VMware Telco Cloud Automation SDK Programming Guide

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

https://docs.vmware.com/

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

Copyright © 2022 VMware, Inc. All rights reserved. Copyright and trademark information.

Contents

Introduction

1

What this programming guide is all about, and the TCA APIs the programmers can use the SDK for.

The TCA SDKs provide language bindings for accessing NFV Orchestration and Management APIs provided by the VMware Telco Cloud Automation™, also known as TCA™. This SDK user-guide discusses SDK-Python which may be used for automating TCA operations using *Python* programming. This programming guide also presents examples of using the SDK in Python.

The VMware TCA™ APIs may be broadly classified as System-Management and NFV-Orchestration APIs. The TCA™ includes a TCA-Manager (also referred to as “TCA”) and a TCA-Control Plane (also referred to as “CP”). In the next section we summarize the APIs offered by the TCA, which a programmer can use the SDK for.

This chapter includes the following topics:

The NFV Orchestration and Management APIs

The APIs offered by the TCA™.

System Management APIs

The System Management API may be used to configure TCA as well as CP. These are typically used to configure the TCA/CP appliances, and troubleshooting. The API include:

- Configure TCA and CP appliances or Cloud-Native deployment services
- Appliance Backup and Restore
- Managing Troubleshooting and Syslog configurations
- Managing TCA and CP System Upgrades

NFV Orchestration APIs

- NFV SOL (2.7.1) APIs - SOL005, SOL003
- CaaS and Virtual-Infrastructure management

- Management of Partner-Systems and Extensions

Glossary

Glossary of Terms used in this guide

NFV - Network Function Virtualization.

TCA - The product VMware Telco Cloud Automation™.

SOL - The ETSI standards specification documents specifying the REST APIs for NFV. The TCA™ supports SOL003 and SOL005.

VIM - Virtual Infrastructure Manager.

CP - Control Plane; precisely, the TCA-Control-Plane.

SDK - Software Development Kit.

API - Application Programming Interface.

CaaS - Containerization As a Service

VNF - Virtualized Network Function

NS - Network Service, in the NFV parlance.

VM - Virtual Machine.

CNF - Cloud-Native Network Function. Often referred to as 'Containerized Network Function' to differentiate it from the 'VM-based Network Function'.

NF - Network Function, in the NFV parlance. The NF represents a generic network function, inclusive of VNF, PNF, CNF.

K8S - Kubernetes

SSL - Secure Socket Layer.

LCM - Lifecycle Management of Network Functions (in the parlance of SOL APIs).

MD5 - "Message Digest" (5) checksum of a binary or an archive (usually a deliverable)

SHA1, SHA256 - "Secure Hash Algorithm" checksum of a binary or an archive (usually a deliverable).

Using the TCA SDK-Python

2

How to consume the SDK to invoke TCA APIs, in Python

The TCA SDK is primarily available for Python programming language. You can download the SDK-Python from https://code.vmware.com/home. This chapter includes the following topics:

- Installing SDK-Python and Managing the Dependencies
- Programming with the SDK

This chapter includes the following topics:

- Installing SDK-Python and Managing the Dependencies
- Programming with the SDK

Installing SDK-Python and Managing the Dependencies

The SDK Intallation guidelines

The TCA SDK-Python is available for download in the form of a tar ball: VMware-Telco-Cloud-Automation-SDK-2.1.0-<buildnumber>.tar, where the buildnumber corresponds to the Released Build number of TCA™. The SDK tar is accompanied by files containing checksums and MD5 hashes of the same. The checksum files must as well be downloaded for the purpose of verifying the integrity of the SDK tar.

The downloaded tar must be verified by a user (or programmer) to validate its integrity using checksum, follow one (or more) of the steps below.

Using SHA1 and SHA256 (on a Unix, UNIX-like, or OSX operating system):

```
# Set ALGO=1; and rerun for validating sha1 hash:
ALGO=256;diff -is <(shasum -a $ALGO VMware-Telco-Cloud-Automation-SDK-2.1.0-buildnumber.tar |
cut -d\  -f1) <(echo `cat VMware-Telco-Cloud-Automation-SDK-2.1.0-buildnumber.tar.sha${ALGO}`)

# The output of the above command should indicate the Hashes are idendical, something like,
>> Files /dev/fd/11 and /dev/fd/12 are identical
```

Using MD5 (on a Unix, UNIX-like):

```
diff -is <(md5sum VMware-Telco-Cloud-Automation-SDK-2.1.0-buildnumber.tar | cut -d\  -f1)
<(echo `cat VMware-Telco-Cloud-Automation-SDK-2.1.0-buildnumber.tar.md5`)
```

Using MD5 (on OSX):

```
diff -is <(md5 -q VMware-Telco-Cloud-Automation-SDK-2.1.0-buildnumber.tar ) <(echo `cat
VMware-Telco-Cloud-Automation-SDK-2.1.0-buildnumber.tar.md5`)
```

Once the tar ball integrity is verified, and is expanded as shown below, the user may use the installer wheel with pip3 to install the SDK.

```
# Expand the tarball (expands to the current working directory)
tar -xvf VMware-Telco-Cloud-Automation-SDK-2.1.0-19575732.tar

# List installed TCA SDK-Python packages
python3.8 -m pip list -l -v -v -v | grep -E "tca|Package|\-\-\-"

# Uninstall (a previous TCA SDK-Python installation)
python3.8 -m pip uninstall vmware-tca-sdk-python

# Install
## From the source:
python3.8 setup.py install

## Using the Wheel installers (uninstalls a previour version)
python3.8 -m pip install wheels/vmware_tca_sdk_python-2.1.0-py3-none-any.whl  --force-
reinstall
```

`Note: The CLI option--`*`force-reinstall`*`uninstalls a previous version (if any) first.`

On untarring the SDK tar, you should be able to see the folder structure similar to as shown below.

```
-rw-r--r--    1 banerjees  staff   3508 Mar 23 14:39 README.md
-rw-r--r--    1 banerjees  staff    406 Mar 23 14:39 README.txt
drwxr-xr-x  719 banerjees  staff  23008 Mar 23 15:13 docs
-rw-r--r--    1 banerjees  staff   1663 Mar 23 15:10 git_push.sh
-rw-r--r--    1 banerjees  staff     95 Mar 23 14:39 requirements.txt
-rw-r--r--    1 banerjees  staff   1620 Mar 23 14:39 setup.pyd
rwxr-xr-x   27 banerjees  staff    864 Mar 23 15:13 tca
drwxr-xr-x   42 banerjees  staff   1344 Mar 23 15:13 test
-rw-r--r--    1 banerjees  staff     80 Mar 23 15:10 test-requirements.txt
-rw-r--r--    1 banerjees  staff    189 Mar 23 14:39 tox.ini
drwxr-xr-x    7 banerjees  staff    224 Mar 23 16:03 wheels
```

The dependency libraries may be found in the file `requirements.txt`. A programmer may refer to the SDK Models in `docs`, which contains the Markdown references to the same. Mode details may be found in the forthcoming sections.

API Reference

Developer references to the SDK models and methods.

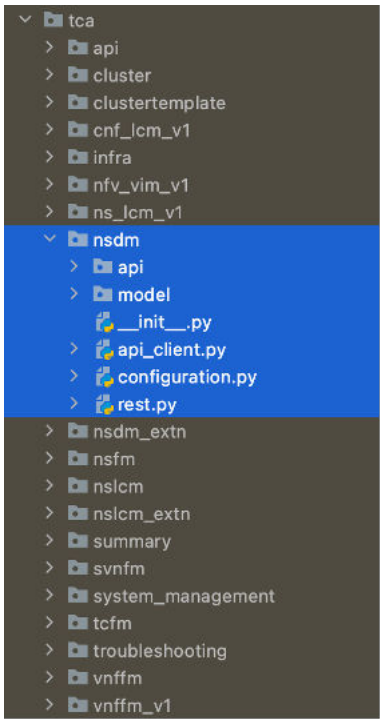
The SDK APIs and Model reference documentation is included in the `/docs` directory, in the Markdown format. Sample codes for reference are included in the `/test` directory, while the Python SDK source files are in the `/tca` directory.

SDK-Python Source Packages

Developers references to the SDK source packages and the naming convention.

The SDK client package names are in `tca.xyz` format, where `xyz` is the name of an API segment. An API segment is a set of APIs that have a specific purpose identified by the TCA. For instance, the segment `nsdm` refers to the set of APIs that manage the NFV Network-Service Descriptors. The package `tca.nsdm`, like the other segment packages, contains the API-Client and the Python object models required by the programmer to invoke the corresponding APIs. This is highlighted in the picture below.

Figure 2-1. SDK-Python source packages



The SDK also ships an inbuilt Session-Management System to heavy-lift the API-Token management, Token expiry and renewal, Configuration-Management, and modules to manage other cross-cutting concerns, so the programmer using the SDK can focus solely on the business logic.

Programming with the SDK

Components and Configurations needed to use the SDK

The TCA APIs have been segregated into broad-level API-segments; each segment having a specific purpose as designated by the TCA. This has been done primarily for the ease of programming and managing the client-side code.

As discussed in SDK-Python Source Packages each segment is represented by a Python package, that contains one or more API-Clients, containing the boilerplate code to abstract various low-level handling of data, serialization, de-serialization, SSL-handshakes etc. An API-Clients offers a façade to the programmer containing Operations; each operation corresponding to an API method that gets invoked on the TCA server. The next sections describe how to list the API-clients and the operations they can invoke.

Listing the API Clients and Operations

Develpopers reference to list the available API Segments.

The SDK ships with an SDKManager to help manage the API-Clients and the operations. A reference code to list the API-Clients, and the operations on a specific API-Client `nsdm`.

```
from tca.api.client.sdk_manager import SDKManager as SM
sdk_mgr = SM()

#LISTS THE AVAILABLE API-CLIENT KEYS, AND THE CORRESPONDING API SEGMENT
sdk_mgr.list()

#LISTS THE AVAILABLE OPERATIONS FOR A GIVEN API CLIENT
## The method list() outputs to the System outstream,
## while get() returns a List or a Dict.
sdk_mgr.get('tca.nsdm:DefaultApi').list()
```

Invoking the operations

Developer reference to invoking a method on a given API Segment

The SDK ships with a set of management helpers to make the programmers' job easier invoking the operations on the API-Clients – Session Manager and Configuration Manager. The Session Manager abstracts the knowledge of managing Authentication tokens, while the Configuration Manager requires a TCAConfiguration object to be configured by the programmer, and implicitly manages it across the API-Client, so the programmer gets a seamless experience. The reference code below may be used from within a programmer's `__init__.py` file

```
from tca.api.client.tca_configuration import TCAConfiguration
from tca.api.client.session_manager import HybridditySessionManager
from tca.api.client.sdk_manager import SDKManager as SM

# Setting up TCAConfigurationglobal
tca_configtca_config = TCAConfiguration() tca_config.set_server('https://
your_tca_server:443') tca_config.set_username('tenant_name@tenant_org.com')
tca_config.set_password('**************************')

# Setting up Session Managerglobal session_manager session_manager = HybridditySessionManager()
```

```
# Setting up the SDKManager
global sdk_manager sdk_manager = SM(). \              with_configuration(tca_config). \
with_session_manager(session_manager)

# Fetch the API-Client, using the corresponding Key name
## To Be discussed in details in forthcoming sections
vnfpkgm_client = sdk_manager.get_api('tca.vnfpkgm:VnfCatalogApi')
```

Please note, there should be no trailing '/' in the input to tca_confg.set_server(…). An advanced programmer at any point of time may choose to write or extend the SDK with his/her own helper classes or can invoke the APIs without the management helpers. The details about how to extend the SDK helpers for the advanced programmers is outside the scope of this document.

Auth and Session Management

The HybridditySessionManager implicitly manages the expiry and renewal of a proprietary API-Token (x-hm-authorization) required by the TCA API-server. The SDK also ships a default SessionManager – NoOpSessionManager; the use of which will be described in the section "Invoking System Management APIs". An advanced programmer may want to write a custom SessionManager, the details about which is outside the scope of this document.

Sync and Async Invocation

An API-client in the SDK, offers operations to be invoked on the TCA-server. Each operation has an implicit Boolean argument 'async_req', which is defaults to False. If set to True, the invocation is done asynchronously, and a provision of call-back is provided to be set by the programmer.

```
# My CallBack handlerdef callback(*args, **kwargs):        print("=======  CALLBACK
========")
#print("***** LOCALS: ", locals())
print("***** RESULTS - COUNT!!!: ", len(args[0]))
print("***** RESULTS: ", args[0])
print("***** KW-ARGS: ", kwargs)
print("======== / CALLBACK ======\n")

# Fetch VNF Catalogs ASYNC;
## vnfpkgm_client obtained as in the previous example
async_inv_vnfds = vnfpkgm_client.get_all_vnf_descriptors('2.7.1',
accept='application/json',                                                    async_req=True)
async_inv_vnfds.callback = callback
```

Note, that the method parameters given below are optional, and may be used with any operation on the API Client:

Param Name	Optional?	Default to:
version	yes	The latest version ('2.7.1', as of writing this guide)
accept	yes	'application/json'
async_req	yes	False

Invocation Using with SSL

The SDK allows the programmer to consume TCA APIs that might be exposed over HTTPS, with a self-signed certificate. The reference code below depicts how the TCAConfiguration can be setup to allow various possibilities with SSL communication between the API-Client and the TCA server.

```
# Setting up TCAConfiguration with SSL
global tca_configtca_config = TCAConfiguration()
tca_config.set_server('https://your_tca_server:443')
tca_config.set_username('tenant_name@tenant_org.com')
tca_config.set_password('**************************')
tca_config.set_verify_ssl(True)
tca_config.cert_file =
    "/com/vmware/sdk-trials/sdk-py/cert.pem"
tca_config.key_file = "/com/vmware/sdk-trials/sdk-py/key.pem"
tca_config.ssl_ca_cert =
    "/com/vmware/sdk-trials/sdk-py/ca_cert.pem"
tca_config.assert_hostname = True

# The tca_config may now be used for subsequent API Calls
## vnfpkgm_client obtained as in the previous example
vnfpkgm_client.get_all_vnf_descriptors('2.7.1',                    accept='application/
json')
```

Invoking Orchestration APIs

The Orchestration APIs are multi-tenant APIs that the TCA offers over the default 443 port. The TCA Authentication is managed using the API-Key ‘x-hm-authorization’. The SDK ships with HybridditySessionManager to manage the Authentication key. Please go through the previous sections for the details. The examples in the previous sections depict the configurations required for the invocation of Orchestration APIs.

Apart from the Orchestration APIs, the SDK also allows managing the TCA System and/or Appliance, which are a set of Administrative APIs and require super-user privileges. The next section deals with the System Management APIs.

Invoking System Management APIs

The System Management APIs are for use by the System Administrators to manage the TCA Systems or the Appliances, including upgrades, backup and restore etc… These APIs require super-user privileges. These APIs are usually exposed by the TCA server on port 9443 and require admin credentials over HTTP Basic authentication. The APIs do not need an authentication session. The code below may be used as a reference.

```
tca_config = TCAConfiguration() tca_config.set_server('https://tca_server')
tca_config.set_username('admin_login') tca_config.set_password('***********')

# Insecure: only for NFV functional validation tca_config.set_verify_ssl(False)
# No session management; it’s HTTP Basic
Authsessionmgr = NoOpSessionManager()
sdk_manager = SDKManager(). \
    with_configuration(tca_config). \
```

```
    with_session_manager(sessionmgr)

# Fetch the API-Client using the corresponding key.
system_mgmt_client = sdk_manager. \
    get_api('tca.system_management:CertificatesApi')

# Invoke a Admin Business Operation - e.g. get TCA Appliance Certificates
system_mgmt_client.api_admin_certificates_get()
```

Running the Sample Applications

Developer Reference to invoke the bundled sample code and testcases.

The TCA SDK for Python provides sample applications, which you can find under the /test directory. To execute a sample test, you need to update the TCAConfiguration to set your server address, tenant credentials, SSL certificate paths (in case the set_verify_ssl has been set to True). Having done the changes, you may run the below command line on a Unix shell to execute the testcase.

```
python3 test/test_auth_token_renew.py
```

Log Control

How to enable debug logging for general troubleshooting.

Each SDK allow setting up log levels in the Configuration Manager. The default log level stays at "INFO", which may be set to a different level by the programmer. Once the level is explicitly set, the log-level applies to all the API clients. The subsequent invocations to any/all of the API clients will reflect the log-level that has been set.

The code below depicts setting the log level to 'DEBUG' (assumes the programmer has referenced 'tca_config', as shown in the previous examples):

```
...

# tca_config creates as shown in the previous examples
tca_config.lg().setLevel('DEBUG')
...

# Create SDK manager with the above configuration
sdk_manager = SDKManager(). \
    with_configuration(tca_config). \
...
```

Often it's convenient to set DEBUG level only for a specific API (say, for troubleshooting). In such cases, the programmer may set the log-level only for a specific API-clients, as shown below (assuming the programmer has created the sdk_manager instace).

```
...
# reference the API-client from the sdk_manager instance
cluster_api_client = \
        sdk_manager.get_api('tca.clustertemplate:ClusterTemplatesApi')
```

```
# Set the log-level for the specific API-client
cluster_api_client.api_client.configuration.lg().setLevel('DEBUG')
...
```

Operations Using SOL APIs

3

How to invoke SOL APIs that are part of NFV Orchestration

The SOL APIs (ETSI SOL003 and SOL005) are part of TCA Orchestration APIs. The APIs detail primarily with Life Cycle Management of Virtualized-Network-Functions, Network-Services, and the corresponding descriptor (or catalogs).

This chapter includes the following topics:

- Onboarding and Fetching Catalog Items
- Instantiating a VNF and Running LCM operations
- Running the Sample Applications

The VNF Catalog and LCM APIs may be invoked using the API segment's `API-Client Key`, as listed in the table below. The key needs to be used with the SDKManager to obtain the corresponding API-Client instance, which in turn enables the programmer to invoke methods corresponding to the API Segment.

API-Client Key to the Segment	API Segment being referred
'tca.nsdm:DefaultApi'	Network Service Descriptor Management – V2
'tca.vnfpkgm:VnfCatalogApi'	VNF Package management - V2
'tca.vnflcm:DefaultApi'	VNF LCM – V2
'tca.nsfm:DefaultApi'	Network Service Fault Management – V2
'tca.nslcm:DefaultApi'	Network Service LCM -V2
'tca.vnffm:DefaultApi'	VNF Fault Management – V2
'tca.vnfpm:DefaultApi'	VNF Performance Management – V2

This chapter includes the following topics:

- Onboarding and Fetching Catalog Items
- Instantiating a VNF and Running LCM operations
- Registering and Listing VIMs

Onboarding and Fetching Catalog Items

Developer reference to onboard CSARs (NFV SOL APIs for Catalog, VNF-Packages)

In the following example, a new VNF CSAR is onboarded, followed by fetching all the catalog items.

```
from packaging import tagsfrom tca.api.client.session_manager
import HybriditySessionManager
from tca.api.client.sdk_manager import SDKManager
from tca.vnfpkgm import CreateVnfPkgInfoRequest, VnfPkgInfoUserDefinedData,
VnfPkgInfoUserDefinedDataTags

def main():
    hybridity_sessionmgr = HybriditySessionManager()
    sdk_manager = SDKManager(). \
            with_configuration(tca_config). \
with_session_manager(hybridity_sessionmgr)

    # Fetch the API-Client, using the corresponding key.
    vnfpkgm_client = sdk_manager.get_api('tca.vnfpkgm:VnfCatalogApi')
    print("[Test-catalog] Fetched VNFPKGM CLIENT: ", vnfpkgm_client)

    #  1.  Create a VNF PkgInfo (Reference: SOL005 2.7.1)
    body = CreateVnfPkgInfoRequest()
    ## For allowed Tags, please refer to the TCA API documentation
    tag = VnfPkgInfoUserDefinedDataTags(
        name="sdk_tag_001:sdk_tag_001", auto_created=False)
    body.user_defined_data = VnfPkgInfoUserDefinedData(
        name="my-vcpuGrantDisabled-sdk-001", tags=[tag])
    ## Create a VNF Descriptor
    ### Please refer to ETSI SOL005 APIs, and the TCA API documentation
    vnfpkg = vnfpkgm_client.create_vnf_descriptor(body,
        version="2.7.1", accept="application/json",
        content_type="application/json")
    print("NEW VNFPKG: ", vnfpkg)

    # 2. Upload the contents to the package
    vnf_pkg_id = vnfpkg.id
    vnfpkgm_client.upload_vnf_descriptor("2.7.1",
        ## Optional 'Accept' request header
        accept="application/json",
        ## The mandatory package-id
        vnf_pkg_id=vnf_pkg_id,
        ## Please refer to the TCA API documentation
        content_type="multipart/form-data",         file="/my/NFV/CSARs/catalog/csar/
MyTest.csar")

    # 3 NOW VALIDATE!
    ## Fetch all descriptors and verify “Onboarded” status of
    ## the new VNF Package
    all_vnf_desc = vnfpkgm_client.get_all_vnf_descriptors(
        '2.7.1',
        accept='application/json')
```

```
    print("ALL Desriptors: ", all_vnf_desc)

if __name__ == '__main__':
    main()
```

Each VNF package has a descriptor-ID, which is used while creating an instance of the VNF. The descriptor-ID may be fetched by listing the VNF packages (as shown in the examples above) and taking the value of the field ‘vnfd_id’ for the desired package record. For details, please refer to the ETSI NFV architecture, and the SOL001 specification.

A CSAR file, an archive containing the NF (or NS) descriptor, may be created with the help of the Designer-tool in the TCA™ UI. For details, please refer to https://docs.vmware.com.

Instantiating a VNF and Running LCM operations

Perform LCM operations on VNF/CNF.

Once a VNF Package is onboarded, you can instantiate it, and perform other Life-Cycle Management operations.

Instantiation of a VNF/CNF

Below is an example code to instantiate a VNF, from a descriptor with the given ‘vnfd_id’.

```
# Create an instance from the VNF Descriptor
def create_vnf_instance_from_descriptor(setup):
    body = CreateVnfRequest(
        vnfd_id="nfd_226693ca-96b3-4e04-8eb5-b26c2e3cdea8",
vnf_instance_name="test-sdk-python",
        vnf_instance_description="sdk test vnf instantiation",
        metadata=MetaData(tags=[], extension_id=None))
    data = client_vnflcm.vnf_instances_post(body,
        accept="application/json",
        content_type="application/json",
        version="2.7.1",
        async_req=False)

    global vnf_instance_id
    vnf_instance_id = data.id
    assert type(data) is VnfInstance

# Instantiate the VNF instance
def vnf_instance_lcm_instantiate_post(setup):
    extcps = [NsInstantiateAdditionalParamsExtCps(cpd_id="ext-vdu1",
                    cp_config=[]),
              NsInstantiateAdditionalParamsExtCps(cpd_id="ext-vdu2",
                    cp_config=[])]
    extvirtuallinks = NsInstantiateAdditionalParamsExtVirtualLinks(id=0,
resource_id="dvportgroup-8712",
             network_name="CaaS-Uplink-612",
             ext_cps=extcps)
    vduparams = [VduParam(vdu_name="vdu1",
                 deployment_profile_id= "45010e8b-1122-4a95-91d3-
```

```
c057824f1127"),
                VduParam(vdu_name="vdu2",
                deployment_profile_id="45010e8b-1122-4a95-91d3-c057824f1127")]
    # Additional Parameters - please refer to TCA API docs
    additionalparams = VnfInstantiateRequestAdditionalParams(
             vdu_params=vduparams,
             lcm_interfaces=None,
             entity_prefix="",
             use_v_app_templates=False,
             catalog_name=None,
catalog_id=None,
             skip_grant=False,
             ignore_grant_failure=None,
             disable_auto_rollback=None,
             disable_grant=None,
             skip_node_customization=None,
             is_vnf_v_app_template=False,
             v_app_template_name=None,
             vimconnectioninfo = VimConnectionInfo(
                 id="vmware_75EDF5914FBF40E2AAE92129A87D8A45",                  vim_type="",
                 extra = VimConnectionInfoExtras(
                     deployment_profile_id="45010e8b-1122-4a95-91d3-c057824f1127"))
    body = InstantiateVnfRequest(
        flavour_id="default", instantiation_level_id=None,
        ext_virtual_links=[extvirtuallinks],
        ext_managed_virtual_links=None,
        vim_connection_info=[vimconnectioninfo],
        localization_language=None,
        additional_params=additionalparams,
        extensions=None, vnf_configurable_properties=None)

    # Instantiate a VNF
    data = client_vnflcm.vnf_instances_vnf_instance_id_instantiate_post(
              body, accept="application/json",
              content_type="application/json", version="2.7.1",
              vnf_instance_id=vnf_instance_id, async_req=False)
```

The Response of the POST operation has a URI in the ‘Location’ header. The Location can be used to track the progress of the LCM operation of the newly created VNF instance. The *vnf_ instance_ id* defined as a global variable in the above sample code is being used in the following examples to do the other LCM operations.

Scaling a VNF/CNF instance

Below is an example code to Scale VNF.

```
def vnf_instance_lcm_scale_post(setup):
    additionalparams = ScaleVnfRequestAdditionalParams(
            vdu_params=[], lcm_interfaces=[])
    # Prepare the parameters
    body = ScaleVnfRequest(
            type="SCALE_OUT", aspect_id="aspect_0",
            number_of_steps=1, additional_params=additionalparams)
    # Scale Operation
```

```
    data = client_vnflcm.vnf_instances_vnf_instance_id_scale_post(body,
accept="application/json",
            content_type="application/json",
            version="2.7.1", vnf_instance_id=vnf_instance_id)
```

Terminating an Instantiated VNF/CNF

Similar to Scaling, an insance may be terminated as shown below.

```
def vnf_instance_lcm_terminate_post(setup):
    additionalparams = TerminateVnfRequestAdditionalParams(
        lcm_interfaces=[])
    body = TerminateVnfRequest(termination_type="GRACEFUL",
              graceful_termination_timeout=120,
              additional_params=additionalparams)
    data = client_vnflcm.vnf_instances_vnf_instance_id_terminate_post(
               body, accept="application/json",
               content_type="application/json",
               version="2.7.1", vnf_instance_id=vnf_instance_id)
```

Registering and Listing VIMs

VIM Management API Invocation

The VIM APIs allow you to register Virtual Infrastructure with the TCA.

```
def main:
        tca_config = TCAConfiguration()
        # Ensure no trailing slash '/' in the tca_server
        tca_config.set_server('https://tca_server')
        tca_config.set_username('tenant@vsphere.local')
        tca_config.set_password('********************')
        # Insecure: False, only for NFV functional validation
        ## False Not recommended in production
        tca_config.set_verify_ssl(True)
        tca_config.cert_file = "/com/vmware/sdk-trials/sdk-py/cert.pem"
        tca_config.key_file = "/com/vmware/sdk-trials/sdk-py/key.pem"
        tca_config.ssl_ca_cert = "/com/vmware/sdk-trials/sdk-py/ca_cert.pem"
        self.hybridity_sessionmgr = HybriditySessionManager()
        session_manager = hybridity_sessionmgr
        self.sdk_manager = SDKManager(). \
            with_configuration(tca_config). \
            with_session_manager(session_manager)
        self.api = sdk_manager.get_api('tca.nfv_vim_v1:VIMApi')

        # Print all VIMS
        vims = self.api.get_vims()
        vim_id = vims.items[0].vim_id
        print("VIMs: ", vims)
```

Below is an example to register a new VIM (generic Kubernetes).

```
        # sdk_manager from the previous example
        self.api = sdk_manager.get_api('tca.nfv_vim_v1:VIMApi')
        # Register a New Kubernetes VIM
        register_body = RegisterVim()

        # Whereabouts of the TCA-CP instance
        register_body.hcx_cloud_url = 'https://hcx_url'
        register_body.vim_name = 'a_name_to_identify_your_cluster_as_vim'
        register_body.cluster_name = 'your_k8s_cluster_name'
        register_body.auth_type = 'kubeconfig'
        # Base64 encoded Kubeconfig content (shortened for brevity)
        register_body.kubeconfig = 'YXBpVmVyc2lvb … jogdjEKY2x1c3RlXUVZSRklFdEZXUzB0TFMwdENnPT0='
        resp = self.api.register_vim(
                tenant_id='D926285F102B46FC9F5FD7DF7367CEF0',
                body=register_body)
        print("Regiter VIM Resp: ", resp)
```

Similarly, you may update an already registered VIM, using the method `modify_vim` on the Api client.

Managing CaaS

4

Invoking APIs for Container Infrastructure Management

TCA™CaaS Management APIs allow you to templatize Kubernetes clusters and thereby allowing you to create multiple K8S clusters of similar configurations. The CaaS also auto-registers the K8S clusters as VIMs to the TCA server.

This chapter includes the following topics:

This chapter includes the following topics:

- Creating and Managing Cluster Templates
- Creating and Managing K8S Clusters from Templates
- Managing 'add-on' to the K8S Workload-Cluster
- Upgrading K8S Clusters
- Fetching K8S Workload-Cluster Artifacts
- Managing CaaS K8S-Cluster Node-Pools

Creating and Managing Cluster Templates

Creating and Managing K8S Infrastructure - Cluster Templates

TCA allows creating and managing templates that hold the blueprint for creating K8S Clusters of a specific configuration, in a uniform manner. The below sample code lists and manages Cluster-Templates.

```
tca_config = TCAConfiguration()
# Ensure no trailing slash '/' in the tca_server
```

```
tca_config.set_server('https://your_tca_server')
tca_config.set_username('tenant@your_domain.com')
tca_config.set_password('**********************')

hybridity_sessionmgr = HybriditySessionManager()
self.sdk_manager = SDKManager(). \
        with_configuration(tca_config). \
        with_session_manager(hybridity_sessionmgr)
self.cluster_template_client = self.sdk_manager.get_api(
        'tca.clustertemplate:InfraClusterTemplateApi')

# The cluster-template id may be fetched from the list of clusters
## fetched using 'get_clusters_templates' operation.
templates = self.cluster_template_client.get_cluster_template(
        id='7e8d9f63-fbb2-4dfd-be63-955073f94840')
```

To list the operations available on the Api Client *cluster_ template_ client*:

```
self.cluster_template_client.list()
```

An example to create a Cluster Template is shown below. Please note, the creation of the *sdk_ manager* is abridged for brevity. The same my be referred in the previous examples.

```
from tca.cluster import ClusterConfigurations, ClusterNetworks, \
    ClusterMasterNodes, ClusterWorkerNodes
from tca.clustertemplate import ClusterTemplate, \
    ClusterTemplateClusterConfigCni, \
    ClusterTemplateClusterConfigCsi, \
    ClusterTemplateClusterConfigProperties, \
    ClusterTemplateConfig, ClusterTemplateConfigCpuManagerPolicy

# Fetch sdk_manager as shown in the previous examples
client = self.sdk_manager.get_api(
    'tca.clustertemplate:ClusterTemplatesApi')

def create_cluster_template(setup):
    # Prepping K8S Cluster Creation - MANAGEMENT CLUSTER-TEMPLATE
    cluster_config = ClusterConfigurations(
            kubernetes_version="v1.21.8+vmware.1")
    networkmodel = []
    networkmodel.append(ClusterNetworks(label='MANAGEMENT'))
    master_node = []
    master_node.append(ClusterMasterNodes(name='master',
             storage=50, cpu=8, memory=16384, replica=3, networks=networkmodel))
    worker_nodes = []
    worker_nodes.append(ClusterWorkerNodes(name='np1',
            replica=2, labels=[], memory=16384, cpu=8,
            storage=80, networks=networkmodel))
    body = ClusterTemplate(name="sdk-test-management-1",
            cluster_type='MANAGEMENT',
            cluster_config=cluster_config, master_nodes=master_node,
            worker_nodes=worker_nodes,
            description="sdk-check", tags=[])
```

```
    # Create Request -  MANAGEMENT CLUSTER-TEMPLATE
    cluster_template_create = client.create_cluster_template(body)
    # Prepping K8S Cluster Creation - WORKLOAD CLUSTER-TEMPLATE
    cni = []    cni.append(ClusterTemplateClusterConfigCni(name='calico'))
    csi = []    csi.append(ClusterTemplateClusterConfigCsi(
                    name='vsphere-csi',
                    properties=ClusterTemplateClusterConfigProperties(
                         name="vsphere-sc", timeout="300",
is_default=True)))
    cluster_config = ClusterConfigurations(
            kubernetes_version="v1.21.8+vmware.1", csi=csi, cni=cni)
    body = {}
    body['config'] = ClusterTemplateConfig(
            cpu_manager_policy=ClusterTemplateConfigCpuManagerPolicy(
                         policy='default',type='kubernetes'))
    worker_nodes = []
    worker_nodes.append(ClusterWorkerNodes(
            name='np1', replica=2, labels=["sdk=wc"],
            memory=16384, cpu=8, storage=80, networks=networkmodel))
    body = ClusterTemplate(name="sdk-test-workload-1",
            cluster_type='WORKLOAD', cluster_config=cluster_config,
            master_nodes=master_node, worker_nodes=worker_nodes,
            description="sdk-check", tags=[])

    # Create Request -  WORKLOAD CLUSTER-TEMPLATE
    cluster_template_create = client.create_cluster_template(body)
    print("created cluster template id: {}".\
            format(cluster_template_create.id))
```

Creating and Managing K8S Clusters from Templates

Creating and Managing K8S Infrastructure - Cluster Instances from Templates

After you created templates, you can use those to create and manage K8S clusters of same (or desired) configurations. Please note, the creation of the *sdk_ manager* is abridged for brevity. The same my be referred in the previous examples.

```
# Example: Create Cluster Method
def create_cluster(self):
    # Create Cluster Connfig Spec
    cluster_metadata = ClusterMetadata(resource_version="rv",
             name="name",
             mgmt_cluster_name="mgmt_name", tca_cp_id="id")
    cluster_config_spec_cidr_blocks_pods = ["100.96.0.0/11"]
    cluster_config_spec_cluster_pods = \
        ClusterSpecClusterConfigSpecClusterNetworkPods(
            cidr_blocks=cluster_config_spec_cidr_blocks_pods)
    cluster_config_spec_cidr_blocks_services = ["100.64.0.0/13"]
    cluster_config_spec_services = \
        ClusterSpecClusterConfigSpecClusterNetworkServices(
            cidr_blocks=cluster_config_spec_cidr_blocks_services)
    cluster_config_spec_cluster_network = \
        ClusterSpecClusterConfigSpecClusterNetwork(
```

```
            api_server_port=123,
            pods=cluster_config_spec_cluster_pods,
            service_domain=None,
            services=cluster_config_spec_services)
    cluster_config_spec_control_plane_endpoint = \
        ClusterSpecClusterConfigSpecControlPlaneEndpoint(
            host="192.168.111.181")
    cluster_config_spec_public_key_ref = \
        ClusterSpecClusterConfigSpecPublicKeyRef(
            api_version=None, field_path=None,
            kind=None, name=None, namespace=None,
            resource_version=None, uid=None)
    cluster_config_spec_rolling_update = \
        ClusterSpecClusterConfigSpecStrategyRollingUpdate(
            max_surge=1, max_unavailable=0)
    cluster_config_spec_strategy = \
        ClusterSpecClusterConfigSpecStrategy(
            rolling_update=cluster_config_spec_rolling_update, type=None)
    ## Please check TCA API docs, for the TKG BOM Release build
    cluster_config_spec_tca_bom_release_ref = \
        ClusterSpecClusterConfigSpecTcaBomReleaseRef(
            name="tbr-bom-2.0.0-v1.21.2---vmware.1-tkg.1-tca.18962437")
    cluster_config_spec_prime_ref = \
        ClusterSpecClusterConfigSpecCloudProvidersPrimeRef(
            vim_id="29f6a010-958d-11ec-b909-0242ac120002")
    cluster_config_spec_cloud_providers = \
        ClusterSpecClusterConfigSpecCloudProviders(
            prime_ref=cluster_config_spec_prime_ref, sub_refs=None)
    cluster_config_spec_proxy_spec = \
        ClusterSpecClusterConfigSpecProxySpec(type="extension",
            extension_id="29f6a010-958d-11ec-b909-0242ac120001",
            http_proxy=None, https_proxy=None,
            noproxy=None, ca_cert=None)
    cluster_config_spec_airgap_spec = \
        ClusterSpecClusterConfigSpecAirgapSpec(type=None,
            extension_id="29f6a010-958d-11ec-b909-0242ac120001",
            fqdn=None, ca_cert=None)
    cluster_config_spec = ClusterSpecClusterConfigSpec(
            cluster_network=cluster_config_spec_cluster_network,
            cni_type="calico",
            control_plane_endpoint=\
                cluster_config_spec_control_plane_endpoint,
            public_key_ref=cluster_config_spec_public_key_ref,
            strategy=cluster_config_spec_strategy,
            tca_bom_release_ref=cluster_config_spec_tca_bom_release_ref,
  cloud_providers=cluster_config_spec_cloud_providers,
proxy_spec=cluster_config_spec_proxy_spec,
            airgap_spec=cluster_config_spec_airgap_spec, ip_families=None)
    control_plane_spec_v_sphere_machine = \
        ClusterSpecControlPlaneSpecCloudMachineTemplateVSphereMachine(
            clone_mode="linkedClone", datacenter="os-test-dc",
            datastore="sharedVmfs-0 ",
            folder="ndc_master_0", num_cores_per_socket=None,
            resource_pool="ndc-compute-rp",
            storage_policy_name=None,
```

```
            template=\
                "/datacenter/template/photon-3-kube-v1.21.2+vmware.1")
    control_plane_spec_cloud_machine_template = \
        ClusterSpecControlPlaneSpecCloudMachineTemplate(
            type="VSphereMachine",
            v_sphere_machine=control_plane_spec_v_sphere_machine)
    control_plane_spec_strategy = \
        ClusterSpecControlPlaneSpecStrategy(rolling_update=None,
            type=None)
    control_plane_spec_tca_bom_release_ref = \
        ClusterSpecControlPlaneSpecTcaBomReleaseRef(
            name="tbr-bom-2.0.0-v1.21.2---vmware.1-tkg.1-tca.18962437")
    # Create Control Plane Spec
    control_plane_spec = \
        ClusterSpecControlPlaneSpec(clone_mode=None,
            cloud_machine_template=\
                 control_plane_spec_cloud_machine_template,
            cluster_name=None, datacenter=None,
            datastore=None, disk_gi_b=30,
            folder=None, kubeadm_config_template=None,
            labels=None, memory_mi_b=8192,
            network=None, num_cp_us=2,
            num_cores_per_socket=None, replicas=1,
            resource_pool=None, storage_policy_name=None,
            strategy=control_plane_spec_strategy,
            tca_bom_release_ref=control_plane_spec_tca_bom_release_ref,
template=None, cloud_provider=None)
    cluster_spec = ClusterSpec(
            cluster_config_spec=cluster_config_spec,
            control_plane_spec=control_plane_spec)
    body = Cluster(metadata=cluster_metadata,
            spec=cluster_spec, status=None)

    # Invoke the API to create the cluster...
    create_cluster = self.cluster_automation_client.create_cluster(body)
    print("CLUSTER CREATION RESPONSE: ", create_cluster)

def get_addon_tca_k8s_resources(self):
    get_cluster_addon_tca_k8s_resources = self.cluster_automation_client.
            get_addon_tca_k8s_resources(cluster_name="test",
            addon_name="addonname",
            # The tca_cp_id corresponds to the CP instance that manages
            ## the K8S cluster as NFV-VIM.
            mgmt_cluster_name="mgmttest", tca_cp_id="1234")

def get_cluster_tca_k8s_resources(self):
    get_cluster_tca_k8s_resources = \
        self.cluster_automation_client.get_cluster_tca_k8s_resources(
                cluster_name="test", mgmt_cluster_name="mgmttest",
                tca_cp_id="1234")

def get_node_pool_tca_k8s_resources(self):
    get_cluster_node_pool_tca_k8s_resources = \
        self.cluster_automation_client.get_node_pool_tca_k8s_resources(
```

```
                cluster_name="test", node_pool_name="nodepoolname",
                mgmt_cluster_name="mgmttest", tca_cp_id="1234")
```

The k8s workload cluster is identified using three parameters: workload-cluster-name, manamenegt-cluster-name and the CP-id. For details refer to the TCA™ product documentation.

The CP-id may be fetched as shown in the examples in the next section.

Managing 'add-on' to the K8S Workload-Cluster

Reconfiguring the workload cluster - 'add-ons'.

The example (a unit-test) below makes use of multiple API-clients to fetch the relevant infomation (CP-id, etc) and use it to configure add-on.

```
from __future__ import absolute_import

import unittest
from api.client.sdk_manager import SDKManager
from api.client.session_manager import HybridditySessionManager
from caas_v2_addon import Addon, AddonMetadata, AddonSpec, AddonSpecClusterRef,
AddonSpecConfig
from cluster import ClustersQueryBody
from tca.api.client.tca_configuration import TCAConfiguration


class TestAddonApi(unittest.TestCase):
    """AddonApi unit test stubs"""

    def setUp(self):
        tca_config = TCAConfiguration()
        # Ensure no trailing slash '/' in the tca_server
        tca_config.set_server('https://tca_server')
        tca_config.set_username('tenant@vsphere.local')
        tca_config.set_password('********************')
        # Insecure: Not recommended for production.
        tca_config.set_verify_ssl(False)

        hybridity_sessionmgr = HybridditySessionManager()
        self.sdk_manager = SDKManager(). \
            with_configuration(tca_config). \
            with_session_manager(hybridity_sessionmgr)

        self.caas_addon_client = self.sdk_manager.get_api(
                'tca.caas_v2_addon:AddonApi')
        self.cluster_client = self.sdk_manager.get_api(
                'tca.cluster:ClusterApi')

    def tearDown(self):
        pass

    def test_create_addon(self):
        workload_cluster_name = ''
        mgmt_cluster_name = ''
```

```
        add_on_name = ''
        harbor_extension_id = ''

        # FETCHING TCA CP-ID
        tca_cp_id = self.get_hcxuuid_for_mgmt_cluster(
                mgmt_cluster_name=mgmt_cluster_name)
        add_on_metadata = AddonMetadata(name=add_on_name,
                cluster_name=workload_cluster_name)
        cluster_ref = AddonSpecClusterRef(
                name=workload_cluster_name,
                namespace=workload_cluster_name)
        config = AddonSpecConfig(partner_system_refs=[harbor_extension_id])
        add_on_spec = AddonSpec(name=add_on_name,
                cluster_ref=cluster_ref, config=config)
        add_on_payload = Addon(metadata=add_on_metadata, spec=add_on_spec)
        self.caas_addon_client.create_addon(add_on_payload,
                cluster_name=workload_cluster_name,
                mgmt_cluster_name=mgmt_cluster_name,
                tca_cp_id=tca_cp_id)

    # FETCH TCA CP-ID FOR THE GIVEN MGMT-CLUSTER
    def get_hcxuuid_for_mgmt_cluster(self, mgmt_cluster_name):
        mgmt_cluster_filter = ClustersQueryBody(
                filter={"clusterName": mgmt_cluster_name})
        cluster_infra_cluster_query_response = self.cluster_client.\
                infra_cluster_query(body=mgmt_cluster_filter)
        mgmt_cluster = cluster_infra_cluster_query_response[0]
        return mgmt_cluster.hcx_uuid
```

Upgrading K8S Clusters

Sample code to Edit or Upgrade a K8S Workload Cluster

The upgrade of a K8S cluster (as a VIM) associated to the TCA-CP, is model as an edit operation using TCA™ CaaS V2 APIs. The example Unit-test below depicts the upgrade operation on a workload cluster. The code also shows how to fetch the CP-id to access a workload cluster.

```
from __future__ import absolute_import

import ast
import unittest
from api.client.sdk_manager import SDKManager
from api.client.session_manager import HybridityTSessionManager
from caas_v2 import ClusterSpec, ClusterSpecClusterConfigSpec,\
    ClusterSpecControlPlaneSpec, \
    ClusterSpecClusterConfigSpecClusterNetwork,\
    ClusterSpecClusterConfigSpecClusterNetworkPods, \
    ClusterSpecClusterConfigSpecClusterNetworkServices,\
    ClusterSpecClusterConfigSpecControlPlaneEndpoint, \
    ClusterSpecClusterConfigSpecStrategyRollingUpdate,\
    ClusterSpecClusterConfigSpecStrategy, \
    ClusterSpecClusterConfigSpecTcaBomReleaseRef,\
    ClusterSpecClusterConfigSpecCloudProviders, \
```

```
    ClusterSpecClusterConfigSpecCloudProvidersPrimeRef,\
    ClusterSpecControlPlaneSpecCloudProvider, \
    ClusterSpecControlPlaneSpecCloudMachineTemplate,\
    ClusterSpecControlPlaneSpecCloudMachineTemplateVSphereMachine, \
    ClusterSpecControlPlaneSpecNetwork,
    ClusterSpecControlPlaneSpecNetworkDevices, Cluster
from cluster import ClustersQueryBody
from tca.api.client.tca_configuration import TCAConfiguration


class TestClusterAutomationApi(unittest.TestCase):

    def setUp(self):
        tca_config = TCAConfiguration()
        # Ensure no trailing slash '/' in the tca_server
        tca_config.set_server('https://tca_server')
        tca_config.set_username('tenant@vsphere.local')
        tca_config.set_password('********************')
        # Insecure: Not recommended for production
        tca_config.set_verify_ssl(False)

        hybridity_sessionmgr = HybriditySessionManager()
        self.sdk_manager = SDKManager(). \
            with_configuration(tca_config). \
            with_session_manager(hybridity_sessionmgr)

        self.cluster_automation_api_client = self.sdk_manager.get_api(
                'tca.caas_v2:ClusterAutomationApi')
        self.generic_cr_api = self.sdk_manager.get_api(
                'tca.caas_k8s_generic_cr:GenericCrApi')
        self.cluster_client = self.sdk_manager.get_api(
                'tca.cluster:ClusterApi')

    def build_control_plane_spec(self, cluster_config_spec, vim_id, data_center):
        clone_mode = ''
        control_plane_name = 'test-cluster-control-plane'
        cloud_provider = ClusterSpecControlPlaneSpecCloudProvider(
                vim_id=vim_id)
        cluster_name = 'test-cluster'
        disk_gib = 30
        tca_bom_release_ref = cluster_config_spec.tca_bom_release_ref
        memory_mi_b = 8192
        num_cpus = 8
        vm_template_path_for_kubernetes_version = ''
        data_store = ''
        resource_pool = ''
        folder = ''
        nameservers = []
        network_name = ''
        devices = ClusterSpecControlPlaneSpecNetworkDevices(
                dhcp4=True, network_name=network_name,
                nameservers=nameservers)
        network = ClusterSpecControlPlaneSpecNetwork(devices=[devices])
        vsphere_machine = \
            ClusterSpecControlPlaneSpecCloudMachineTemplateVSphereMachine(
```

```
datacenter=data_center,


datastore=data_store,


resource_pool=resource_pool,


template=vm_template_path_for_kubernetes_version,

                folder=folder)

        cloud_machine_template = \
            ClusterSpecControlPlaneSpecCloudMachineTemplate(
                v_sphere_machine=vsphere_machine)
        return ClusterSpecControlPlaneSpec(
                clone_mode=clone_mode,
                control_plane_name=control_plane_name,
                cloud_provider=cloud_provider, cluster_name=cluster_name,
                disk_gi_b=disk_gib,
                tca_bom_release_ref=tca_bom_release_ref,
                memory_mi_b=memory_mi_b, num_cp_us=num_cpus,
                network=network,
                cloud_machine_template=cloud_machine_template)

    def build_cluster_config_spec(self, mgmt_cluster_name,
                kubernetes_version_for_cluster,
                vim_id, data_center):
        host_ip = ''
        cluster_network = self.build_cluster_network()
        control_plane_endpoint = \
            ClusterSpecClusterConfigSpecControlPlaneEndpoint(
                host=host_ip)
        strategy = self.get_cluster_config_spec_strategy()
        tca_bom_release_ref = self.build_tca_bom_release_ref(
                mgmt_cluster_name=mgmt_cluster_name,
                kubernetes_version_for_cluster= \
                    kubernetes_version_for_cluster)
        cloud_providers = self.build_cloud_providers_ref(vim_id,
                    data_center)
        return ClusterSpecClusterConfigSpec(
                cluster_network=cluster_network,
                control_plane_endpoint=control_plane_endpoint,
                cni_type='antrea',
strategy=strategy,
                tca_bom_release_ref=tca_bom_release_ref,
                cloud_providers=cloud_providers)

    def build_cloud_providers_ref(self, vim_id, data_center):
        prime_ref = \
            ClusterSpecClusterConfigSpecCloudProvidersPrimeRef(
                vim_id=vim_id, datacenter=data_center)
        return ClusterSpecClusterConfigSpecCloudProviders(
```

```
                    prime_ref=prime_ref)

    # Build the BOM for the cluster upgrade
    def build_tca_bom_release_ref(self, mgmt_cluster_name,
                kubernetes_version_for_cluster):
        mgmt_cluster_filter = ClustersQueryBody(
                filter={"clusterName": mgmt_cluster_name})
        cluster_infra_cluster_query_response = self.cluster_client.\
                infra_cluster_query(body=mgmt_cluster_filter)
        mgmt_cluster = cluster_infra_cluster_query_response[0]
        mgmt_cluster_name = mgmt_cluster.cluster_name
        tca_cp_id = self.get_tca_cp_id_for_mgmt_cluster(
                mgmt_cluster_name=mgmt_cluster_name)
        self.tca_cp_id = tca_cp_id
        tca_bom_list = self.generic_cr_api.\

list_tca_bom_releases(tca_cp_id=tca_cp_id,

                mgmt_cluster_name=mgmt_cluster_name)
        items = ast.literal_eval(tca_bom_list)["items"]
        tca_bom_ref = None
        for item in items:
            if kubernetes_version_for_cluster in item['metadata']['name']:
                tca_bom_ref = item
                break
        tca_bom_ref_name = tca_bom_ref['metadata']['name']
        return ClusterSpecClusterConfigSpecTcaBomReleaseRef(
                name=tca_bom_ref_name)

    def get_cluster_config_spec_strategy(self):
        rolling_update = ClusterSpecClusterConfigSpecStrategyRollingUpdate(
                max_surge=1, max_unavailable=0)
        return ClusterSpecClusterConfigSpecStrategy(
                type='RollingUpdate', rolling_update=rolling_update)

    def build_cluster_network(self):
        pods = ClusterSpecClusterConfigSpecClusterNetworkPods(
                cidr_blocks=['100.96.0.0/11'])
        services = ClusterSpecClusterConfigSpecClusterNetworkServices(
                cidr_blocks=['100.64.0.0/13'])
        return ClusterSpecClusterConfigSpecClusterNetwork(
                pods=pods, services=services)

    # FETCH THE TCA CP-ID
    def get_tca_cp_id_for_mgmt_cluster(self, mgmt_cluster_name):
        mgmt_cluster_filter = ClustersQueryBody(
                filter={"clusterName": mgmt_cluster_name})
        cluster_infra_cluster_query_response = self.cluster_client.infra_cluster_query(
                body=mgmt_cluster_filter)
        mgmt_cluster = cluster_infra_cluster_query_response[0]
        return mgmt_cluster.hcx_uuid


    def test_upgrade_cluster(self):
        workload_cluster_name = ''
```

```
        desired_kubernetes_version_for_cluster = 'v1.21.11'
        vm_template_path_for_desired_kubernetes_version = ''
        mgmt_cluster_name = ''
        desired_bom_release_ref = self.build_tca_bom_release_ref(
                mgmt_cluster_name=mgmt_cluster_name,
                kubernetes_version_for_cluster=\
                desired_kubernetes_version_for_cluster)
        tca_cp_id = self.get_tca_cp_id_for_mgmt_cluster(
                mgmt_cluster_name=mgmt_cluster_name)
        current_cluster = self.cluster_automation_api_client.get_cluster(

cluster_name=workload_cluster_name,
                mgmt_cluster_name=mgmt_cluster_name,
                tca_cp_id=tca_cp_id)
        current_cluster.spec.cluster_config_spec.tca_bom_release_ref = \
                desired_bom_release_ref
        current_cluster.spec.\
            control_plane_spec.cloud_machine_template.\
            v_sphere_machine.template = \
                vm_template_path_for_desired_kubernetes_version
        current_cluster.spec.\
            control_plane_spec.tca_bom_release_ref = \
                desired_bom_release_ref
        self.cluster_automation_api_client.edit_cluster(
                current_cluster,
                cluster_name=workload_cluster_name,

mgmt_cluster_name=mgmt_cluster_name,

                tca_cp_id=tca_cp_id)

    def test_get_all_clusters(self):
        data = self.cluster_automation_api_client.\
            get_all_clusters(async_req=False)
        print("get all clusters response is", data)

    def test_get_cluster(self):
        workload_cluster_name = 'workload-v2'
        mgmt_cluster_name = 'mgmt-cluster'
        tca_cp_id = self.get_tca_cp_id_for_mgmt_cluster(
                mgmt_cluster_name=mgmt_cluster_name)
        data = self.cluster_automation_api_client.

get_cluster(cluster_name=workload_cluster_name,

mgmt_cluster_name=mgmt_cluster_name,
                tca_cp_id=tca_cp_id, async_req=False)
        print("get cluster response is", data)
```

The upgrade specification is made via the BOM, which is referenced duing the edit operation. For details, refer to the TCA™ product documentation.

Fetching K8S Workload-Cluster Artifacts

Sample code to fetch the workload cluster artifacts: BOM, kubeconfig, CP-id etc.

Below is an example code (unit-tests) to fetch the details of a K8S workload-cluster.

```
from __future__ import absolute_import

import unittest

from cluster import ClustersQueryBody
from tca.api.client.sdk_manager import SDKManager
from tca.api.client.session_manager import Hybridity SessionManager
from tca.api.client.tca_configuration import TCAConfiguration


class TestGenericCrApi(unittest.TestCase):

    def setUp(self):
        tca_config = TCAConfiguration()
        tca_config.set_server('https://tca_server')
        tca_config.set_username('tenant@vsphere.local')
        tca_config.set_password('********************')
        # Insecure: Not recommended for production
        tca_config.set_verify_ssl(False)

        hybridity_sessionmgr = HybriditySessionManager()
        self.sdk_manager = SDKManager(). \
            with_configuration(tca_config). \
            with_session_manager(hybridity_sessionmgr)

        self.generic_cr_api_client = self.sdk_manager.get_api(
                'tca.caas_k8s_generic_cr:GenericCrApi')
        self.cluster_client = self.sdk_manager.get_api(
                'tca.cluster:ClusterApi')

    def test_get_workload_cluster_kube_config(self):
        workload_cluster_name = ''
        mgmt_cluster_name = ''
        tca_cp_id = self.get_hcxuuid_for_mgmt_cluster(
                mgmt_cluster_name=mgmt_cluster_name)
        workload_cluster_kubeconfig = self.generic_cr_api_client.\
            get_workload_cluster_kube_config(
                workload_cluster_name=workload_cluster_name,
                secret_name=workload_cluster_name + '-kubeconfig',
                mgmt_cluster_name=mgmt_cluster_name,
                tca_cp_id=tca_cp_id)
        print("For cluster ", workload_cluster_name, "kubeconfig is:%s",
                workload_cluster_kubeconfig)

    # FETCH TCA CP-ID FOR THE GIVEN MGMT-CLUSTER
    def get_hcxuuid_for_mgmt_cluster(self, mgmt_cluster_name):
        mgmt_cluster_filter = ClustersQueryBody(
                filter={"clusterName": mgmt_cluster_name})
        cluster_infra_cluster_query_response = self.cluster_client.\
```

```
            infra_cluster_query(
                body=mgmt_cluster_filter)
        mgmt_cluster = cluster_infra_cluster_query_response[0]
        return mgmt_cluster.hcx_uuid

    def test_list_tca_bom_releases(self):
        mgmt_cluster_filter = ClustersQueryBody(
                filter={"clusterType": "MANAGEMENT"})
        cluster_infra_cluster_query_response = self.cluster_client.\
            infra_cluster_query(body=mgmt_cluster_filter)
        cluster = cluster_infra_cluster_query_response[0]
        mgmt_cluster_name = cluster.cluster_name
        tca_cp_id = cluster.hcx_uuid
        tca_bom_list = self.generic_cr_api_client.\

list_tca_bom_releases(tca_cp_id=tca_cp_id,

                mgmt_cluster_name=mgmt_cluster_name)
        print("For cluster ", mgmt_cluster_name,
                " TCA BOM list is:%s", tca_bom_list)

if __name__ == '__main__':
    unittest.main()
```

Managing CaaS K8S-Cluster Node-Pools

Sample code to manage the K8S workload cluster node-pools.

Below is a Unit-test code to create and manage K8S Cluster node-pools.

```
from __future__ import absolute_import

import ast
import unittest

from api.client.sdk_manager import SDKManager
from api.client.session_manager import HybriditySessionManager
from caas_v2_nodepool import Nodepool, NodepoolMetadata, \
    NodepoolSpec, NodepoolSpecCloudMachineTemplate, \
    NodepoolSpecCloudMachineTemplateVSphereMachine,
    NodepoolSpecTcaBomReleaseRef, NodepoolSpecNetwork, \
    NodepoolSpecNetworkDevices, NodepoolSpecKubeadmConfigTemplate, \
    NodepoolSpecKubeadmConfigTemplateJoinConfiguration, \
    NodepoolSpecKubeadmConfigTemplateJoinConfigurationNodeRegistration, \
    NodepoolSpecCloudProvider
from cluster import ClustersQueryBody
from tca.api.client.tca_configuration import TCAConfiguration


class TestNodePoolApi(unittest.TestCase):
    """NodePoolApi unit test stubs"""

    def setUp(self):
        tca_config = TCAConfiguration()
```

```
        tca_config.set_server('https://tca_server')
        tca_config.set_username('tenant@vsphere.local')
        tca_config.set_password('********************')
        # Insecure: Not recommended for production
        tca_config.set_verify_ssl(False)

        hybridity_sessionmgr = HybriditySessionManager()
        self.sdk_manager = SDKManager(). \
            with_configuration(tca_config). \
            with_session_manager(hybridity_sessionmgr)

        self.nodepool_api_client = self.sdk_manager.get_api(
                'tca.caas_v2_nodepool:NodePoolApi')
        self.generic_cr_api = self.sdk_manager.get_api(
                'tca.caas_k8s_generic_cr:GenericCrApi')
        self.cluster_client = self.sdk_manager.get_api(
                'tca.cluster:ClusterApi')

    def test_create_node_pool(self):
        workload_cluster_name = 'test-cluster'
        self.mgmt_cluster_name = ''
        self.kubernetes_version_for_cluster = 'v1.20.15'
        nodepool_metadata = NodepoolMetadata(
                name=workload_cluster_name + '-np1')
        vim_id = ''
        datacenter = ''
        resource_pool = ''
        datastore = ''
        storage_policy_name = ''
        folder = ''
        template = ''
        network_name = ''
        name_servers = []
        vsphere_machine = NodepoolSpecCloudMachineTemplateVSphereMachine(

datacenter=datacenter,
                resource_pool=resource_pool,
                datastore=datastore,

storage_policy_name=storage_policy_name,
                folder=folder, template=template)
        cloud_machine_template = NodepoolSpecCloudMachineTemplate(
                v_sphere_machine=vsphere_machine)
        tca_bom_release_ref = self.build_tca_bom_release_ref()
        devices = NodepoolSpecNetworkDevices(
                dhcp4=True, network_name=network_name,
                nameservers=name_servers)
                network = NodepoolSpecNetwork(devices=[devices])

        kubelet_extra_args = {'cpu-manager-policy': 'static',
                'system-reserved': 'cpu=1,memory=1Gi'}
        node_registration =
NodepoolSpecKubeadmConfigTemplateJoinConfigurationNodeRegistration(
            kubelet_extra_args=kubelet_extra_args)
        join_configuration = NodepoolSpecKubeadmConfigTemplateJoinConfiguration(
```

```
                node_registration=node_registration)
        kube_admin_config_template = NodepoolSpecKubeadmConfigTemplate(
                join_configuration=join_configuration)
        cloud_provider = NodepoolSpecCloudProvider(vim_id=vim_id)
        spec = NodepoolSpec(clone_mode='linkedClone',
                cloud_machine_template=cloud_machine_template,
                cluster_name=workload_cluster_name,
                disk_gi_b=30, memory_mi_b=8192, num_cp_us=8,
                tca_bom_release_ref=tca_bom_release_ref,
                network=network,
                kubeadm_config_template=kube_admin_config_template,
                cloud_provider=cloud_provider)

        nodepool_payload = Nodepool(metadata=nodepool_metadata, spec=spec)
        tca_cp_id = self.get_hcxuuid_for_mgmt_cluster(
                mgmt_cluster_name=self.mgmt_cluster_name)
        self.nodepool_api_client.create_node_pool(
                nodepool_payload, cluster_name=workload_cluster_name,
                mgmt_cluster_name=self.mgmt_cluster_name,
                tca_cp_id=tca_cp_id)

    def build_tca_bom_release_ref(self):
        mgmt_cluster_filter = ClustersQueryBody(
                filter={"clusterName": self.mgmt_cluster_name})
        cluster_infra_cluster_query_response = self.cluster_client.infra_cluster_query(
                body=mgmt_cluster_filter)
        mgmt_cluster = cluster_infra_cluster_query_response[0]
        mgmt_cluster_name = mgmt_cluster.cluster_name
        tca_cp_id = mgmt_cluster.hcx_uuid
        self.tca_cp_id = tca_cp_id
        tca_bom_list = self.generic_cr_api.list_tca_bom_releases(
                tca_cp_id=tca_cp_id,

mgmt_cluster_name=mgmt_cluster_name)
        items = ast.literal_eval(tca_bom_list)["items"]
        tca_bom_ref = None
        for item in items:
            if self.kubernetes_version_for_cluster in \
                        item['metadata']['name']:
                tca_bom_ref = item
                break
        tca_bom_ref_name = tca_bom_ref['metadata']['name']
        return NodepoolSpecTcaBomReleaseRef(name=tca_bom_ref_name)

    # FETCH TCA CP-ID FOR THE GIVEN MGMT-CLUSTER
    def get_hcxuuid_for_mgmt_cluster(self, mgmt_cluster_name):
        mgmt_cluster_filter = ClustersQueryBody(
                filter={"clusterName": mgmt_cluster_name})
        cluster_infra_cluster_query_response = \
            self.cluster_client.infra_cluster_query(
                body=mgmt_cluster_filter)
        mgmt_cluster = cluster_infra_cluster_query_response[0]
        return mgmt_cluster.hcx_uuid
```

System and Appliance Management

5

Invoking APIs for TCA (Manager and Control-Plane) Administration

TCA Offers System Management APIs, that require Admin (or Super User) credentials. For more details, see Administering VMware TCA at http://docs.vmware.com.

While, there are a handful of Adinistration APIs supported by the TCA, this chapter focusses on two of those:

- System Backup and Restore
- Import Certificates to the TCA Trust Store

This chapter includes the following topics:

System Backup and Restore

TCA Appliance Backup and Restore

TCA Backup and Restore APIs can be used to take Appliance backups and restore at a later point of time. Below is an example of using SDK to backup and restore.

```
from tca.system_management import TcaJob, BackupJobDetails, BackupRequest

# TCA configuration with Admin/SU privilege
tca_config = TCAConfiguration() tca_config.set_server('https://tca_server')
tca_config.set_username('admin_login') tca_config.set_password('***********')

# Insecure: only for NFV functional validation
## For SSL communication, please refer to Invocation Using with SSL.
tca_config.set_verify_ssl(False)

# Does not require session management; it's HTTP Basic
Authsessionmgr = NoOpSessionManager()
sdk_manager = SDKManager(). \
    with_configuration(tca_config). \
    with_session_manager(sessionmgr)

# API Client for Backup and Restore
```

```
client = sdk_manager.get_api('tca.system_management:BackupAndRestoreApi')
body = BackupRequest(upload_to_server=False)
data = client.backup_tca_post(accept="application/json",
                content_type="application/json", body=body)

# Backup Job Id, which may be used to query the state of the Backup progress.job_id =
data.job_id
job = client.backup_job_status_job_id_get(job_id,
                accept="application/json")
assert type(data) is BackupJobDetails
# job.filename references the URI to download the backed up resource.
print("Job Status = %s, File: %s" % (job.state, job.filename))
```

Import Certificates to the TCA Trust Store

Invoking admin API to fetch and import Certificates into the TCA truststore

Often self-signed certificates are required to be imported to the TCA, for making REST calls to the external entities. The example below suggests how to list the trusted certificates and to import a new. The code assumes you have the sdk_manager instance as shown in the previous examples.

```
# API Client for Backup and Restoreclient =
sdk_manager.get_api('tca.system_management:CertificatesApi')
# Copy-Paste the certificate body
body = AdminCertificatesBody(
    certificate="-----BEGIN CERTIFICATE-----
MIIDjjCCAnYCCQDoSwcBlIOpnjANBgkqhkiG9w0BAQsF"
"ADCBiDELMAkGA1UEBhMC\nVVMxEzARBgNVBAgMCkNhbGlmb3JuaWExCzAJBgNVBAcMAkNBMQ8w"
…                "NifHQDDHB9DANQAXguGCDrEN+p1y0atpH3ol8u\nBV0=\n-----END CERTIFICATE-----")

resp = client.api_admin_certificates_post(body,
                accept="application/json",
content_type="application/json", async_req=False)

# Fetch and Print the fingerprint
ssha256 = resp.data.get('items')[0].\
        get('fingerprints').\
        get('sha256')
print(ssha256)

# Fetch the imported (trusted) certificates to verify the successful import
certs = client.api_admin_certificates_get(accept="application/json")
```