ONTAP® Select 9

Deploy API Guide

Using ONTAP Select Deploy 2.6

December 2017 | 215-12638_A0
doccomments@netapp.com

Updated for ONTAP Select 9.3

NetApp®

Contents

Deciding whether to use the Deploy API guide

This guide includes everything you need to understand and use the REST application programming interface provided with the ONTAP Select Deploy administration utility. You can use the API to deploy and administer ONTAP Select clusters.

> **Attention:** The Deploy API can change at any time. The API included with each release of the Deploy administration utility is assigned a version number. You should be aware of the version or versions supported by your release of the Deploy utility. The current release of the ONTAP Select Deploy administration utility includes version 2 (v2) of the API. Refer to the ONTAP Select *Release Notes* for any changes or updates to the API.

Related tasks

Accessing the Swagger API web page on page 10

Related references

Code samples on page 14

Related information

ONTAP Select 9.3 Release Notes

Understanding the ONTAP Select Deploy API

The ONTAP Select Deploy administration utility includes a REST web services API that provides access to the functionality you need when deploying and administering ONTAP Select nodes and clusters. Before using the Deploy API, you should understand its design, architectural components, and limitations.

REST web services foundation

Representational State Transfer (REST) establishes a collection of technologies and design principles for exposing server-based resources and managing their states. It uses mainstream protocols and standards, which provides a flexible and extensible foundation for managing ONTAP Select deployments.

Resources and state representation

The core aspects of RESTful web services design include the following:

- Identification of system or server-based resources
  Every system uses and maintains resources. A resource can be a file, business information, a process, or an administrative entity. One of the first tasks in designing an application based on RESTful web services is to identify the resources.
- Definition of resource states and associated state operations
  Resources are always in one of a finite number of states. The states must be clearly defined, as well as the operations used to affect the state changes.

Messages are exchanged between the client and server to access and change the state of the resources according to the generic CRUD (Create, Read, Update, and Delete) model.

HTTP messages

Hypertext Transfer Protocol (HTTP) is the specific protocol used by the web services client and server to exchange request and response messages about the resources. As part of designing an application based on RESTful web services, the HTTP verbs (such as, GET and POST) are mapped to the resources and corresponding state management actions.

HTTP is stateless. Therefore, to associate a set of related requests and responses under one identity, additional information is typically added to the data flows, including HTTP headers or cookies.

URI endpoints

Every REST resource must be defined and made available using a well-defined addressing scheme. The endpoints where the resources are located and identified use a Uniform Resource Identifier (URI). The URI provides a general framework for creating a unique name for each resource used in the network. Resources are exposed in a structure similar to a hierarchical file directory.

The Uniform Resource Locator (URL) is a type of URI used with RESTful web services to identify and access a resource.

JSON formatting

While information can be structured and transferred between a client and server in several ways, the most popular option (and the one used with the ONTAP Select Deploy API) is JavaScript Object Notation (JSON). JSON is a standard for representing simple data structures, including objects and

arrays, in plain text. JSON is used by RESTl web services to represent and transfer state information describing each resource.

Multiple access paths

Because of the inherent flexibility of RESTl web services, the ONTAP Select Deploy API can be accessed in several different ways:

- **Deploy utility native user interface** – The primary way you access the API is through the ONTAP Select Deploy native web user interface. The browser accesses the API and reformats the data according to the design of the user interface. You can also access the Deploy utility through the command line interface.
- **Swagger web page** – The ONTAP Select Deploy API uses the Swagger platform to provide an alternative access point using a browser. Swagger provides a description of each API call, including input parameters and other options.

  Attention: Any API operation you perform using the Swagger user interface is a live operation. You should be careful not to mistakenly create, update, or delete configuration or other data.

- **Custom program** – You can access the Deploy API using one of several different programming languages and tools. Popular choices include Python, Java, and cURL. A program, script, or tool that uses the API acts as a RESTful web services client. Using a programming language enables you to better understand the API as well as to automate an ONTAP Select deployment.

Basic operation

There are several operational characteristics of the ONTAP Select Deploy API that you should be aware before using the API.

Resource categories

The REST resources provided through by the Deploy API are organized in several different categories. You must refer to the Swagger web page for a complete list of the API calls and the details of each call.

Security

The Deploy API provides security using the following technologies:

- Transport Layer Security
  All traffic sent over the network between the Deploy server and client is encrypted using TLS/SSL. Using the HTTP protocol over an unencrypted channel is not supported.
- HTTP authentication
  Basic authentication is used. An HTTP header is added to every request which includes the administrator user name and password in a base 64 string.

Synchronous versus asynchronous requests

There are two primary ways that a server performs an HTTP request received from a client. In general, the type of request is based on the specific HTTP status code indicating success.

- Synchronous request
  The server performs the request immediately and responds with a status code of 200 or 204.
- Asynchronous request
  The server accepts the request and responds with a status code of 202. At the time of the response by the server, the request has not yet been completed and can subsequently fail.

Host status

Each ONTAP Select host has a status value that falls into one of two categories. You must first add a host, which is then assigned a status value reflecting the authentication of the host to the hypervisor. You can then configure the host, which is then assigned a new status value based on the result of the configuration action.

Cluster state

Each ONTAP Select cluster is composed of one or more hosts, and has an associated state.

How asynchronous operation works

Many of the API calls, particularly those that create a resource, can take a longer time to complete than most other API calls. ONTAP Select Deploy processes these types of requests asynchronously.

With asynchronous processing, the initial successful HTTP response indicates that the request has been accepted but not necessarily completed. Therefore, after making an asynchronous request, you must test the resource instance for completion of the request.

Note: You should refer to the Swagger web page for documentation to help determine whether a specific API call operates asynchronously.

Checking a resource after an add or create request

After issuing an API call that adds, creates or updates a resource instance, you should poll the status or state of the resource to verify completion of the request. A request is complete when the new resource reaches the appropriate status or state.

You should use the following high level process when asynchronously when adding or creating a new resource:

1. Issue the API call to add or create a resource.
2. Receive an HTTP response indicating successful acceptance of the request.
3. Within a loop, perform the following in each cycle:
   a. Get the current status or state of the resource.
   b. If the resource is not in the appropriate status or state, perform the loop again.
4. Stop when the resource reaches the appropriate status or state.

Checking for resource removal after a delete request

After issuing an API call that deletes a resource instance, you should poll the resource to verify that it has been removed. A request is complete when the resource no longer exists.

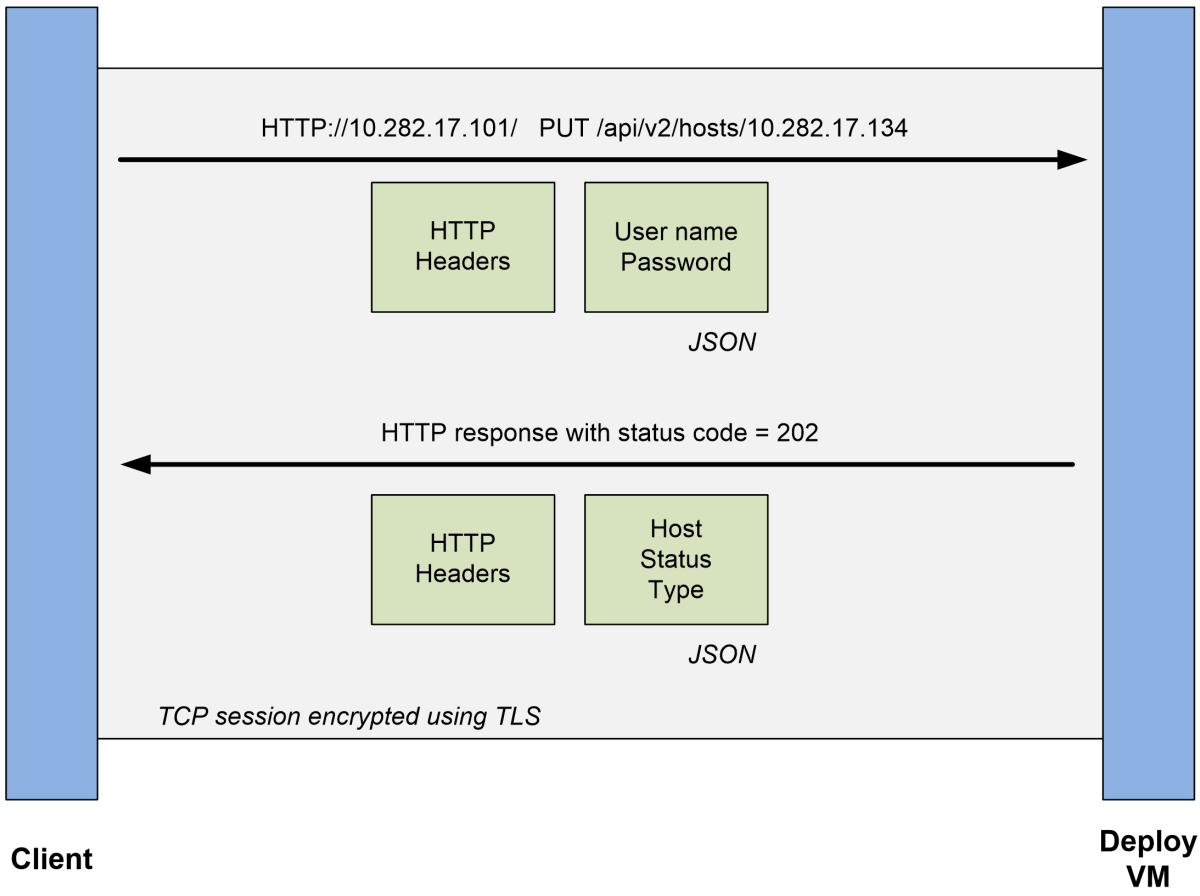
You should use the following high level process when asynchronously when asynchronously removing a resource:

1. Issue the API call to delete a resource.
2. Receive an HTTP response indicating successful acceptance of the request.
3. Within a loop, perform the following in each cycle:
   a. Get the current status or state of the resource.
   b. If resource is located, perform the loop again.
4. Stop when the get request is not found (HTTP code 404).

Illustration of a Deploy API call

The following figure illustrates a REST API call used to add an ESXi host at the Deploy utility virtual machine. In this example, Deploy authenticates the ESXi host and does not use VMware vCenter. The HTTP traffic exchanged between the client and Deploy VM is encrypted using TLS.

There are many programming languages available at the client. Python is a popular choice, especially for datacenter automation.



HTTP request

The request sent from the client consists of the following:

- HTTP PUT verb
- HTTP request headers, including authorization (basic authentication)
- JSON structure containing the ESXi host user name and password

HTTP response

The request sent from the client consists of the following:

- HTTP response with status code 202
- HTTP response headers
- JSON structure containing the host ID, status, and type

Note: The HTTP status code 202 indicates the request has been accepted by Deploy but not yet completed.

Related concepts

Accessing and using the API with Python on page 12

Accessing and using the API with a browser

The Deploy API is exposed through the Swagger web page at an ONTAP Select Deploy virtual machine. The API calls are organized on the page according to resource type and displayed in a consistent format. You should understand how to interpret the API documentation on the Swagger page, as well as how to issue an API call.

Accessing the Swagger API web page

You must access the Swagger web page to display the API documentation, as well as to manually issue an API call.

Before you begin

You must have the following:

- IP address or domain name of the ONTAP Select Deploy virtual machine
- User name and password for administrator

About this task

Version 2 (v2) of the Deploy API is used.

Steps

1. Type the URL in your browser and press **Enter**:

   `http://<ip_address>/api/v2/ui`

2. Sign in using the administrator user name and password.

Result

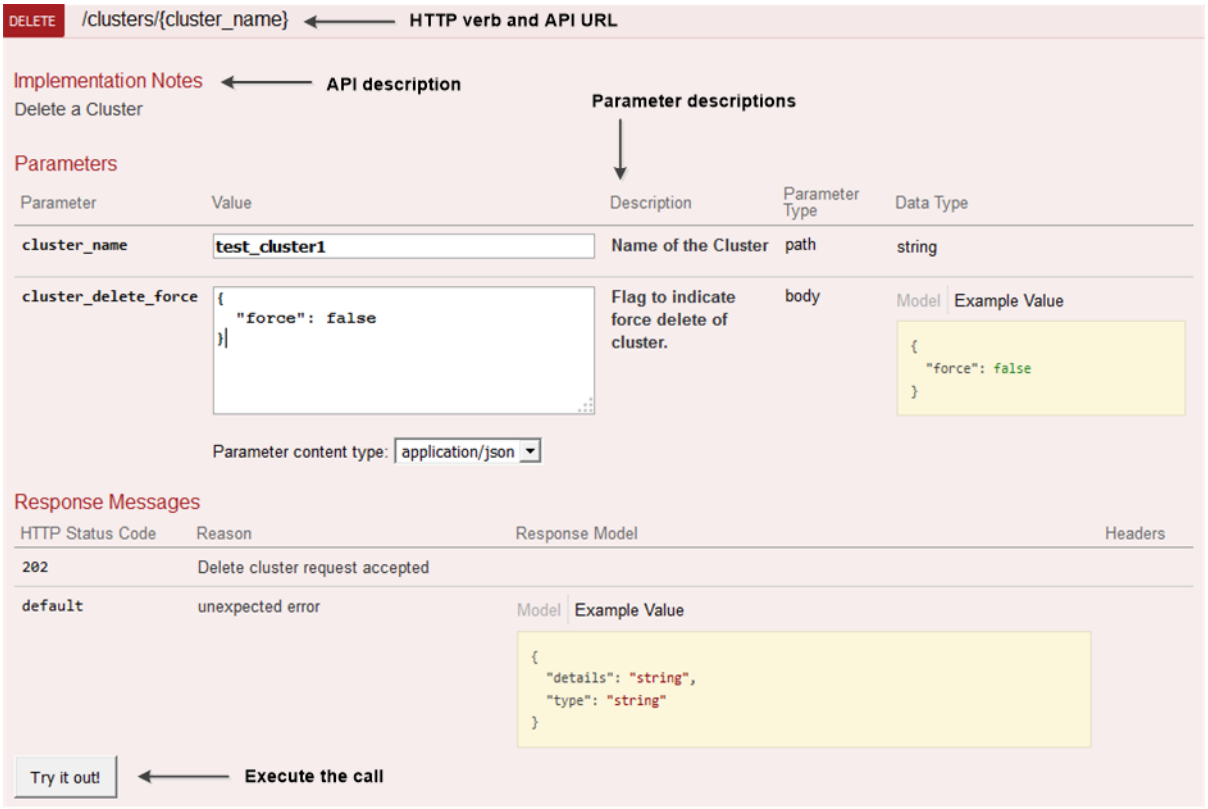
The main Swagger page is displayed with the calls organized by resource category.

Understanding and executing an API call

The details of all the API calls are included on the Swagger web page. All of the API calls are documented and displayed using a common format. By understanding a single API call, you can access and interpret the details of all the API calls.

Steps

1. Sign in to the Swagger page at the ONTAP Select Deploy virtual machine.
2. On the main Swagger page, click **Cluster**.
3. Click **DELETE** to display the details of the API call used to delete an ONTAP Select cluster.



4. Examine the entire page to understand the API call and what you must enter.

   You should note the HTTP verb and URL, required input parameters, the HTTP status codes used, and any implementation notes.

5. Type the cluster name, set the cluster force flag, and click **Try it out!**.

Related tasks

Accessing the Swagger API web page on page 10

Accessing and using the API with Python

In addition to accessing the ONTAP Select Deploy API through the product's native user interfaces and the Swagger web page, you can use a programming language or other command-line tool. By accessing the API programmatically, you gain a deeper understanding of the API and can automate the cluster deployment and administrative tasks.

While there are many options when choosing a programming language or tool to access the Deploy API, Python is a good choice. It is a popular and widely available scripting language. And in many cases, the Python code can be easily integrated into the automation processes typically used in most IT environments.

Prerequisites when using the code samples

You must prepare the environment before running the Python scripts.

Before you run the Python scripts, you must make sure the environment is configured properly:

- The latest applicable version of Python2 must be installed.
  The sample code has been tested using Python2. They should also be portable to Python3, but have not been tested for compatibility.
- The Requests and urllib3 libraries must be installed.
  You can use `pip` or another Python management tool as appropriate for your environment.
- The client workstation where the scripts run must have network access to the ONTAP Select Deploy virtual machine.

In addition, you must have the following information:

- IP address of the Deploy virtual machine
- User name and password of a Deploy administrator account

Flexible implementation and use

The Python code samples provide flexibility regarding how they can be used. The code can be implemented in several different environments as needed.

There are several design characteristics that provide implementation flexibility.

Object oriented

An object-oriented design is used to assure that the code can be easily updated and maintained. Also, by isolating the core code in several classes, there is a clear separation between the functionality needed to access a Deploy server and the client code, (including the user interface.

The following classes contain the code supporting API requests:

- Deploy API
  An instance of the class represents the Deploy API, and includes the client code needed to send HTTP requests to the server and process the responses.
- Messages
  This class includes support for the Python logger facility as well as string constants for common messages.

Note: You must rewrite the logger functions as needed when deploying the sample code outside the CLI environment.

- REST resources
  Every REST resource is represented as a different class, such as Host and Cluster.

Characteristics of the Python functions

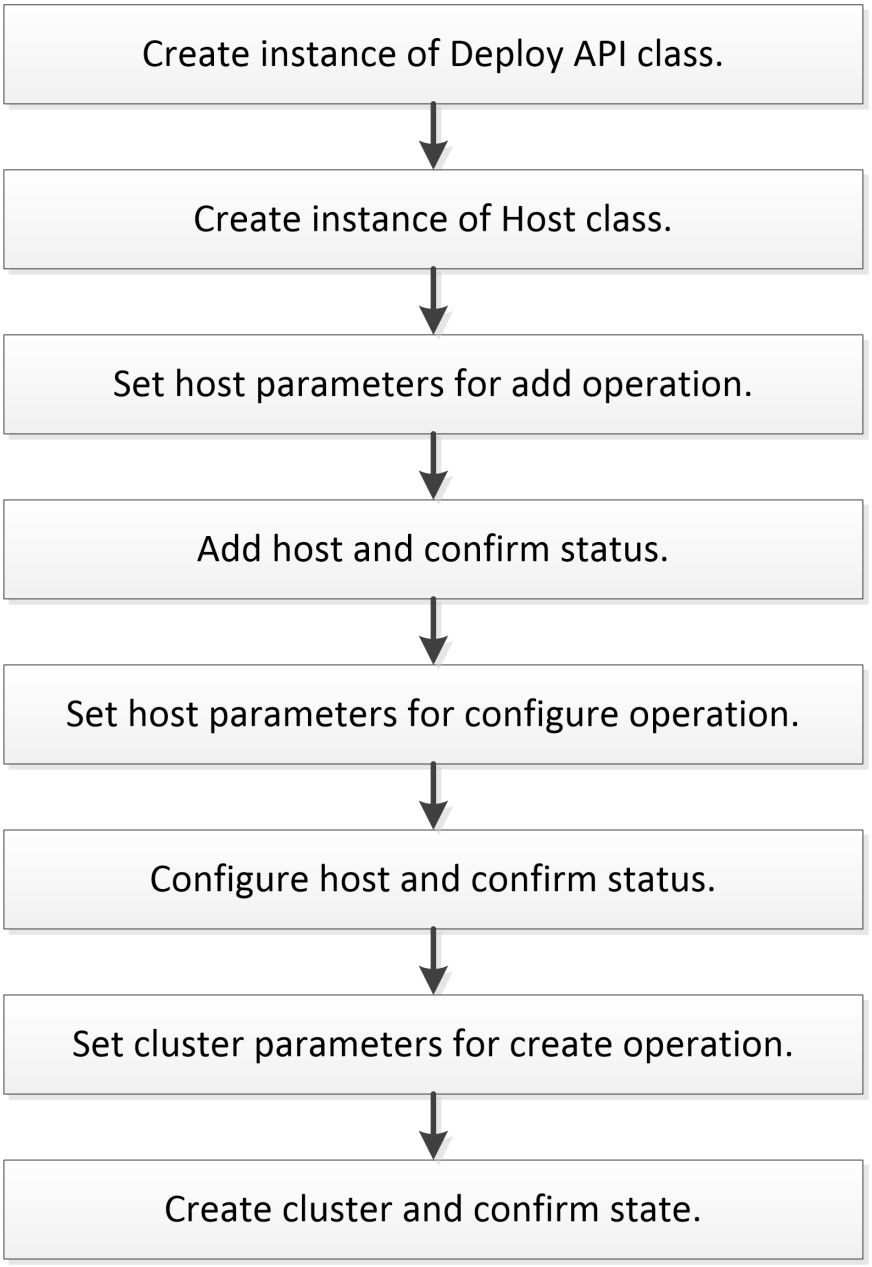
You must use one or more of the functions in each of the REST resource classes to perform an API request. Collectively, the functions in each class form a public or external interface to the resource. These functions have several common characteristics, including:

- Easy to identify
  The name of each function begins with `req_`.
- Use of the Messages class
  All of the functions use the Messages class, which implements the Python logging facility. Error messages are generated immediately after a failure is detected.

ONTAP Select cluster creation workflow

You can use the Deploy API to create an ONTAP Select cluster. You should used the Python code samples as described in the following workflow diagram.

Note: This workflow is designed to create a single-node cluster. To create a multi-node cluster, you must add and configure each of the additional hosts before creating the cluster.



Code samples

Several code samples are provided to help you to better understand the ONTAP Select Deploy REST API. Version 2 (v2) of the API is used in the samples.

Common support module

All of the Python scripts use a common set of classes contained in a single module.

```
#!/usr/bin/env python
##-----------------------------------------------------------------------
#
# Module: deploy_api.py
#
# Description: Contains the classes supporting client calls to the
# ONTAP Select Deploy administration utility REST API. Classes include:
```

```
#
#   Deploy_API
#     Supports communication with the Deploy virtual machine
#
#   Messages
#     Implements the Python logger facility to display messages at
#     different levels and includes common message text
#
# In addition, there is one class for each of the REST resources:
#
#   Host
#     Represents an ONTAP Select host
#
#   Cluster
#     Represents an ONTAP Select cluster
#
#
# NetApp publication: 215-12638_A0 (December 4, 2017)
# API version used: v2
#
#
# (C) Copyright 2017 NetApp, Inc.
#
# This sample code is provided AS IS, with no support or warranties of
# any kind, including but not limited for warranties of merchantability
# or fitness of any kind, expressed or implied. Permission to use,
# reproduce, modify and create derivatives of the sample code is granted
# solely for the purpose of researching, designing, developing and
# testing a software application product for use with NetApp products,
# provided that the above copyright notice appears in all copies and
# that the software application product is distributed pursuant to terms
# no less restrictive than those set forth herein.
#
##------------------------------------------------------------------------

import json
import base64
import requests
import urllib3
import time
import sys
import logging


## --- Deploy API  ------------------------------------------------------

class Deploy_API:
    """ Deploy API class
        Used by the REST resource classes to communicate with the
        Deploy virtual machine
    """

    def __init__(self, d_ip, d_user, d_pwd, log_level):
        """ Constructor """

        # Copy input parameters to instance variables
        self.deploy_ip   = d_ip
        self.deploy_user = d_user
        self.deploy_pwd  = d_pwd

        # Core instance variables supporting an HTTP session
        self.sess1       = requests.Session()
        self.url         = ""
        self.headers     = {}
        self.basic_auth  = "Basic "
        self.basic_auth  += base64.b64encode(bytes(self.deploy_user +
            ":" + self.deploy_pwd))

        # HTTP status code from most recent request
        self.http_code = 0
```

```
        # Suppress SSL unsigned certificate warning

urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)

        # Messages class
        self.messages = Messages(self, log_level)
        self.log      = self.messages.logger

        # Global termination flag
        self.exit = False

    def req_create_url(self, post_text):
        """ Create URL to be used in an HTTP request """

        self.url = ""
        self.url = "https://" + self.deploy_ip + "/api/v2"
        self.url += post_text

        self.log.debug(Messages.MSG_HTTP_URL + self.url)

        return

    def req_create_headers(self):
        """ Create common headers for an HTTP request """

        self.headers.clear
        self.headers['Host'] = self.deploy_ip
        self.headers['Accept'] = "application/json"
        self.headers['Accept-Language'] = "en-US,en;q=0.5"
        self.headers['Accept-Encoding'] = "gzip, deflate, br"
        self.headers['Content-Type']  = "application/json"
        self.headers['Authorization']  = self.basic_auth
        self.headers['Connection']  = "keep-alive"

        return

    def req_add_header(self, key, value):
        """ Add an HTTP header """

        self.headers[key] = value

        return

    def req_shutdown(self):
        """ Call function to close the session and clean up """

        self._shutdown()

        return

    def _shutdown(self):
        """ Close the session and clean up """

        self.exit = True
        self.sess1.close()

        return


## --- Messages ------------------------------------------------------

class Messages:
    """ Messages class
        Used by all the other classes to display or log messages
    """

    MSG_CONNECTION_FAILED  = "Error connecting to the Deploy API server"
    MSG_AUTH_DEPLOY_FAILED = "Authentication to the Deploy API server
failed"
    MSG_AUTH_HOST_FAILED   = "Authentication of the host failed"
    MSG_HTTP_URL           = "URL = "
```

```
    MSG_HTTP_CODE          = "HTTP status code: "
    MSG_HTTP_CODE_BAD      = "Unexpected HTTP status code: "
    MSG_HOST_DUP           = "Host already exists"
    MSG_HOST_STATUS        = "Host status: "
    MSG_HOST_STATUS_BAD    = "Unexpected host status: "
    MSG_HOST_TYPE          = "Host hypervisor type: "
    MSG_HOST_NOT_FOUND     = "Host not found"
    MSG_CLUSTER_DUP        = "Cluster already exists"
    MSG_CLUSTER_STATE      = "Cluster state: "
    MSG_CLUSTER_NOT_FOUND  = "Cluster not found"
    MSG_CLUSTER_STATE_BAD  = "Unexpected cluster state: "
    MSG_REQ_TRY            = "Attempting Deploy API request: "
    MSG_REQ_FAILED         = "Deploy API request failed: "
    MSG_POLL_TIMES         = "Polling - Times checked = "
    MSG_POLL_LINE          = "------------------------------"

    def __init__(self, api, log_level):
        """ Constructor """

        # Save instance of Deploy API
        self.das = api

        # Set logging level
        if (log_level == "debug"):
            self.level = logging.DEBUG
        elif (log_level == "info"):
            self.level = logging.INFO
        elif (log_level == "warning"):
            self.level = logging.WARNING
        elif (log_level == "error"):
            self.level = logging.ERROR
        elif (log_level == "critical"):
            self.level = logging.CRITICAL
        else:
            self.level = logging.DEBUG

        # Create root logger to display messages at the CLI console
        self.logger = logging.getLogger()
        self.logger.setLevel(self.level)
        self.ch = logging.StreamHandler(sys.stdout)
        self.ch.setLevel(self.level)
        self.formatter = logging.Formatter('%(levelname)s - %(message)s')
        self.ch.setFormatter(self.formatter)
        self.logger.addHandler(self.ch)

        return


## --- Host ----------------------------------------------------------

class Host:
    """ Host class """

    # Hypervisors
    HYPER_TYPE_ESX         = "esx"
    HYPER_TYPE_KVM         = "kvm"
    HYPER_TYPE_UNKNOWN     = "unknown"

    # Client request types
    REQ_TYPE_ADD           = "Host add"
    REQ_TYPE_STATUS        = "Host get status"
    REQ_TYPE_REMOVE        = "Host remove"
    REQ_TYPE_CONFIG        = "Host configure"

    # Status values for host authentication
    STAT_AUTH_NOT_FOUND    = "not found"
    STAT_AUTH_NONE         = 'not_authenticated'
    STAT_AUTH_IN_PROGRESS  = 'authentication_in_progress'
    STAT_AUTH_FAILED       = 'authentication_failed'
    STAT_AUTH_AUTHENTICATED = 'authenticated'
```

```
    # Status values for host configuration
    STAT_CONF_NONE          = 'not_configured'
    STAT_CONF_IN_PROGRESS   = 'configuration_in_progress'
    STAT_CONF_FAILED        = 'configuration_failed'
    STAT_CONF_CONFIGURED    = 'configured'

    def __init__(self, api):
        """ Constructor """

        # Deploy API server
        self.das = api

        # Initialize all parameters used in REST API calls
        self.parms = {}
        self.parms['type'] = Host.HYPER_TYPE_UNKNOWN
        self.parms['status'] = Host.STAT_AUTH_NOT_FOUND
        self.parms['id'] = ""
        self.parms['user'] = ""
        self.parms['pwd'] = ""
        self.parms['vcenter'] = ""
        self.parms['loc'] = ""
        self.parms['eval'] = True
        self.parms['instance'] = ""
        self.parms['net_mgmt_name'] = ""
        self.parms['net_mgmt_vlan'] = 0
        self.parms['net_data_name'] = ""
        self.parms['net_data_vlan'] = 0
        self.parms['net_int_name']  = ""
        self.parms['net_int_vlan']  = 0
        self.parms['mgmt_ip'] = ""
        self.parms['s_pool'] = []

        # Holds JSON used with some of the API calls
        self.data = {}

        # Resource type for URL (everything after IP and port)
        self.resource = ""

        # Logger instance from Messages class in Deploy API
        self.log = api.messages.logger

    def req_set_parms(self, p1):
        """ Called by external clients

            Accept dictionary of zero to N host parameters and
            update the host parms dictionary. Each key must already
            exist in the host parms dictionary, else return the key
            causing the error.

            Return "":  request successful
            Return key: request failed
        """

        for key, value in p1.iteritems():
            if key in self.parms:
                self.parms[key] = value
            else:
                return key

        return ""

    def req_get_parm(self, key):
        """ Called by external clients

            Return value of key in parms dictionary. If key not found
            in parms, return None.

            Return value: request successful
            Return None:  request failed
        """
```

```
        if key in self.parms:
            return self.parms[key]
        else:
            return None

    def req_add(self):
        """ Called by external clients

            Send request to Deploy server to add a host.

            Return True:  request successful
            Return False: request failed
        """

        # Check if the host already exists at the Deploy API server
        if (self._get_status()):
            if (self.parms['status'] != Host.STAT_AUTH_NOT_FOUND):
                self.log.warning(Messages.MSG_HOST_DUP)
        else:
            self.log.error(Messages.MSG_REQ_FAILED + Host.REQ_TYPE_ADD)
            return False

        # Prepare to send HTTP request
        self.resource = "/hosts/" + self.parms['id']
        self.das.req_create_url(self.resource)
        self.das.req_create_headers()
        self.data.clear()
        self.data['password'] = self.parms['pwd']
        self.data['username'] = self.parms['user']
        if (self.parms['vcenter'] != ""):
            self.data['vcenter'] = self.parms['vcenter']

        self.log.debug(Messages.MSG_REQ_TRY + Host.REQ_TYPE_ADD)

        try:

            resp1 = self.das.sess1.request('put', self.das.url,
                headers=self.das.headers, data=json.dumps(self.data),
                allow_redirects=True, verify=False)

        except requests.exceptions.ConnectionError:

            self.log.critical(Messages.MSG_CONNECTION_FAILED)
            self.parms['status'] = Host.STAT_AUTH_NOT_FOUND
            self.das.req_shutdown()
            sys.exit(1)

        # Retrieve HTTP status code
        self.das.http_code = resp1.status_code
        self.log.debug(Messages.MSG_HTTP_CODE + str(self.das.http_code))

        if self.das.http_code == 202:
            self.parms['status'] = json.loads(resp1.text)['status']
            self.log.debug(Messages.MSG_HOST_STATUS  +
self.parms['status'])
            self._poll_add()
        elif self.das.http_code == 401:
            self.log.error(Messages.MSG_AUTH_DEPLOY_FAILED)
            self.log.error(Messages.MSG_REQ_FAILED + Host.REQ_TYPE_ADD)
            self.parms['status'] = Host.STAT_AUTH_NOT_FOUND
            self.das.exit = True
        else:
            self.log.error(Messages.MSG_HTTP_CODE_BAD +
                str(self.das.http_code))
            self.log.error(Messages.MSG_REQ_FAILED + Host.REQ_TYPE_ADD)
            self.parms['status'] = Host.STAT_AUTH_NOT_FOUND
            self.das.exit = True

        return not self.das.exit

    def req_get_status(self):
```

```
        """ Called by external clients

            Send request to Deploy server to get current host status.

            Return True:  request successful
            Return False: request failed
        """

        # Call internal function to get status
        return self._get_status()

    def _get_status(self):
        """ Internal utility function. Send request to Deploy server
            to get current host status.

            Return True:  request successful
            Return False: request failed
        """

        # Prepare to send HTTP request
        self.resource = "/hosts/" + self.parms['id']
        self.das.req_create_url(self.resource)
        self.das.req_create_headers()
        self.log.debug(Messages.MSG_REQ_TRY + Host.REQ_TYPE_STATUS)

        try:

            resp2 = self.das.sess1.request('get', self.das.url,
                    headers=self.das.headers, allow_redirects=True,
                    verify=False)

        except requests.exceptions.ConnectionError:

            self.log.critical(Messages.MSG_CONNECTION_FAILED)
            self.parms['status'] = Host.STAT_AUTH_NOT_FOUND
            self.das.req_shutdown()
            sys.exit(1)

        # Retrieve HTTP status code
        self.das.http_code = resp2.status_code
        self.log.debug(Messages.MSG_HTTP_CODE + str(self.das.http_code))

        if self.das.http_code == 200:
            # Get host status
            self.parms['status'] = json.loads(resp2.text)['status']
            # Get hypervisor type (esx, kvm)
            self.parms['type'] = Host.HYPER_TYPE_UNKNOWN
            if ('physical_config' in json.loads(resp2.text)):
                if ('hypervisor_config' in json.loads(resp2.text)
['physical_config']):
                    if ('esx' in json.loads(resp2.text)
['physical_config']['hypervisor_config']):
                        self.parms['type'] = Host.HYPER_TYPE_ESX
                    elif ('kvm' in json.loads(resp2.text)
['physical_config']['hypervisor_config']):
                        self.parms['type'] = Host.HYPER_TYPE_KVM
            self.log.debug(Messages.MSG_HOST_TYPE + self.parms['type'])
        elif self.das.http_code == 404:
            self.parms['status'] = Host.STAT_AUTH_NOT_FOUND
        elif self.das.http_code == 401:
            self.log.error(Messages.MSG_AUTH_DEPLOY_FAILED)
            self.log.error(Messages.MSG_REQ_FAILED +
                Host.REQ_TYPE_STATUS)
            self.parms['status'] = Host.STAT_AUTH_NOT_FOUND
            self.das.exit = True
        else:
            self.log.error(Messages.MSG_HTTP_CODE_BAD +
                 str(self.das.http_code))
            self.log.error(Messages.MSG_REQ_FAILED +
                Host.REQ_TYPE_STATUS)
            self.parms['status'] = Host.STAT_AUTH_NOT_FOUND
```

```
            self.das.exit = True

        return not self.das.exit

    def req_remove(self):
        """ Called by external clients

            Send request to Deploy server to remove a host.

            Return True:  request successful
            Return False: request failed
        """

        # Exit if the host does not exist
        if (self._get_status()):
            if (self.parms['status'] == Host.STAT_AUTH_NOT_FOUND):
                self.log.warning(Messages.MSG_HOST_NOT_FOUND)
                return False
        else:
            return False

        # Prepare to send HTTP request
        self.resource = "/hosts/" + self.parms['id']
        self.das.req_create_url(self.resource)
        self.das.req_create_headers()
        self.log.debug(Messages.MSG_REQ_TRY + Host.REQ_TYPE_REMOVE)

        try:

            resp3 = self.das.sess1.request('delete', self.das.url,
                    headers=self.das.headers, allow_redirects=True,
                    verify=False)

        except requests.exceptions.ConnectionError:

            self.log.critical(Messages.MSG_CONNECTION_FAILED)
            self.parms['status'] = Host.STAT_AUTH_NOT_FOUND
            self.das.req_shutdown()
            sys.exit(1)

        # Retrieve HTTP status code
        self.das.http_code = resp3.status_code
        self.log.debug(Messages.MSG_HTTP_CODE + str(self.das.http_code))

        if self.das.http_code == 204:
            self._poll_remove()
        elif self.das.http_code == 404:
            self.parms['status'] = Host.STAT_AUTH_NOT_FOUND
        elif self.das.http_code == 401:
            self.log.error(Messages.MSG_AUTH_DEPLOY_FAILED)
            self.log.error(Messages.MSG_REQ_FAILED +
                Host.REQ_TYPE_REMOVE)
            self.parms['status'] = Host.STAT_AUTH_NOT_FOUND
            self.das.exit = True
        else:
            self.log.error(Messages.MSG_HTTP_CODE_BAD +
                 str(self.das.http_code))
            self.log.error(Messages.MSG_REQ_FAILED +
                Host.REQ_TYPE_REMOVE)
            self.parms['status'] = Host.STAT_AUTH_NOT_FOUND
            self.das.exit = True

        return not self.das.exit

    def req_configure(self):
        """ Called by external clients

            Send request to Deploy server to configure a host.

            Return True:  request successful
            Return False: request failed
```

```
        """

        # Confirm that the host exists and is authenticated
        if (self._get_status()):
            if (self.parms['status'] != Host.STAT_AUTH_AUTHENTICATED):
                self.log.warning("Host is not authenticated")
                return False
        else:
            self.log.error(Messages.MSG_REQ_FAILED +
Host.REQ_TYPE_CONFIG)
            return False

        # Prepare to send HTTP request
        self.resource = "/hosts/" + self.parms['id'] + "/configuration"
        self.das.req_create_url(self.resource)
        self.das.req_create_headers()
        self.data.clear()
        self.data['data_network'] = {'name': self.parms['net_data_name']}
        self.data['eval'] = self.parms['eval']
        self.data['instance_type'] = self.parms['instance']
        self.data['location'] = self.parms['loc']
        self.data['mgmt_network'] = {'name': self.parms['net_mgmt_name']}
        self.data['storage_pool'] = self.parms['s_pool']

        self.log.debug(Messages.MSG_REQ_TRY + Host.REQ_TYPE_CONFIG)

        try:

            resp4 = self.das.sess1.request('put', self.das.url,
                headers=self.das.headers, data=json.dumps(self.data),
                allow_redirects=True, verify=False)

        except requests.exceptions.ConnectionError:

            self.log.critical(Messages.MSG_CONNECTION_FAILED)
            self.parms['status'] = Host.STAT_AUTH_NOT_FOUND
            self.das.req_shutdown()
            sys.exit(1)

        # Retrieve HTTP status code
        self.das.http_code = resp4.status_code
        self.log.debug(Messages.MSG_HTTP_CODE + str(self.das.http_code))

        if self.das.http_code == 202:
            self._poll_configure()
        elif self.das.http_code == 401:
            self.log.error(Messages.MSG_AUTH_DEPLOY_FAILED)
            self.log.error(Messages.MSG_REQ_FAILED +
Host.REQ_TYPE_CONFIG)
            self.parms['status'] = Host.STAT_AUTH_NOT_FOUND
            self.das.exit = True
        else:
            self.log.error(Messages.MSG_HTTP_CODE_BAD +
                str(self.das.http_code))
            self.log.error(Messages.MSG_REQ_FAILED +
Host.REQ_TYPE_CONFIG)
            self.parms['status'] = Host.STAT_AUTH_NOT_FOUND
            self.das.exit = True

        return not self.das.exit

    def _poll_add(self):
        """ Internal utility function. Poll status of host after
            add to confirm it is authenticated.

            Return True:  request successful
            Return False: request failed
        """

        ready   = False
        times   = 0
```

```
        seconds = 1

        while (not ready and not self.das.exit):

            times = times + 1

            self.log.debug(Messages.MSG_POLL_LINE)
            self.log.debug(Messages.MSG_POLL_TIMES + str(times))

            if (self._get_status()):

                self.log.debug(Messages.MSG_HOST_STATUS +
                    self.parms['status'])

                if (self.parms['status'] ==
Host.STAT_AUTH_AUTHENTICATED):
                    ready = True
                    break
                elif (self.parms['status'] ==
Host.STAT_AUTH_IN_PROGRESS):
                    time.sleep(seconds)
                elif (self.parms['status'] == Host.STAT_AUTH_FAILED):
                    self.log.debug(Messages.MSG_REQ_FAILED +
                         Host.REQ_TYPE_ADD)
                    self.log.error(Messages.MSG_AUTH_HOST_FAILED)
                    self.das.exit = True
                    break
                elif (self.parms['status'] == Host.STAT_AUTH_NOT_FOUND):
                    self.log.error(Messages.MSG_HOST_NOT_FOUND)
                    self.log.debug(Messages.MSG_REQ_FAILED +
                         Host.REQ_TYPE_ADD)
                    self.das.exit = True
                    break
                else:
                    self.log.error(Messages.MSG_HOST_STATUS_BAD +
                        self.parms['status'])
                    self.log.debug(Messages.MSG_REQ_FAILED +
                         Host.REQ_TYPE_ADD)
                    self.das.exit = True
                    break

            else:

                break

        return ready

    def _poll_remove(self):
        """ Internal utility function. Poll status of host to
            confirm it has been removed.

            Return True:  request successful
            Return False: request failed
        """

        ready   = False
        times   = 0
        seconds = 1

        while (not ready and not self.das.exit):

            times = times + 1

            self.log.debug(Messages.MSG_POLL_LINE)
            self.log.debug(Messages.MSG_POLL_TIMES + str(times))

            if (self._get_status()):

                self.log.debug(Messages.MSG_HOST_STATUS +
                    self.parms['status'])
```

```
                if (self.parms['status'] == Host.STAT_AUTH_NOT_FOUND):
                    ready = True
                    break
                else:
                    time.sleep(seconds)

            else:

                break

        return ready

    def _poll_configure(self):
        """ Internal utility function. Poll status of host after
            configure to confirm it is configured.

            Return True:  request successful
            Return False: request failed
        """

        ready   = False
        times   = 0
        seconds = 1

        while (not ready and not self.das.exit):

            times = times + 1

            self.log.debug(Messages.MSG_POLL_LINE)
            self.log.debug(Messages.MSG_POLL_TIMES + str(times))

            if (self._get_status()):

                self.log.debug(Messages.MSG_HOST_STATUS +
                    self.parms['status'])

                if (self.parms['status'] == Host.STAT_CONF_CONFIGURED):
                    ready = True
                    break
                elif (self.parms['status'] ==
Host.STAT_CONF_IN_PROGRESS):
                    time.sleep(seconds)
                elif (self.parms['status'] == Host.STAT_CONF_FAILED):
                    self.log.debug(Messages.MSG_REQ_FAILED +
                         Host.REQ_TYPE_CONFIG)
                    self.log.error(Messages.MSG_AUTH_HOST_FAILED)
                    self.das.exit = True
                    break
                elif (self.parms['status'] == STAT_AUTH_NOT_FOUND):
                    self.log.error(Messages.MSG_HOST_NOT_FOUND)
                    self.log.debug(Messages.MSG_REQ_FAILED +
                         Host.REQ_TYPE_CONFIG)
                    self.das.exit = True
                    break
                else:
                    self.log.error(Messages.MSG_HOST_STATUS_BAD +
                        self.parms['status'])
                    self.log.debug(Messages.MSG_REQ_FAILED +
                         Host.REQ_TYPE_CONFIG)
                    self.das.exit = True
                    break

            else:

                break

        return ready


## --- Cluster ----------------------------------------------------
```

```
class Cluster:
    """ Cluster class """

    # Client request types
    REQ_TYPE_CREATE         = "Cluster create"
    REQ_TYPE_STATE          = "Cluster state"

    # State values (partial list)
    STAT_CREATING           = 'creating'
    STAT_CREATE_FAILED      = 'create_failed'
    STAT_HOST_CONFIG_IN_PROGRESS = 'host_configuration_in_progress'
    STAT_DEPLOYING_NODES    = 'deploying_nodes'
    STAT_CREATING_DATA_DISKS = 'creating_data_disks'
    STAT_POST_DEPLOY_SETUP  = 'post_deploy_setup'
    STAT_ONLINE             = 'online'
    STAT_OFFLINE            = 'offline'
    STAT_NOT_FOUND          = "not found"

    def __init__(self, api):
        """ Constructor """

        # Deploy API server
        self.das = api

        # Initialize all parameters used in REST API calls
        self.parms = {}
        self.parms['name'] = ""
        self.parms['state'] = ""
        self.parms['admin_pwd'] = ""
        self.parms['mgmt_ip'] = ""
        self.parms['netmask'] = ""
        self.parms['gateway'] = ""
        self.parms['nodes'] = []

        # Holds list of cluster nodes used with some of the REST calls
        self.cnodes = []

        # Holds JSON used with some of the API calls
        self.data = {}

        # Resource type for URL (everything after IP and port)
        self.resource = ""

        # Cluster state from the most recent request
        self.parms['state'] = Cluster.STAT_NOT_FOUND

        # Logger instance from Messages class in Deploy API
        self.log = api.messages.logger

        return

    def req_set_parms(self, p1):
        """ Called by external clients

            Accept dictionary of zero to N cluster parameters and
            update the cluster parms dictionary. Each key must already
            exist in the cluster parms dictionary, else return the key
            causing the error.

            Return "":  request successful
            Return key: request failed
        """

        for key, value in p1.iteritems():
            if key in self.parms:
                self.parms[key] = value
            else:
                return key

        return ""
```

```
    def req_get_parm(self, key):
        """ Called by external clients

            Return value of key in parms dictionary. If key not found
            in parms, return None.

            Return value: request successful
            Return None:  request failed
        """

        if key in self.parms:
            return self.parms[key]
        else:
            return None

    def req_create(self):
        """ Called by external clients

            Send request to Deploy server to create a cluster.

            Return True:  request successful
            Return False: request failed
        """

        # Check if the cluster already exists at the Deploy API server
        if (self._get_state()):
            if (self.parms['state'] != Cluster.STAT_NOT_FOUND):
                self.log.warning(Messages.MSG_CLUSTER_DUP)
                return False
        else:
            self.log.error(Messages.MSG_REQ_FAILED +
Cluster.REQ_TYPE_CREATE)
            return False

        # Construct list of cluster nodes from host nodes
        self.cnodes = []
        for n in self.parms['nodes']:
            self.temp_node = {}
            self.temp_node['host'] = n.req_get_parm('id')
            self.temp_node['node_mgmt_ip'] = n.req_get_parm('mgmt_ip')
            self.cnodes.append(self.temp_node)

        # Prepare to send HTTP request
        self.resource = "/clusters"
        self.das.req_create_url(self.resource)
        self.das.req_create_headers()
        self.data.clear()
        self.data['name'] = self.parms['name']
        self.data['admin_password'] = self.parms['admin_pwd']
        self.data['cluster_mgmt_ip'] = self.parms['mgmt_ip']
        self.data['netmask'] = self.parms['netmask']
        self.data['gateway'] = self.parms['gateway']
        self.data['nodes'] = self.cnodes

        self.log.debug(Messages.MSG_REQ_TRY + Cluster.REQ_TYPE_CREATE)

        try:

            resp1 = self.das.sess1.request('post', self.das.url,
                headers=self.das.headers, data=json.dumps(self.data),
                allow_redirects=True, verify=False)

        except requests.exceptions.ConnectionError:

            self.log.critical(Messages.MSG_CONNECTION_FAILED)
            self.parms['state'] = Cluster.STAT_NOT_FOUND
            self.das.req_shutdown()
            sys.exit(1)

        # Retrieve HTTP status code
        self.das.http_code = resp1.status_code
```

```
        self.log.debug(Messages.MSG_HTTP_CODE + str(self.das.http_code))

        if self.das.http_code == 202:
            self.parms['state'] = json.loads(resp1.text)['state']
            self.log.debug(Messages.MSG_CLUSTER_STATE  +
self.parms['state'])
            self._poll_create()
        elif self.das.http_code == 401:
            self.log.error(Messages.MSG_AUTH_DEPLOY_FAILED)
            self.log.error(Messages.MSG_REQ_FAILED +
Cluster.REQ_TYPE_CREATE)
            self.parms['state'] = Cluster.STAT_NOT_FOUND
            self.das.exit = True
        else:
            self.log.error(Messages.MSG_HTTP_CODE_BAD +
                str(self.das.http_code))
            self.log.error(Messages.MSG_REQ_FAILED +
Cluster.REQ_TYPE_CREATE)
            self.parms['state'] = Cluster.STAT_NOT_FOUND
            self.das.exit = True

        return not self.das.exit

    def req_get_state(self):
        """ Called by external clients

            Send request to Deploy server to get current cluster state

            Return True:  request successful
            Return False: request failed
        """

        # Call internal function to get state
        return self._get_state()

    def _get_state(self):
        """ Internal utility function. Send request to Deploy server
            to get current cluster state.

            Return True:  request successful
            Return False: request failed
        """

        # Prepare to send HTTP request
        self.resource = "/clusters/" + self.parms['name']
        self.das.req_create_url(self.resource)
        self.das.req_create_headers()
        self.log.debug(Messages.MSG_REQ_TRY + Cluster.REQ_TYPE_STATE)

        try:

            resp2 = self.das.sess1.request('get', self.das.url,
                    headers=self.das.headers, allow_redirects=True,
                    verify=False)

        except requests.exceptions.ConnectionError:

            self.log.critical(Messages.MSG_CONNECTION_FAILED)
            self.parms['state'] = Cluster.STAT_NOT_FOUND
            self.das.req_shutdown()
            sys.exit(1)

        # Retrieve HTTP status code
        self.das.http_code = resp2.status_code
        self.log.debug(Messages.MSG_HTTP_CODE + str(self.das.http_code))

        if self.das.http_code == 200:
            self.parms['state'] = json.loads(resp2.text)['state']
        elif self.das.http_code == 404:
            self.parms['state'] = Cluster.STAT_NOT_FOUND
        elif self.das.http_code == 401:
```

```
            self.log.error(Messages.MSG_AUTH_DEPLOY_FAILED)
            self.log.error(Messages.MSG_REQ_FAILED +
                Cluster.REQ_TYPE_STATE)
            self.parms['state'] = Cluster.STAT_NOT_FOUND
            self.das.exit = True
        else:
            self.log.error(Messages.MSG_HTTP_CODE_BAD +
                 str(self.das.http_code))
            self.log.error(Messages.MSG_REQ_FAILED +
                Cluster.REQ_TYPE_STATE)
            self.parms['state'] = Cluster.STAT_NOT_FOUND
            self.das.exit = True

        return not self.das.exit

    def _poll_create(self):
        """ Internal utility function. Poll status of cluster after
            create to confirm it is online.

            Return True:  request successful
            Return False: request failed
        """

        ready   = False
        times   = 0
        seconds = 3

        while (not ready and not self.das.exit):

            times = times + 1

            self.log.debug(Messages.MSG_POLL_LINE)
            self.log.debug(Messages.MSG_POLL_TIMES + str(times))

            if (self._get_state()):

                self.log.debug(Messages.MSG_CLUSTER_STATE +
                    self.parms['state'])

                if (self.parms['state'] == Cluster.STAT_ONLINE):
                    ready = True
                    break
                elif (self.parms['state'] == Cluster.STAT_CREATE_FAILED):
                    self.log.debug(Messages.MSG_REQ_FAILED +
                         Cluster.REQ_TYPE_CREATE)
                    self.das.exit = True
                    break
                elif (self.parms['state'] == Cluster.STAT_NOT_FOUND):
                    self.log.error(Messages.MSG_CLUSTER_NOT_FOUND)
                    self.log.debug(Messages.MSG_REQ_FAILED +
                         Cluster.REQ_TYPE_CREATE)
                    self.das.exit = True
                    break
                else:
                     time.sleep(seconds)

            else:

                break

        return ready
```

Create a cluster

You can use the following CLI script to create a single-node cluster using an evaluation license, based on parameters defined within the script. The host is authenticated to the vCenter server.

```
#!/usr/bin/env python
##-----------------------------------------------------------------------
```

```
#
# Module: depl_create_cluster.py
#
# Description: CLI script to create a single-node ONTAP Select cluster
# with an evaluation license. Major steps include:
#
#   * Add host (authenticating to vCenter)
#   * Configure host
#   * Create cluster
#
# Usage example:
#
#   python depl_create_cluster.py
#
# Parameters:
#
#   All parameters are defined within this script. Additional input parms
#   are not accepted at the CLI.
#
# NetApp publication: 215-12638_A0 (December 4, 2017)
# API version used: v2
#
#
# (C) Copyright 2017 NetApp, Inc.
#
# This sample code is provided AS IS, with no support or warranties of
# any kind, including but not limited for warranties of merchantability
# or fitness of any kind, expressed or implied. Permission to use,
# reproduce, modify and create derivatives of the sample code is granted
# solely for the purpose of researching, designing, developing and
# testing a software application product for use with NetApp products,
# provided that the above copyright notice appears in all copies and
# that the software application product is distributed pursuant to terms
# no less restrictive than those set forth herein.
#
##------------------------------------------------------------------------

import sys
import deploy_api
import argparse
import json

##------------------------------------------------------------------------
##  Set variables used in API calls
##------------------------------------------------------------------------

# Deploy API server
depl_ip       = "10.202.18.17"
depl_user     = "admin"
depl_password = "deploy123"

# vCenter
vcenter_ip    = "10.202.18.16"
host_user     = "administrator@vsphere.local"
host_password = "netapp"

# Host ID
host_id       = "10.202.18.12"

# Logging level
conf_level    = "info"
#conf_level    = "debug"

# List of hosts (only use one in this script)
hosts = []

##------------------------------------------------------------------------
##  Create an instance of the Deploy API class
##------------------------------------------------------------------------

das = deploy_api.Deploy_API(depl_ip, depl_user, depl_password,
```

```
          conf_level)

##-------------------------------------------------------------------------
##  Create an instance of the Host class
##-------------------------------------------------------------------------

hosts.append( deploy_api.Host(das) )

##-------------------------------------------------------------------------
##  Prepare to add the host
##-------------------------------------------------------------------------

# Can optionally test each of the req_set_parms() calls
bad_key = hosts[0].req_set_parms( {'id': host_id} )
if (bad_key != ""):
    print("Function req_set_parms() failed with bad key: " + bad_key)
    das.req_shutdown()
    sys.exit(1)

# Complete host parameters for host add
hosts[0].req_set_parms( {'vcenter': vcenter_ip} )
hosts[0].req_set_parms( {'user': host_user, 'pwd': host_password} )

##-------------------------------------------------------------------------
##  Add the host to the Deploy server and display status
##-------------------------------------------------------------------------

if (hosts[0].req_add()):
    print("Host add request successfully completed")
else:
    print("Host add request failed")
    das.req_shutdown()
    sys.exit(1)

if (hosts[0].req_get_status()):
    print("Host status = " + hosts[0].req_get_parm("status"))
else:
    print("Get status request failed")
    das.req_shutdown()
    sys.exit(1)

##-------------------------------------------------------------------------
##  Prepare to configure the host
##-------------------------------------------------------------------------

hosts[0].req_set_parms( {'loc': 'HQ', 'eval': True} )
hosts[0].req_set_parms( {'instance': 'small'} )

hosts[0].req_set_parms( {'net_mgmt_name': 'sDOT_Network',
                     'net_data_name': 'sDOT_Network'} )

# Datastore minimum is 500GB
hosts[0].req_set_parms( {'s_pool': [{'capacity': 500, 'name':
'sDOT-01'}]} )

# Node mgmt IP needed for cluster create
hosts[0].req_set_parms( {'mgmt_ip': '10.202.18.21'} )

##-------------------------------------------------------------------------
##  Configure the host and display status
##-------------------------------------------------------------------------

if (hosts[0].req_configure()):
    print("Host configure request successfully completed")
else:
    print("Host configure request failed")
    das.req_shutdown()
    sys.exit(1)

if (hosts[0].req_get_status()):
    print("Host status = " + hosts[0].req_get_parm("status"))
```

```
else:
    print("Get status request failed")
    das.req_shutdown()
    sys.exit(1)

##--------------------------------------------------------------------
##  Create an instance of the Cluster class
##--------------------------------------------------------------------

cluster = deploy_api.Cluster(das)

##--------------------------------------------------------------------
##  Prepare to create the cluster
##--------------------------------------------------------------------

cluster.req_set_parms( {'name': 'mycluster'} )
cluster.req_set_parms( {'admin_pwd': 'mypwd123'} )
cluster.req_set_parms( {'mgmt_ip': '10.202.18.20'} )
cluster.req_set_parms( {'gateway': '10.202.18.1'} )
cluster.req_set_parms( {'netmask': '255.255.255.192'} )
cluster.req_set_parms( {'nodes': hosts} )

##--------------------------------------------------------------------
##  Create the cluster
##--------------------------------------------------------------------

if (cluster.req_create()):
    print("Cluster create request successfully completed")
else:
    print("Cluster create request failed")

if (cluster.req_get_state()):
    print("Cluster state = " + cluster.req_get_parm("state"))
else:
    print("Get state request failed")
    das.req_shutdown()
    sys.exit(1)

##--------------------------------------------------------------------
##  Clean up and exit
##--------------------------------------------------------------------

das.req_shutdown()
```

Display the status of a host

You can use the following script to display the current status of a host.

```
#!/usr/bin/env python
##--------------------------------------------------------------------
#
# Module: depl_host_status.py
#
# Description: CLI script to display the status of an ONTAP Select host
# based on input parameters that are accepted and parsed.
#
# Usage example:
#
#   python depl_host_status.py -i 10.63.66.22 -u admin -p password1
#          -d host01 -z debug
#
# Parameters:
#
#   -i  IP address of ONTAP Select Deploy virtual machine
#   -u  user name at the Deploy server
#   -p  password for Deploy user
#   -d  host id (IP address or domain name)
#   -z  log level (debug, info, warning, error, critical)
#
```

```
#
# NetApp publication: 215-12638_A0 (December 4, 2017)
# API version used: v2
#
#
# (C) Copyright 2017 NetApp, Inc.
#
# This sample code is provided AS IS, with no support or warranties of
# any kind, including but not limited for warranties of merchantability
# or fitness of any kind, expressed or implied. Permission to use,
# reproduce, modify and create derivatives of the sample code is granted
# solely for the purpose of researching, designing, developing and
# testing a software application product for use with NetApp products,
# provided that the above copyright notice appears in all copies and
# that the software application product is distributed pursuant to terms
# no less restrictive than those set forth herein.
#
##------------------------------------------------------------------------

import sys
import deploy_api
import argparse
import json

##------------------------------------------------------------------------
##  Parse and save the input parameters
##------------------------------------------------------------------------

parser = argparse.ArgumentParser(description='ONTAP Select Deploy v01',
                  add_help = True)

parser.add_argument("-i", "--deploy_ip", action="store", dest="d_ip",
       default=None, help='(req) management IP address of Deploy VM',
       required=True)
parser.add_argument("-u", "--deploy_user", action="store", dest="d_user",
       default=None, help='(req) Deploy user account', required=True)
parser.add_argument("-p", "--deploy_pwd", action="store", dest="d_pwd",
       default=None, help='(req) Deploy password', required=True)
parser.add_argument("-d", "--host_id", action="store", dest="h_id",
       default=None, help='(req) host id',
       required=True)
parser.add_argument("-z", "--level", action="store", dest="c_level",
       default=None, help='(opt) logging level',
       required=True)

args = parser.parse_args()

##------------------------------------------------------------------------
##  Prepare the environment
##------------------------------------------------------------------------

depl_ip       = args.d_ip
depl_user     = args.d_user
depl_password = args.d_pwd

host_id       = args.h_id

conf_level    = args.c_level

script_name   = sys.argv[0]

##------------------------------------------------------------------------
##  Create an instance of the Deploy API class
##------------------------------------------------------------------------

das = deploy_api.Deploy_API(depl_ip, depl_user, depl_password,
                  conf_level)

##------------------------------------------------------------------------
##  Create and prepare an instance of the Host class
##------------------------------------------------------------------------
```

```
host = deploy_api.Host(das)

host.req_set_parms( {'id': host_id} )

##------------------------------------------------------------------
##  Get the host status and display
##------------------------------------------------------------------

if (host.req_get_status()):
    print("Host status = " + host.req_get_parm('status'))
    print("Host type = " + host.req_get_parm('type'))

##------------------------------------------------------------------
##  Clean up and exit
##------------------------------------------------------------------

das.req_shutdown()
```

Copyright information

Trademark information

How to send comments about documentation and receive update notifications

You can help us to improve the quality of our documentation by sending us your feedback. You can receive automatic notification when production-level (GA/FCS) documentation is initially released or important changes are made to existing production-level documents.

If you have suggestions for improving this document, send us your comments by email.

doccomments@netapp.com

To help us direct your comments to the correct division, include in the subject line the product name, version, and operating system.

If you want to be notified automatically when production-level documentation is released or important changes are made to existing production-level documents, follow Twitter account @NetAppDoc.

You can also contact us in the following ways:

- NetApp, Inc., 495 East Java Drive, Sunnyvale, CA 94089 U.S.
- Telephone: +1 (408) 822-6000
- Fax: +1 (408) 822-4501
- Support telephone: +1 (888) 463-8277

Index