



Script to create a cluster

ONTAP Select

NetApp

November 21, 2019

This PDF was generated from https://docs.netapp.com/us-en/ontap-select/reference_api_script_cc.html on December 04, 2020. Always check docs.netapp.com for the latest.

Table of Contents

Script to create a cluster 1

Script to create a cluster

You can use the following script to create a cluster based on parameters defined within the script and a JSON input file.

```
1 #!/usr/bin/env python
2 ##-----
3 #
4 # File: cluster.py
5 #
6 # (C) Copyright 2019 NetApp, Inc.
7 #
8 # This sample code is provided AS IS, with no support or warranties of
9 # any kind, including but not limited for warranties of merchantability
10 # or fitness of any kind, expressed or implied. Permission to use,
11 # reproduce, modify and create derivatives of the sample code is granted
12 # solely for the purpose of researching, designing, developing and
13 # testing a software application product for use with NetApp products,
14 # provided that the above copyright notice appears in all copies and
15 # that the software application product is distributed pursuant to terms
16 # no less restrictive than those set forth herein.
17 #
18 ##-----
19
20 import traceback
21 import argparse
22 import json
23 import logging
24
25 from deploy_requests import DeployRequests
26
27
28 def add_vcenter_credentials(deploy, config):
29     """ Add credentials for the vcenter if present in the config """
30     log_debug_trace()
31
32     vcenter = config.get('vcenter', None)
33     if vcenter and not deploy.resource_exists('/security/credentials',
34                                             'hostname', vcenter['hostname']):
35         log_info("Registering vcenter {} credentials".format(vcenter['hostname']))
36         data = {k: vcenter[k] for k in ['hostname', 'username', 'password']}
37         data['type'] = "vcenter"
38         deploy.post('/security/credentials', data)
39
40
41 def add_standalone_host_credentials(deploy, config):
```

```

42     """ Add credentials for standalone hosts if present in the config.
43         Does nothing if the host credential already exists on the Deploy.
44     """
45     log_debug_trace()
46
47     hosts = config.get('hosts', [])
48     for host in hosts:
49         # The presense of the 'password' will be used only for standalone hosts.
50         # If this host is managed by a vcenter, it should not have a host 'password'
51         in the json.
52         if 'password' in host and not deploy.resource_exists('/security/credentials',
53                                                                 'hostname', host[
54                                                                 'name']):
55             log_info("Registering host {} credentials".format(host['name']))
56             data = {'hostname': host['name'], 'type': 'host',
57                     'username': host['username'], 'password': host['password']}
58             deploy.post('/security/credentials', data)
59
60 def register_unkown_hosts(deploy, config):
61     """ Registers all hosts with the deploy server.
62         The host details are read from the cluster config json file.
63
64         This method will skip any hosts that are already registered.
65         This method will exit the script if no hosts are found in the config.
66     """
67     log_debug_trace()
68
69     data = {"hosts": []}
70     if 'hosts' not in config or not config['hosts']:
71         log_and_exit("The cluster config requires at least 1 entry in the 'hosts'
72 list got {}".format(config))
73
74     missing_host_cnt = 0
75     for host in config['hosts']:
76         if not deploy.resource_exists('/hosts', 'name', host['name']):
77             missing_host_cnt += 1
78             host_config = {"name": host['name'], "hypervisor_type": host['type']}
79             if 'mgmt_server' in host:
80                 host_config["management_server"] = host['mgmt_server']
81                 log_info(
82                     "Registering from vcenter {mgmt_server}".format(**host))
83
84             if 'password' in host and 'user' in host:
85                 host_config['credential'] = {
86                     "password": host['password'], "username": host['user']}
87
88             log_info("Registering {type} host {name}".format(**host))

```

```

87         data["hosts"].append(host_config)
88
89     # only post /hosts if some missing hosts were found
90     if missing_host_cnt:
91         deploy.post('/hosts', data, wait_for_job=True)
92
93
94 def add_cluster_attributes(deploy, config):
95     ''' POST a new cluster with all needed attribute values.
96         Returns the cluster_id of the new config
97     '''
98     log_debug_trace()
99
100    cluster_config = config['cluster']
101    cluster_id = deploy.find_resource('/clusters', 'name', cluster_config['name'])
102
103    if not cluster_id:
104        log_info("Creating cluster config named {name}".format(**cluster_config))
105
106        # Filter to only the valid attributes, ignores anything else in the json
107        data = {k: cluster_config[k] for k in [
108            'name', 'ip', 'gateway', 'netmask', 'ontap_image_version', 'dns_info',
109            'ntp_servers']}
110
111        num_nodes = len(config['nodes'])
112
113        log_info("Cluster properties: {}".format(data))
114
115        resp = deploy.post('/v3/clusters?node_count={}'.format(num_nodes), data)
116        cluster_id = resp.headers.get('Location').split('/')[-1]
117
118    return cluster_id
119
120 def get_node_ids(deploy, cluster_id):
121     ''' Get the the ids of the nodes in a cluster. Returns a list of node_ids.'''
122     log_debug_trace()
123
124     response = deploy.get('/clusters/{}/nodes'.format(cluster_id))
125     node_ids = [node['id'] for node in response.json().get('records')]
126     return node_ids
127
128
129 def add_node_attributes(deploy, cluster_id, node_id, node):
130     ''' Set all the needed properties on a node '''
131     log_debug_trace()
132
133     log_info("Adding node '{}' properties".format(node_id))

```

```

134
135     data = {k: node[k] for k in ['ip', 'serial_number', 'instance_type',
136                                'is_storage_efficiency_enabled'] if k in node}
137     # Optional: Set a serial_number
138     if 'license' in node:
139         data['license'] = {'id': node['license']}
140
141     # Assign the host
142     host_id = deploy.find_resource('/hosts', 'name', node['host_name'])
143     if not host_id:
144         log_and_exit("Host names must match in the 'hosts' array, and the
nodes.host_name property")
145
146     data['host'] = {'id': host_id}
147
148     # Set the correct raid_type
149     is_hw_raid = not node['storage'].get('disks') # The presence of a list of disks
indicates sw_raid
150     data['passthrough_disks'] = not is_hw_raid
151
152     # Optionally set a custom node name
153     if 'name' in node:
154         data['name'] = node['name']
155
156     log_info("Node properties: {}".format(data))
157     deploy.patch('/clusters/{}/nodes/{}'.format(cluster_id, node_id), data)
158
159
160 def add_node_networks(deploy, cluster_id, node_id, node):
161     ''' Set the network information for a node '''
162     log_debug_trace()
163
164     log_info("Adding node '{}' network properties".format(node_id))
165
166     num_nodes = deploy.get_num_records('/clusters/{}/nodes'.format(cluster_id))
167
168     for network in node['networks']:
169
170         # single node clusters do not use the 'internal' network
171         if num_nodes == 1 and network['purpose'] == 'internal':
172             continue
173
174         # Deduce the network id given the purpose for each entry
175         network_id = deploy.find_resource('/clusters/{}/nodes/{}/networks'.format
(cluster_id, node_id),
176                                         'purpose', network['purpose'])
177         data = {"name": network['name']}
178         if 'vlan' in network and network['vlan']:

```

```

179         data['vlan_id'] = network['vlan']
180
181         deploy.patch('/clusters/{}/nodes/{}/networks/{}'.format(cluster_id, node_id,
network_id), data)
182
183
184 def add_node_storage(deploy, cluster_id, node_id, node):
185     ''' Set all the storage information on a node '''
186     log_debug_trace()
187
188     log_info("Adding node '{}' storage properties".format(node_id))
189     log_info("Node storage: {}".format(node['storage']['pools']))
190
191     data = {'pool_array': node['storage']['pools']} # use all the json properties
192     deploy.post(
193         '/clusters/{}/nodes/{}/storage/pools'.format(cluster_id, node_id), data)
194
195     if 'disks' in node['storage'] and node['storage']['disks']:
196         data = {'disks': node['storage']['disks']}
197         deploy.post(
198             '/clusters/{}/nodes/{}/storage/disks'.format(cluster_id, node_id), data)
199
200
201 def create_cluster_config(deploy, config):
202     ''' Construct a cluster config in the deploy server using the input json data '''
203     log_debug_trace()
204
205     cluster_id = add_cluster_attributes(deploy, config)
206
207     node_ids = get_node_ids(deploy, cluster_id)
208     node_configs = config['nodes']
209
210     for node_id, node_config in zip(node_ids, node_configs):
211         add_node_attributes(deploy, cluster_id, node_id, node_config)
212         add_node_networks(deploy, cluster_id, node_id, node_config)
213         add_node_storage(deploy, cluster_id, node_id, node_config)
214
215     return cluster_id
216
217
218 def deploy_cluster(deploy, cluster_id, config):
219     ''' Deploy the cluster config to create the ONTAP Select VMs. '''
220     log_debug_trace()
221     log_info("Deploying cluster: {}".format(cluster_id))
222
223     data = {'ontap_credential': {'password': config['cluster']['
'ontap_admin_password']}}
224     deploy.post('/clusters/{}/deploy?inhibit_rollback=true'.format(cluster_id),

```

```

225         data, wait_for_job=True)
226
227
228 def log_debug_trace():
229     stack = traceback.extract_stack()
230     parent_function = stack[-2][2]
231     logging.getLogger('deploy').debug('Calling %s()' % parent_function)
232
233
234 def log_info(msg):
235     logging.getLogger('deploy').info(msg)
236
237
238 def log_and_exit(msg):
239     logging.getLogger('deploy').error(msg)
240     exit(1)
241
242
243 def configure_logging(verbose):
244     FORMAT = '%(asctime)-15s:%(levelname)s:%(name)s: %(message)s'
245     if verbose:
246         logging.basicConfig(level=logging.DEBUG, format=FORMAT)
247     else:
248         logging.basicConfig(level=logging.INFO, format=FORMAT)
249     logging.getLogger('requests.packages.urllib3.connectionpool').setLevel(
250         logging.WARNING)
251
252
253 def main(args):
254     configure_logging(args.verbose)
255     deploy = DeployRequests(args.deploy, args.password)
256
257     with open(args.config_file) as json_data:
258         config = json.load(json_data)
259
260         add_vcenter_credentials(deploy, config)
261
262         add_standalone_host_credentials(deploy, config)
263
264         register_unkown_hosts(deploy, config)
265
266         cluster_id = create_cluster_config(deploy, config)
267
268         deploy_cluster(deploy, cluster_id, config)
269
270
271 def parseArgs():
272     parser = argparse.ArgumentParser(description='Uses the ONTAP Select Deploy API to

```



```
    construct and deploy a cluster.')
273     parser.add_argument('-d', '--deploy', help='Hostname or IP address of Deploy
server')
274     parser.add_argument('-p', '--password', help='Admin password of Deploy server')
275     parser.add_argument('-c', '--config_file', help='Filename of the cluster config')
276     parser.add_argument('-v', '--verbose', help='Display extra debugging messages for
seeing exact API calls and responses',
277                         action='store_true', default=False)
278     return parser.parse_args()
279
280 if __name__ == '__main__':
281     args = parseArgs()
282     main(args)
```

Copyright Information

Copyright © 2020 NetApp, Inc. All rights reserved. Printed in the U.S. No part of this document covered by copyright may be reproduced in any form or by any means-graphic, electronic, or mechanical, including photocopying, recording, taping, or storage in an electronic retrieval system-without prior written permission of the copyright owner.

Software derived from copyrighted NetApp material is subject to the following license and disclaimer:

THIS SOFTWARE IS PROVIDED BY NETAPP “AS IS” AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, WHICH ARE HEREBY DISCLAIMED. IN NO EVENT SHALL NETAPP BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

NetApp reserves the right to change any products described herein at any time, and without notice. NetApp assumes no responsibility or liability arising from the use of products described herein, except as expressly agreed to in writing by NetApp. The use or purchase of this product does not convey a license under any patent rights, trademark rights, or any other intellectual property rights of NetApp.

The product described in this manual may be protected by one or more U.S. patents, foreign patents, or pending applications.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.277-7103 (October 1988) and FAR 52-227-19 (June 1987).

Trademark Information

NETAPP, the NETAPP logo, and the marks listed at <http://www.netapp.com/TM> are trademarks of NetApp, Inc. Other company and product names may be trademarks of their respective owners.