync` method on the `SubscribedItem` service and pass as argument the item ID. In this case, the content of the item is always synchronized and the Boolean value is ignored when the call is executed.
Synchronized automatically	If the subscribed library is synchronized automatically, you can also use the `sync` method to force the synchronization of an item. Method execution depends on whether the subscribed library is synchronized on demand.

Removing the Content of a Library Item

You can remove the content from a library item to free space on your storage.

If you create a subscribed library with the option to synchronize library content on demand, only the metadata for the library items is stored. When you want to use the items in the library, you must force synchronization on the items to download their content. When you no longer need the files in an item, you can remove the cached content of the library item and free storage space. To achieve this task use the `evict` function of the `SubscribedItem` object.

Deleting a Library Item

You can remove a library item from a local library when you no longer need it.

To remove a library item from a library, you can call the `delete` method on the `Item` object and pass the library item ID as an argument. The item content is asynchronously removed from the storage.

You cannot remove items from a subscribed library. If you remove an item from a local library, the item is removed from the subscribed library when you perform a synchronization task on the subscribed library item.

Content Library Support for OVF Packages 7

Open Virtualization Format (OVF) is an industry standard that describes metadata about a virtual machine image in an XML format. An OVF package includes an XML descriptor file and optionally disk images, resource files (such as ISO files), manifest files, and certificate files.

With the vSphere Automation API, you can use the virtual machine (VM) and vApp templates from an OVF package in a content library to deploy VMs and virtual appliances on hosts, resource pools, and clusters. You can also use the API to create OVF packages in content libraries from virtual appliances and VMs on hosts, resource pools, and clusters.

When you create library items to store OVF packages, you must set the item type to `ovf`. You can use the objects and methods provided by the Content Library API to manage OVF packages. To comply with the specific standards of the OVF packages, the vSphere Automation API provides the `LibraryItem` class.

This chapter includes the following topics:

- Using the Content Library Service to Handle OVF Packages
- Using the LibraryItem Service to Execute OVF-Specific Tasks

Using the Content Library Service to Handle OVF Packages

You can upload an OVF package to a library item by using the `UpdateSession` interface. The location of the OVF package determines whether you can pull the content from a URL or push the content directly to a content library.

For information about uploading content to library items, see Upload a File from a Local System to a Library Item and Upload a File from a URL to a Library Item.

To download the files that are included in an OVF package to your local file system, use the `DownloadSession` interface. For more information, see Download Files to a Local System from a Library Item .

Upload an OVF Package from a URL to a Library Item

You can upload an OVF package from a Web server to a library item.

Procedure

1 Create an empty library item.

2 Create an update session object.

3 Create an `AddSpec` object to describe the properties and the upload location of the descriptor file of the OVF package.

4 Link the `AddSpec` object to the update session.

   All files that are included in the OVF package are automatically uploaded.

5 Complete the asynchronous transfer.

Python Example of Uploading an OVF Package from a URL to a Library Item

This example is based on the `ovf_import_export.py` sample file.

Note For a complete and up-to-date version of the sample code, see the vSphere Automation SDK Python samples at GitHub.

```
...

# 1 – Create a empty library item to describe the virtual machine.
item_model = library_client.ItemModel()
item_model.name = “ubuntu-vm”
item_model.description = “ubuntu 7.0”
item_model.library_id = my_library_id
item_model.type = “ovf”
client_token = str(uuid.uuid4())
item_stub = library_client.Item(my_stub_config)
item_id = item_stub.create(create_spec=item_model,
client_token=client_token)

# 2 – Create an update session.
update_session_model = item_client.UpdateSessionModel()
update_session_model.library_item_id = item_id
client_token = str(uuid.uuid4())
update_session_stub = update_session_client.UpdateSession(my_stub_config)
session_id = update_session_stub.create(create_spec=update_session_model,
client_token=client_token)

# 3 – Create a file specification for the OVF envelope file.
file_spec = update_session_client.AddSpec()
file_spec.name = “ubuntu.ovf”
file_spec.source_type = File.SourceType.PULL
endpoint = item_client.TransferEndpoint()
endpoint.uri = “http://www.example.com/images/ubuntu.ovf”
file_spec.source_endpoint = endpoint

# 4 – Link the file specification to the update session.
update_file_stub = update_session_client.File(my_stub_config)
update_file_stub.File.add(update_session_id=session_id,
file_spec=file_spec)
```

```
# 5 – Initiate the asynchronous transfer.
update_session_stub.complete(session_id)
```

Upload an OVF Package from a Local File System to a Library Item

You can upload an OVF package from a local file system. This procedure describes how to use the `AddSpec` object after you have created a library item and initiated an update session.

Procedure

1. Create a library item.
2. Create an update session.
3. Create an `AddSpec` object to describe the properties and the upload location of the descriptor file of the OVF package.
4. Link the `AddSpec` object to the update session.
5. Create an `AddSpec` object for each VMDK file included in the OVF package.
6. Add all `AddSpec` objects to the update session.

   Steps 5 and 6 must be repeated for each VMDK file included in the OVF package.
7. Initiate the upload operation.
8. Complete the update session.
9. Delete the session.

Python Example of Uploading an OVF Package to a Library Item from Your Local File System

This example is based on the `ovf_import_export.py` sample file.

Note For a complete and up-to-date version of the sample code, see the vSphere Automation SDK Python samples at GitHub.

```
...

# 1 – Create an empty library item to describe the VM/VApp.
item_model = library_client.ItemModel()
item_model.name = “ubuntu-vm”
item_model.description = “ubuntu 7.0”
item_model.library_id = my_library_id
item_model.type = “ovf”
client_token = str(uuid.uuid4())
item_stub = library_client.Item(my_stub_config)
item_id = item_stub.create(create_spec=item_model, client_token=client_token)

# 2 – Create an update session.
update_session_model = item_client.UpdateSessionModel()
update_session_model.library_item_id = item_id
```

```
client_token = str(uuid.uuid4())
update_session_stub = update_session_client.UpdateSession(my_stub_config)
session_id = update_session_stub.create(create_spec=update_session_model, client_token=client_token)

try :
  # 3 – Create a file spec for the OVF envelope file.
  file_spec = update_session_client.AddSpec()
  file_spec.name = “ubuntu.ovf”
  file_spec.source_type = update_session_client.File.SourceType.PUSH

  # 4 – Link the OVF file spec to the update session.
  update_file_stub = update_session_client.File(my_stub_config)
  file_info = update_file_stub.File.add(update_session_id=session_id, file_spec=file_spec)
  upload_uri = file_info.upload_endpoint.uri

  # 5 – Use HTTP library to push the file, out of band.
  file_name = “/medias/vms/ubuntu.ovf”
  host = urlparse.urlsplit(upload_uri)
  connection = httplib.HTTPConnection(host.netloc)
  with open(file_name, “rb”) as f :
    connection.request(“PUT”, upload_uri, f)

  # 6 – Create a file spec for the VMDK file.
  file_spec = update_session_client.AddSpec()
  file_spec.name = “ubuntu_disk.vmdk”
  file_spec.source_type = File.SourceType.PUSH

  # 7 – Add the VMDK file spec to the update session.
  file_info = update_file_stub.File.add(update_session_id=session_id, file_spec=file_spec)
  upload_uri = file_info.upload_endpoint().uri

  # 8 – Use HTTP library to push the file.
  file_name = “/medias/storage/ubuntu_disk.vmdk”
  host = urlparse.urlsplit(upload_uri)
  connection = httplib.HTTPConnection(host.netloc)
  with open(file_name, “rb”) as f :
    connection.request(“PUT”, upload_uri, f)

  # 9 – Commit the updates.
  update_session_stub.complete(session_id)

finally :
  # 10 – Delete the session.
  update_session_stub.delete(session_id)
```

Using the LibraryItem Service to Execute OVF-Specific Tasks

You can deploy virtual machines and vApps on hosts, clusters, and resource pools in your environment. You use the VM templates and vApp templates from an OVF package that is stored as a content library item.

With the vSphere Automation API, you can use the `LibraryItem` service to deploy virtual machines and virtual applications from library items that contain OVF packages. You can also use the `LibraryItem` vSphere Automation service to create library items from existing virtual machines and virtual appliances.

Deploy a Virtual Machine or Virtual Appliance from an OVF Package in a Content Library

You can use the `LibraryItem` service to deploy a virtual machine or virtual appliance on a host, cluster, or resource pool from a library item.

Procedure

1. Create a `DeploymentTarget` instance to specify the deployment location of the virtual machine or virtual appliance.
2. Instantiate the `ResourcePoolDeploymentSpec` class to define all necessary parameters for the deployment operation.

   For example, you can assign a name for the deployed virtual machine or virtual appliance, and accept the End User License Agreements (EULAs) to complete the deployment successfully.
3. (Optional) Retrieve information from the descriptor file of the OVF package and use the information during the OVF package deployment.
4. Invoke the `deploy` method on the `LibraryItem` service.
5. Verify the outcome of the deployment operation.

Python Example of Deploying a Virtual Machine from a Library Item on a Resource Pool

This example is based on the `deploy_ovf_template.py` sample file.

Note For a complete and up-to-date version of the sample code, see the vSphere Automation SDK Python samples at GitHub.

```
...

# Create a VM deployment specification to accept any network resource.
deployment_spec = ovf_client.LibraryItem.ResourcePoolDeploymentSpec()
deployment_spec.accept_all_eula = True

# Create deployment target spec to accept any resource pool.
target_spec = ovf_client.LibraryItem.DeploymentTarget()

# Initiate synchronous deployment operation.
item_stub = ovf_client.LibraryItem(my_stub_config)
result = item_stub.deploy(my_library_item_id,
                          target_spec,
                          deployment_spec,
```

```
                            client_token=str(uuid.uuid4())

# Verify deployment status.
print("Resource Type={} (ID={}) status:".format(result.resource_id.type, result.resource_id.id))
if result.succeeded == True :
  print("Resource instantiated.")
else :
  print("Instantiation failed.")
if result.error is not None :
  for error in result.error.errors :
    print("Error {}".format(error.message))
  if len(result.error.warnings) > 0 :
    print("Warnings:")
    for warning in result.error.warnings :
      print("{}".format(warning.message))
  if len(result.error.information) > 0 :
    print("Messages:")
    for info in result.error.information :
      for message in info.messages :
        print("{}".format(message))
```

Create an OVF Package in a Content Library from a Virtual Machine

You can create library items from existing virtual machines or virtual appliances. Use those library items later to deploy virtual machines and virtual appliances on hosts and clusters in the vSphere Automation environment.

Procedure

1. Create a `DeployableIdentity` instance to specify the source virtual machine or virtual appliance to be captured in an OVF package.
2. Create a `CreateTarget` instance to identify the content library where the OVF package is stored.
3. Create a `CreateSpec` instance to specify the properties of the OVF package.
4. Initiate a synchronous create operation by invoking the `create` function of the `LibraryItem` service.
5. Verify the results of the `create` operation.

Python Example of Creating an OVF Package in a Content Library from a Virtual Machine

This example shows how to capture a virtual machine in an OVF package and store the files in a new library item in a specified library.

Note For related code samples, see the vSphere Automation SDK Python samples at GitHub.

```
...

# Specify the resource to be captured.
```

```
deployable_identity = ovf_client.LibraryItem.DeployableIdentity();
deployable_identity.type = “VirtualMachine”
deployable_identity.id = “vm-32”

# Create a target spec to identify a library to hold the new item.
create_target = ovf_client.LibraryItem.CreateTarget()
create_target.library_id = my_library_id

# Specify OVF properties.
create_spec = ovf_client.LibraryItem.CreateSpec()
create_spec.name = “snap-32”
create_spec.description = “Snapshot of VM-32”

# Initiate synchronous capture operation.
lib_item_stub = ovf_client.LibraryItem(my_stub_config)
client_token = str(uuid.uuid4())
result = lib_item_stub.create(source=deployable_identity,
                              target=create_target,
                              create_spec=create_spec,
                              client_token=client_token)

# Verify capture status.
print("Resource Type={} (ID={}) status:".format(deployable_identity.type, deployable_identity.id))

if result.succeeded == True :
  print("Resource captured.")
else :
  print("Capture failed.")
if result.error is not None :
  for error in result.error.errors :
    print("Error {}".format(error.message))
  if len(result.error.warnings) > 0 :
    print("Warnings:")
    for warning in result.error.warnings :
      print("{}".format(warning.message))
  if len(result.error.information) > 0 :
    print("Messages:")
    for info in result.error.information :
      for message in info.messages :
        print("{}".format(message))
```

Tagging Service

8

The vSphere Automation Tagging Service supports the definition of tags that you can associate with vSphere objects or vSphere Automation resources. vSphere has a tagging feature but no public API to manage tags. With the vSphere Automation SDK, you can manage tags programmatically.

For example, to tag your VMs by guest operating system type, you might create a category called **`operating system`**, and specify that it applies to VMs only. You might also specify that only a single tag can be applied to a VM at any time. The tags in this category might be **`Windows`**, **`Linux`**, and **`Mac OS`**.

Creating Tags

When you create a tag, you create a tag category and create a tag under the category. After you create the tag, you can associate the tag with an object.

Tags and categories have global scope. The Platform Services Controller stores tags and categories makes them available to any vCenter Server system that is registered with the Platform Services Controller.

- Creating a Tag Category

  You create tags in the context of a tag category. You must create a category before you can add tags within that category.

- Creating a Tag

  After you create a tag category, you can create tags within that category

Creating a Tag Category

You create tags in the context of a tag category. You must create a category before you can add tags within that category.

A tag category has the following properties:

- name
- description
- cardinality, or how many tags it can contain
- the types of elements to which the tags can be assigned

You can associate tags with both vSphere API managed objects and VMware vSphere Automation API resources.

Python Example of Creating a Tag Category

This example is based on code in the `tagging_workflow.py` sample file.

The category `create()` function returns an identifier that you use when you create a tag for that category. The empty set for the associable types indicates that any object type can be associated with a tag in this category.

Note For a complete and up-to-date version of the sample code, see the vSphere Automation SDK Python samples at GitHub.

```
...

category_stub = tagging_client.Category(my_stub_config)

# Set up a tag category create spec.
tc_create_spec = category_stub.CreateSpec(name = 'favorites',
                                          description = 'My favorite virtual machines',
                                          cardinality = CategoryModel.Cardinality.MULTIPLE,
                                          associable_types = set())

# Create the tag category.
fav_category_id = category_stub.create(create_spec)
```

Creating a Tag

After you create a tag category, you can create tags within that category

A tag has the following properties:

- name
- description
- category ID

Python Example of Creating a Tag

This example is based on code in the `tagging_workflow.py` sample file.

This example creates a tag specification and then uses it to create the tag. The tag specification references the category identifier that was returned from the category create operation. Use the returned tag identifier for subsequent operations on the tag.

Note For a complete and up-to-date version of the sample code, see the vSphere Automation SDK Python samples at GitHub.

```
...

# Set up a tag create spec.
tag_create_spec = tag_stub.CreateSpec(name='red',
                                      description='My favorite color',
                                      category_id=fav_category_id)

# Create the tag.
tag_stub = tagging_client.Tag(my_stub_config)
tag_id = tag_stub.create(create_spec)
```

Creating Tag Associations

After you create a tag category and create a tag within the category, you can associate the tag with a vSphere managed object or a vSphere Automation resource. An association is a simple link that contains no data of its own. You can enumerate objects that are attached to a tag or tags that are attached to an object.

Tag associations are local to a vCenter Server instance. When you request a list of tag associations from a vCenter Server system, it enumerates only the associations that it has stored.

When you associate a tag with an object, the object's type must match one of the associable types specified for the category to which the tag belongs.

- Assign the Tag to a Content Library

  After you create a tag, you can assign the tag to a vSphere Automation resource.

- Assign a Tag to a Cluster

  After you create a tag, you can assign the tag to a vSphere managed object. Tags make the inventory objects in your virtual environment more sortable and searchable.

Assign the Tag to a Content Library

After you create a tag, you can assign the tag to a vSphere Automation resource.

Procedure

1. Construct a dynamic object identifier for the library.

   The dynamic identifier includes the type and ID of the object.

2. Attach the tag to the content library.

Python Example of Assigning a Tag to a Content Library

This example is based on code in the `tagging_workflow.py` sample file.

Note For a complete and up-to-date version of the sample code, see the vSphere Automation SDK Python samples at GitHub.

```
...

# 1 – Create a dynamic type ID for the content library.
library_dynamic_id = DynamicID(type=Library.RESOURCE_TYPE,
                               id=my_library.id)

# 2– Attach the tag to the ClusterComputeResource managed object.
tag_association_stub = tagging_client.TagAssociationStub(my_stub_config)
tag_association_stub.attach(tag_id,
                            library_dynamic_id)
```

Assign a Tag to a Cluster

After you create a tag, you can assign the tag to a vSphere managed object. Tags make the inventory objects in your virtual environment more sortable and searchable.

This procedure describes the steps for applying tag a to a cluster object in your inventory.

Prerequisites

Obtain the managed object identifier for the specified cluster.

To get the managed object identifier of the `ClusterComputeResource`, you must access vCenter Server by using the vSphere Web Services API. For more information about how to access Web Services, see Create a Web Services Session.

Procedure

1 Construct a dynamic object identifier for the cluster.

   The dynamic identifier includes the type and ID of the managed object reference.

2 Attach the tag to the cluster.

Python Example of Assigning a Tag to a Cluster

This example is based on code in the `tagging_workflow.py` sample file.

This example is based on the information that is provided in .

Note For a complete and up-to-date version of the sample code, see the vSphere Automation SDK Python samples at GitHub.

```
...

# 1 – Determine the MOID of the ClusterComputeResource from its name.
cluster_object = get_obj(service_content,
                         [vim.ClusterComputeResource],
                         my_cluster_name)
cluster_moid = cluster_obj._GetMoId()

# 2 – Create a dynamic type ID for the cluster.
dynamic_id = DynamicID(type='ClusterComputeResource', id=cluster_moid)

# 3 – Attach the tag to the ClusterComputeResource managed object.
tag_association_stub = tagging_client.TagAssociation(my_stub_config)
tag_association_stub.attach(tag_id=tag_id,
                            object_id=dynamic_id)
```

Updating a Tag

To update a tag, you must create an update spec for the tag. In the update spec, you set values for the fields to be changed, and omit values for the other fields. When you do an update operation using the update spec, only the fields that contain values are changed.

For example, you might use a timestamp in a tag description to identify a resource's last reconfiguration. After reconfiguring the resource, you update the tag description to contain the current time.

Python Example of Updating a Tag Description

This example is based on code in the `tagging_workflow.py` sample file.

This example adds timestamp in a tag description to identify when a resource was last reconfigured. The tag description is updated with the timestamp after the resources is reconfigured.

Note For a complete and up-to-date version of the sample code, see the vSphere Automation SDK Python samples at GitHub.

```
...

tag_stub = tagging_client.Tag(my_stub_config

# 1 – Format the current time.
date_time = time.strftime('%d/%m/%Y %H:%M:%S')
description = 'Server tag updated at ' + date_time

# 2 – Set up a tag update spec.
tag_update_spec = tag_stub.UpdateSpec()
tag_update_spec.description = description

# 3 – Apply the update spec to change the tag description.
tag_stub.update(tag_id, tag_update_spec)
```

Virtual Machine Configuration and Operations

9

A virtual machine is a software computer that, like a physical computer, runs an operating system and applications. The virtual machine consists of a set of specification and configuration files and is backed by the physical resources of a host. Each virtual machine encapsulates a complete computing environment and runs independently of the underlying hardware.

Starting with vSphere 6.5, you can configure virtual machine settings and perform power operations through the vSphere Automation SDK for Python.

This chapter includes the following topics:

Filtering Virtual Machines

You can retrieve a list of virtual machines that match a specific filter or a group of up to one thousand virtual machines available in a specific vCenter Server instance.

You can retrieve a list of up to one thousand virtual machine IDs for a single vCenter Server instance by filtering them based on a specific requirement, such as a host, cluster, datacenter, or resource pool on which the VMs are running.

Call the `list` methods of the `VM` service to retrieve only a list of the virtual machines that match your specific criteria. The method takes as parameter the `VMTypes.FilterSpec` instance that you can use to describe the virtual machine filter.

Python Example of Filtering Virtual Machines

This example is based on the code in the `vm_helper.py` sample file.

Note For a complete and up-to-date version of the sample code, see the vSphere Automation SDK Python samples at GitHub.

```
...

def get_vms(stub_config, vm_names):
    """Return identifiers of a list of vms"""
    vm_svc = VM(stub_config)
    vms = vm_svc.list(VM.FilterSpec(names=vm_names))

    if len(vms) == 0:
        print('No vm found')
        return None

    print("Found VMs '{}' ({})".format(vm_names, vms))
    return vms
```

Create a Virtual Machine

You can create a virtual machine by using the `VM.create` method. The method takes as parameter a `CreateSpec` instance that allows you to specify the attributes of the virtual machine.

To create a virtual machine you must specify the virtual machine attributes by using the `CreateSpec` class. For example, you can specify a name, boot options, networking, and memory for the virtual machine. See Configuring a Virtual Machine.

All attributes are optional except the virtual machine placement information that you must provide by using the `PlacementSpec` class. Use the virtual machine placement specification to set the datastore, cluster, folder, host, or resource pool of the created virtual machine and make sure that all these vSphere objects are located in the same data center in a vCenter Server instance.

For more information refer to the *API Reference* documentation inside the SDK.

Python Example of Creating a Basic Virtual Machine

This example is based on the code in the `create_basic_vm.py` sample file.

Note For a complete and up-to-date version of the sample code, see the vSphere Automation SDK Python samples at GitHub.

```
...

def create_basic_vm(stub_config, placement_spec, standard_network):
    """
    Create a basic VM.

    Using the provided PlacementSpec, create a VM with a selected Guest OS
```

```
    and provided name.

    Create a VM with the following configuration:
    * Create 2 disks and specify one of them on scsi0:0 since it's the boot disk
    * Specify 1 ethernet adapter using a Standard Portgroup backing
    * Setup for PXE install by selecting network as first boot device

    Use guest and system provided defaults for most configuration settings.
    """
    guest_os = testbed.config['VM_GUESTOS']

    boot_disk = Disk.CreateSpec(type=Disk.HostBusAdapterType.SCSI,
                                scsi=ScsiAddressSpec(bus=0, unit=0),
                                new_vmdk=Disk.VmdkCreateSpec())
    data_disk = Disk.CreateSpec(new_vmdk=Disk.VmdkCreateSpec())

    nic = Ethernet.CreateSpec(
        start_connected=True,
        backing=Ethernet.BackingSpec(
            type=Ethernet.BackingType.STANDARD_PORTGROUP,
            network=standard_network))

    boot_device_order = [BootDevice.EntryCreateSpec(BootDevice.Type.ETHERNET),
                         BootDevice.EntryCreateSpec(BootDevice.Type.DISK)]

    vm_create_spec = VM.CreateSpec(name=vm_name,
                                   guest_os=guest_os,
                                   placement=placement_spec,
                                   disks=[boot_disk, data_disk],
                                   nics=[nic],
                                   boot_devices=boot_device_order)
    print('\n# Example: create_basic_vm: Creating a VM using spec\n-----')
    print(pp(vm_create_spec))
    print('-----')

    vm_svc = VM(stub_config)
    vm = vm_svc.create(vm_create_spec)

    print("create_basic_vm: Created VM '{}' ({})".format(vm_name, vm))

    vm_info = vm_svc.get(vm)
    print('vm.get({}) -> {}'.format(vm, pp(vm_info)))

    return vm

...
```

Configuring a Virtual Machine

You can configure a virtual machine during creation. You can also reconfigure an existing virtual machine by adding or changing the type of the storage controllers, configure the virtual disks, boot options, CPU and memory information, and networks.

Name and Placement

You specify the display name and the location of the virtual machine by using the `CreateSpec` and `PlacementSpec` classes.

When you create your virtual machine, use the `setName` method of the `CreateSpec` class to pass as argument the display name of the virtual machine.

You must create also a `PlacementSpec` instance that describes the location of the virtual machine in regards to the resources of a given vCenter Server instance. Use the `setPlacement(PlacementSpec placement)` method of the `CreateSpec` class to set the placement information for the virtual machine. You can set one or all of the following vSphere resources: datastore, cluster, folder, host, and resource pool.

Python Example of Configuring the Placement of a Virtual Machine

This example is based on the code in the `vm_placement_helper.py` sample file.

Note For a complete and up-to-date version of the sample code, see the vSphere Automation SDK Python samples at GitHub.

```
...

def get_placement_spec_for_resource_pool(stub_config,
                                         datacenter_name,
                                         vm_folder_name,
                                         datastore_name):
    """
    Returns a VM placement spec for a resourcepool. Ensures that the
    vm folder and datastore are all in the same datacenter which is specified.
    """
    resource_pool = resource_pool_helper.get_resource_pool(stub_config,
                                                           datacenter_name)

    folder = folder_helper.get_folder(stub_config,
                                      datacenter_name,
                                      vm_folder_name)

    datastore = datastore_helper.get_datastore(stub_config,
                                               datacenter_name,
                                               datastore_name)

    # Create the vm placement spec with the datastore, resource pool and vm
    # folder
    placement_spec = VM.PlacementSpec(folder=folder,
                                      resource_pool=resource_pool,
                                      datastore=datastore)

    print("get_placement_spec_for_resource_pool: Result is '{}'".
          format(placement_spec))
    return placement_spec
```

Boot Options

You can configure the boot options of a virtual machine by using the `setBoot(CreateSpec boot)` method of the `CreateSpec` class.

The method takes as argument the `BootTypes.CreateSpec` class. You can select one of the following settings when booting the virtual machine:

- Delay - Indicates a delay in milliseconds before starting the firmware boot process when the virtual machine is powered on.
- Retry - Indicates whether the virtual machine automatically retries to boot after a failure.
- Retry delay - Indicates a delay in milliseconds before retrying the boot process after a failure.
- Enter setup mode - If set to `true`, indicates that the firmware boot process automatically enters BIOS setup mode the next time the virtual machine boots. The virtual machine resets this flag to `false` once it enters setup mode.
- EFI legacy boot - If set to `true`, indicates that the EFI legacy boot mode is used.

Python Example of Configuring the Boot Options

The following example is based on the code of the `boot.py` sample file.

Note For a complete and up-to-date version of the sample code, see the vSphere Automation SDK Python samples at GitHub.

```
...

"""
Demonstrates how to configure the settings used when booting a virtual machine.

Sample Prerequisites:
The sample needs an existing VM.
"""

vm = None
vm_name = None
stub_config = None
boot_svc = None
cleardata = False
orig_boot_info = None

...

def run():
    global vm
    vm = get_vm(stub_config, vm_name)
    if not vm:
        exit('Sample requires an existing vm with name ({}). '
             'Please create the vm first.'.format(vm_name))
    print("Using VM '{}' ({}) for Boot Sample".format(vm_name, vm))
```

```
    # Create Boot stub used for making requests
    global boot_svc
    boot_svc = Boot(stub_config)

    print('\n# Example: Get current Boot configuration')
    boot_info = boot_svc.get(vm)
    print('vm.hardware.Boot.get({}) -> {}'.format(vm, pp(boot_info)))

    # Save current Boot info to verify that we have cleaned up properly
    global orig_boot_info
    orig_boot_info = boot_info

    print('\n# Example: Update firmware to EFI for Boot configuration')
    update_spec = Boot.UpdateSpec(type=Boot.Type.EFI)
    print('vm.hardware.Boot.update({}, {})'.format(vm, update_spec))
    boot_svc.update(vm, update_spec)
    boot_info = boot_svc.get(vm)
    print('vm.hardware.Boot.get({}) -> {}'.format(vm, pp(boot_info)))

    print('\n# Example: Update boot firmware to tell it to enter setup mode on '
          'next boot')
    update_spec = Boot.UpdateSpec(enter_setup_mode=True)
    print('vm.hardware.Boot.update({}, {})'.format(vm, update_spec))
    boot_svc.update(vm, update_spec)
    boot_info = boot_svc.get(vm)
    print('vm.hardware.Boot.get({}) -> {}'.format(vm, pp(boot_info)))

    print('\n# Example: Update boot firmware to introduce a delay in boot'
          ' process and to reboot')
    print('# automatically after a failure to boot. '
          '(delay=10000 ms, retry=True,')
    print('# retry_delay=30000 ms')
    update_spec = Boot.UpdateSpec(delay=10000,
                                  retry=True,
                                  retry_delay=30000)
    print('vm.hardware.Boot.update({}, {})'.format(vm, update_spec))
    boot_svc.update(vm, update_spec)
    boot_info = boot_svc.get(vm)
    print('vm.hardware.Boot.get({}) -> {}'.format(vm, pp(boot_info)))

...
```

Operating System

The guest operating system that you specify affects the supported devices and available number of virtual CPUs.

You specify the guest operating system by using the `setGuestOS(GuestOS guestOS)` method of the `VMTypes.CreateSpec` class. The `GuestOS` class defines the valid guest OS types that you can use to configure a virtual machine.

CPU and Memory

The `CreateSpec` class allows you to specify the CPU and memory configuration of a virtual machine.

To change the CPU and memory configuration settings, use the `CpuTypes.UpdateSpec` and `MemoryTypes.UpdateSpec` classes.

CPU Configuration

You can set the number of CPU cores in the virtual machine by using the `setCount` method of the `CpuTypes.UpdateSpec` class. The supported range of CPU cores depends on the guest operating system and virtual hardware version of the virtual machine. If you set `CpuTypes.Info.getHotAddEnabled()` and `CpuTypes.Info.getHotRemoveEnabled()` to `true`, you allow virtual processors to be added or removed from the virtual machine at runtime.

Memory Configuration

You can set the memory size of a virtual machine by using the `setSizeMiB` method of the `MemoryTypes.UpdateSpec` class. The supported range of memory sizes depends on the configured guest operating system and virtual hardware version of the virtual machine. If you set `MemoryTypes.UpdateSpec.setHotAddEnabled()` to `true` while the virtual machine is not powered on, you enable adding memory while the virtual machine is running.

Python Example of Configuring the CPU and Memory of a Virtual Machine

These examples are based on the code in the `cpu.py` and `memory.py` sample files.

Note For a complete and up-to-date version of the sample code, see the vSphere Automation SDK Python samples at GitHub.

The following example shows how you can update the CPU configuration of a virtual machine.

```
...

vm = None
vm_name = None
stub_config = None
cpu_svc = None
cleardata = False
orig_cpu_info = None

...
        server, username, password, cleardata, skip_verification, vm_name = \
            parse_cli_args_vm(testbed.config['VM_NAME_DEFAULT'])
        stub_config = vapiconnect.connect(server,
                                          username,
                                          password,
                                          skip_verification)
...
def run():
    global vm
```

```
    vm = get_vm(stub_config, vm_name)
    if not vm:
        exit('Sample requires an existing vm with name ({}). '
             'Please create the vm first.'.format(vm_name))
    print("Using VM '{}' ({}) for Cpu Sample".format(vm_name, vm))

    # Create CPU stub used for making requests
    global cpu_svc
    cpu_svc = Cpu(stub_config)

    # Get the current CPU configuration
    cpu_info = cpu_svc.get(vm)
    print('vm.hardware.Cpu.get({}) -> {}'.format(vm, pp(cpu_info)))

    # Save current CPU info to verify that we have cleaned up properly
    global orig_cpu_info
    orig_cpu_info = cpu_info

    # Update the number of CPU cores of the virtual machine
    update_spec = Cpu.UpdateSpec(count=2)
    print('vm.hardware.Cpu.update({}, {})'.format(vm, update_spec))
    cpu_svc.update(vm, update_spec)

    # Get the new CPU configuration
    cpu_info = cpu_svc.get(vm)
    print('vm.hardware.Cpu.get({}) -> {}'.format(vm, pp(cpu_info)))

    # Update the number of cores per socket and
    # enable adding CPUs while the virtual machine is running
    update_spec = Cpu.UpdateSpec(cores_per_socket=2, hot_add_enabled=True)
    print('vm.hardware.Cpu.update({}, {})'.format(vm, update_spec))
    cpu_svc.update(vm, update_spec)
...
```

The following example demonstrates how you can add memory to a running virtual machine.

```
...

vm = None
vm_name = None
stub_config = None
memory_svc = None
cleardata = False
orig_memory_info = None

...
    server, username, password, cleardata, skip_verification, vm_name = \
            parse_cli_args_vm(testbed.config['VM_NAME_DEFAULT'])
    stub_config = vapiconnect.connect(server,
                                      username,
                                      password,
                                      skip_verification)

...
```

```
    global vm
    vm = get_vm(stub_config, vm_name)
    if not vm:
        exit('Sample requires an existing vm with name ({}). '
             'Please create the vm first.'.format(vm_name))
    print("Using VM '{}' ({}) for Memory Sample".format(vm_name, vm))

    # Create Memory stub used for making requests
    global memory_svc
    memory_svc = Memory(stub_config)

    # Get the current Memory configuration
    memory_info = memory_svc.get(vm)
    print('vm.hardware.Memory.get({}) -> {}'.format(vm, pp(memory_info)))

    # Update the memory size of the virtual machine
    update_spec = Memory.UpdateSpec(size_mib=8 * 1024)
    print('vm.hardware.Memory.update({}, {})'.format(vm, update_spec))
    memory_svc.update(vm, update_spec)

    # Get the new Memory configuration
    memory_info = memory_svc.get(vm)
    print('vm.hardware.Memory.get({}) -> {}'.format(vm, pp(memory_info)))

    # Enable adding memory while the virtual machine is running
    update_spec = Memory.UpdateSpec(hot_add_enabled=True)
    print('vm.hardware.Memory.update({}, {})'.format(vm, update_spec))
    memory_svc.update(vm, update_spec)

...
```

Networks

You configure network settings so that a virtual machine can communicate with the host and with other virtual machines. When you configure a virtual machine, you can add network adapters (NICs) and specify the adapter type.

You can add virtual Ethernet adapters to a virtual machine by using the `VMTypes.CreateSpec.setNics` method. Pass as argument a `List` of `EthernetTypes.CreateSpec` objects that provide the configuration information of the created virtual Ethernet adapters. You can set the MAC address type to `EthernetTypes.MacAddressType.MANUAL`, `EthernetTypes.MacAddressType.GENERATED`, or `EthernetTypes.MacAddressType.ASSIGNED`. Select `MANUAL` to specify the MAC address explicitly.

You can specify also the physical resources that back a virtual Ethernet adapter by using the `EthernetTypes.BackingSpec.setType` method. The method takes as argument one of the following types: `EthernetTypes.BackingType.STANDARD_PORTGROUP`, `HOST_DEVICE`, `DISTRIBUTED_PORTGROUP`, or `OPAQUE_NETWORK`.

Python Example of Configuring the Virtual Machine Network

This example is based on the code in the `ethernet.py` sample file.

Note For a complete and up-to-date version of the sample code, see the vSphere Automation SDK Python samples at GitHub.

```
...

vm = None
vm_name = None
stub_config = None
ethernet_svc = None
cleardata = False
nics_to_delete = []
orig_nic_summaries = None
...

        server, username, password, cleardata, skip_verification, vm_name = \
            parse_cli_args_vm(testbed.config['VM_NAME_DEFAULT'])
        stub_config = vapiconnect.connect(server,
                                          username,
                                          password,
                                          skip_verification)

    global vm
    vm = get_vm(stub_config, vm_name)
    if not vm:
        exit('Sample requires an existing vm with name ({}). '
             'Please create the vm first.'.format(vm_name))
    print("Using VM '{}' ({}) for Disk Sample".format(vm_name, vm))

    # Get standard portgroup to use as backing for sample
    standard_network = network_helper.get_standard_network_backing(
        stub_config,
        testbed.config['STDPORTGROUP_NAME'],
        testbed.config['VM_DATACENTER_NAME'])

    # Create Ethernet stub used for making requests
    global ethernet_svc
    ethernet_svc = Ethernet(stub_config)
    vm_power_svc = Power(stub_config)
    nic_summaries = ethernet_svc.list(vm=vm)

    # Save current list of Ethernet adapters to verify that we have cleaned
    # up properly
    global orig_nic_summaries
    orig_nic_summaries = nic_summaries

    global nics_to_delete

    # Create Ethernet Nic using STANDARD_PORTGROUP with the default settings
    nic_create_spec = Ethernet.CreateSpec(
        backing=Ethernet.BackingSpec(
```

```
            type=Ethernet.BackingType.STANDARD_PORTGROUP,
            network=standard_network))
    nic = ethernet_svc.create(vm, nic_create_spec)
    nics_to_delete.append(nic)
    nic_info = ethernet_svc.get(vm, nic)

    # Create Ethernet Nic by using STANDARD_PORTGROUP
    nic_create_spec = Ethernet.CreateSpec(
        start_connected=True,
        allow_guest_control=True,
        mac_type=Ethernet.MacAddressType.MANUAL,
        mac_address='01:23:45:67:89:10',
        wake_on_lan_enabled=True,
        backing=Ethernet.BackingSpec(
            type=Ethernet.BackingType.STANDARD_PORTGROUP,
            network=standard_network))
    nic = ethernet_svc.create(vm, nic_create_spec)
    nics_to_delete.append(nic)
    nic_info = ethernet_svc.get(vm, nic)

    # Update the Ethernet NIC with a different backing
    nic_update_spec = Ethernet.UpdateSpec(
        backing=Ethernet.BackingSpec(
            type=Ethernet.BackingType.STANDARD_PORTGROUP,
            network=standard_network))
    ethernet_svc.update(vm, nic, nic_update_spec)
    nic_info = ethernet_svc.get(vm, nic)

    # Update the Ethernet NIC configuration
    nic_update_spec = Ethernet.UpdateSpec(
        wake_on_lan_enabled=False,
        mac_type=Ethernet.MacAddressType.GENERATED,
        start_connected=False,
        allow_guest_control=False)
    ethernet_svc.update(vm, nic, nic_update_spec)
    nic_info = ethernet_svc.get(vm, nic)

    # Powering on the VM to connect the virtual Ethernet adapter to its backing
    vm_power_svc.start(vm)
    nic_info = ethernet_svc.get(vm, nic)

    # Connect the Ethernet NIC after powering on the VM
    ethernet_svc.connect(vm, nic)

    # Disconnect the Ethernet NIC while the VM is powered on
    ethernet_svc.disconnect(vm, nic)
...
```

Performing Virtual Machine Power Operations

You can start, stop, reboot, and suspend virtual machines by using the methods of the `Power` class.

A virtual machine can have one of the following power states:

- `PowerTypes.State.POWERED_ON` - Indicates that the virtual machine is running. If a guest operating system is not currently installed, you can perform the guest OS installation in the same way as for a physical machine.
- `PowerTypes.State.POWERED_OFF` - Indicates that the virtual machine is not running. You can still update the software on the physical disk of the virtual machine, which is impossible for physical machines.
- `PowerTypes.State.SUSPENDED` - Indicates that the virtual machine is paused and can be resumed. This state is the same as when a physical machine is in standby or hibernate state.

To perform a power operation on a virtual machine, you can use one of the methods of the `Power` class. Before you call one of the methods to change the power state of a virtual machine, you must first check the current state of the virtual machine by using the `Power.get` method. Pass as argument the virtual machine identifier.

Following is a list if the power operations:

- `Power.start` - Powers on a powered off or suspended virtual machine. The method takes as argument the virtual machine identifier.
- `Power.stop` - Powers off a powered on or suspended virtual machine. The method takes as argument the virtual machine identifier.
- `Power.suspend` - Pauses all virtual machine activity for a powered on virtual machine. The method takes as argument the virtual machine identifier.
- `Power.reset` - Shuts down and restarts the guest operating system without powering off the virtual machine. Although this method functions as a `stop` method that is followed by a `start` method, the two operations are atomic with respect to other clients, meaning that other power operations cannot be performed until the `reset` method completes.

Python Example of Powering On a Virtual Machine

This example is based on the code in the `ethernet.py` sample file.

Note For a complete and up-to-date version of the sample code, see the vSphere Automation SDK Python samples at GitHub.

```
...

    vm = None
    vm_name = None
    stub_config = None

    server, username, password, cleardata, skip_verification, vm_name = \
             parse_cli_args_vm(testbed.config['VM_NAME_DEFAULT'])
     stub_config = vapiconnect.connect(server, username, password, skip_verification)

     # Get the virtual machine ID
```

```
    vm = get_vm(stub_config, vm_name)

    # Create the Power stub  for running power operations on virtual machines
    vm_power_svc = Power(stub_config)
    vm_power_svc.start(vm)
```