ONTAP

NetApp Automation

NetApp
February 11, 2024

Table of Contents

ONTAP

Day 0/1

Overview of the ONTAP day 0/1 solution

You can use the ONTAP day 0/1 automation solution to deploy and configure an ONTAP cluster using Ansible. The solution is available from the BlueXP automation catalog.

Flexible ONTAP deployment options

Depending on your requirements, you can use on-premises hardware or Simulate ONTAP to deploy and configure an ONTAP cluster using Ansible.

On-premises hardware

You can deploy this solution using on-premises hardware running ONTAP, such as a FAS or an AFF system. You must use a Linux VM to deploy and configure the ONTAP cluster using Ansible.

Simulate ONTAP

To deploy this solution using an ONTAP simulator, you must download the latest version of Simulate ONTAP from the NetApp support site. Simulate ONTAP is a virtual simulator for ONTAP software. Simulate ONTAP runs in a VMware hypervisor on a Windows, Linux, or Mac system. For Windows and Linux hosts, you must use the VMware Workstation hypervisor to run this solution. If you have a Mac OS, use the VMware Fusion hypervisor.

Layered design

The Ansible framework simplifies the development and reuse of automation execution and logic tasks. The framework makes a distinction between decision-making tasks (logic layer), and the execution steps (execution layer) in automation. Understanding how these layers work enables you to customize the configuration.

An Ansible "playbook" executes a series of tasks from start to finish. The `site.yml` playbook contains the `logic.yml` playbook and the `execution.yml` playbook.

When a request is run, the `site.yml` playbook makes a call to the `logic.yml` playbook first, and then calls the `execution.yml` playbook to execute the service request.

You are not required to use the logic layer of the framework. The logic layer provides options to expand the capability of the framework beyond the hard-coded values for execution. This enables you to customize the framework capabilities if required.

Logic layer

The logic layer consists of the following:

- The `logic.yml` playbook
- Logic task files within the `logic-tasks` directory

The logic layer provides the capability for complex decision making without the need for significant custom integration (for example, connecting to ServiceNOW). The logic layer is configurable and provides the input to microservices.

The ability to bypass the logic layer is also provided. If you want to bypass the logic layer, do not define the `logic_operation` variable. Direct invocation of the `logic.yml` playbook provides the ability to do some level of debugging without execution. You can use a "debug" statement to verify that the value of the `raw_service_request` is correct.

Important considerations:

- The `logic.yml` playbook searches for the `logic_operation` variable. If the variable is defined in the request, it loads a task file from the `logic-tasks` directory. The task file must be a .yml file. If there is no matching task file and the `logic_operation` variable is defined, the logic layer fails.
- The default value of the `logic_operation` variable is `no-op`. If the variable is not explicitly defined, it defaults to `no-op`, which does not run any operations.
- If the `raw_service_request` variable is already defined, then execution proceeds to the execution layer. If the variable is not defined, the logic layer fails.

Execution layer

The execution layer consists of the following:

- The `execution.yml` playbook

The execution layer makes the API calls to configure an ONTAP cluster. The `execution.yml` playbook requires that the `raw_service_request` variable is defined when executed.

Support for customization

You can customize this solution in various ways depending on your requirements.

Customization options include:

- Modifying Ansible playbooks
- Adding roles

Customize Ansible files

The following table describes the customizable Ansible files contained in this solution.

Location	Description
`playbooks/inventory/hosts`	Contains a single file with a list of hosts and groups.
`playbooks/group_vars/all/*`	Ansible provides a convenient way to apply variables to multiple hosts at once. You can modify any or all files in this this folder including `cfg.yml`, `clusters.yml`, `defaults.yml`, `services.yml`, `standards.yml`, and `vault.yml`.
`playbooks/logic-tasks`	Supports decision-making tasks within Ansible and maintains the separation of logic and execution. You can add files to this folder that correspond to the relevant service.
`playbooks/vars/*`	Dynamic values used within Ansible playbooks and roles to enable customization, flexibility, and reusability of configurations. If necessary, you can modify any or all files in this folder.

Customize roles

You can also customize the solution by adding or changing Ansible roles, also called microservices. For more details, see Customize.

Prepare to use the ONTAP day 0/1 solution

Before you deploy the automation solution, you must prepare the ONTAP environment and install and configure Ansible.

Initial planning considerations

You should review the following requirements and considerations before using this solution to deploy an ONTAP cluster.

Basic requirements

You must meet the following basic requirements to use this solution:

- You must have access to ONTAP software, either on-premises or through an ONTAP simulator.
- You must know how to use ONTAP software.
- You must know how to use Ansible automation software tools.

Planning considerations

Before deploying this automation solution, you must decide:

- The location where you are going to run the Ansible control node.
- The ONTAP system, either on-premises hardware or an ONTAP simulator.
- Whether or not you will require customization.

Prepare the ONTAP system

Whether you are using an on-premises ONTAP system or Simulate ONTAP, you must prepare the environment before you can deploy the automation solution.

Optionally, install and configure Simulate ONTAP

If you want to deploy this solution through an ONTAP simulator, you must download and run Simulate ONTAP.

Before you begin

- You must download and install the VMware hypervisor that you are going to use to run Simulate ONTAP.
  - If you have a Windows or Linux OS, use VMware Workstation.
  - If you have a Mac OS, use VMware Fusion.

    > ⓘ If you are using a Mac OS, you must have an Intel processor.

Steps

Use the following procedure to install two ONTAP simulators in your local environment:

1. Download Simulate ONTAP from the NetApp support site.

Although you install two ONTAP simulators, you only need to download one copy of the software.

2. If it is not already running, start your VMware application.
3. Locate the simulator file that was downloaded and right click to open it with the VMware application.
4. Set the name of the first ONTAP instance.
5. Wait for the simulator boot up and follow the directions to create a single node cluster.

   Repeat the steps for the second ONTAP instance.

6. Optionally, add a full disk complement.

   From each cluster, run the following commands:

```
security unlock -username <user_01>
security login password -username <user_01>
set -priv advanced
systemshell local
disk assign -all -node <Cluster-01>-01
```

State of the ONTAP system

You must verify the initial state of the ONTAP system, whether it is on-premises or running through an ONTAP simulator.

Verify that the following ONTAP system requirements are met:

- ONTAP is installed and running with no cluster defined yet.
- ONTAP is booted and displaying the IP address to access the cluster.
- The network is reachable.
- You have admin credentials.
- The Message of the Day (MOTD) banner is displayed with the management address.

Install the required automation software

This section provides information on how to install Ansible and prepare the automation solution for deployment.

Install Ansible

Ansible can be installed on Linux or Windows systems.

The default communication method that Ansible uses to communicate with an ONTAP cluster is SSH.

Refer to Getting Started with NetApp and Ansible: Install Ansible to install Ansible.

Ansible must be installed on the control node of the system.

Download and prepare the automation solution

You can use the following steps to download and prepare the automation solution for deployment.

1. Download the ONTAP - Day 0/1 & Health Checks automation solution through the BlueXP web UI. The solution is packaged as `ONTAP_DAY0_DAY1.zip`.
2. Extract the zip folder and copy the files to the desired location on the control node within your Ansible environment.

Initial Ansible framework configuration

Perform the initial configuration of the Ansible framework:

1. Navigate to `playbooks/inventory/group_vars/all`.
2. Decrypt the `vault.yml` file:

   ```
   ansible-vault decrypt playbooks/inventory/group_vars/all/vault.yml
   ```

   When prompted for the vault password, enter the following temporary password:

   ```
   NetApp123!
   ```

   > "NetApp123!" is a temporary password to decrypt the `vault.yml` file and the corresponding vault password. After first use, you **must** encrypt the file using your own password.

3. Modify the following Ansible files:
   - `clusters.yml` - Modify the values in this file to suit your environment.
   - `vault.yml` - After decrypting the file, modify the ONTAP cluster, username and password values to suit your environment.
   - `cfg.yml` - Set the file path for `log2file` and set `show_request` under `cfg` to `True` to display the `raw_service_request`.

     The `raw_service_request` variable is displayed in the log files and during execution.

     > Each file listed contains comments with instructions on how to modify it according to your requirements.

4. Re-encrypt the `vault.yml` file:

   ```
   ansible-vault encrypt playbooks/inventory/group_vars/all/vault.yml
   ```

   > You are prompted to choose a new password for the vault upon encryption.

5. Navigate to `playbooks/inventory/hosts` and set a valid Python interpreter.
6. Deploy the `framework_test` service:

   The following command runs the `na_ontap_info` module with a `gather_subset` value of `cluster_identity_info`. This validates that the basic configuration is correct and verifies that you can communicate with the cluster.

```
ansible-playbook -i inventory/hosts site.yml -e
cluster_name=<CLUSTER_NAME>
-e logic_operation=framework-test
```

Run the command for each cluster.

If successful, you should see output similar to the following example:

```
PLAY RECAP
*************************************************************************
*********
localhost : ok=12 changed=1 unreachable=0 failed=0 skipped=6
The key is ‘rescued=0’ and ‘failed=0’..
```

Deploy the ONTAP cluster using the solution

After completing the preparation and planning, you are ready to use the ONTAP day 0/1 solution to quickly configure an ONTAP cluster using Ansible.

At any time during the steps in this section, you can choose to test a request instead of actually executing it. To test a request, change the `site.yml` playbook on the command line to `logic.yml`.

> ⓘ The `docs/tutorial-requests.txt` location contains the final version of all service requests used throughout this procedure. If you have difficulty running a service request, you can copy the relevant request from the `tutorial-requests.txt` file to the `playbooks/inventory/group_vars/all/tutorial-requests.yml` location and modify the hard-coded values as required (IP address, aggregate names and so on). You should then be able to successfully run the request.

Before you begin

- You must have Ansible installed.
- You must have downloaded the ONTAP day 0/1 solution and extracted the folder to the desired location on the Ansible control node.
- The ONTAP system state must meet the requirements and you must have the necessary credentials.
- You must have completed all required tasks outlined in the Prepare section.

> ⓘ The examples throughout this solution use "Cluster_01" and "Cluster_02" as the names for the two clusters. You must replace these values with the names of the clusters in your environment.

Step 1: Initial cluster configuration

At this stage, you must perform some initial cluster configuration steps.

Steps

1. Navigate to the `playbooks/inventory/group_vars/all/tutorial-requests.yml` location and review the `cluster_initial` request in the file. Make any necessary changes for your environment.
2. Create a file in the `logic-tasks` folder for the service request. For example, create a file called `cluster_initial.yml`.

   Copy the following lines to the new file:

```
- name: Validate required inputs
  ansible.builtin.assert:
    that:
    - service is defined

- name: Include data files
  ansible.builtin.include_vars:
    file:   "{{ data_file_name }}.yml"
  loop:
  - common-site-stds
  - user-inputs
  - cluster-platform-stds
  - vserver-common-stds
  loop_control:
    loop_var:    data_file_name

- name: Initial cluster configuration
  set_fact:
    raw_service_request:
```

3. Define the `raw_service_request` variable.

   You can use one of the following options to define the `raw_service_request` variable in the `cluster_initial.yml` file you created in the `logic-tasks` folder:

   - **Option 1**: Manually define the `raw_service_request` variable.

     Open the `tutorial-requests.yml` file using an editor and copy the content from line 11 to line 165. Paste the content under the `raw service request` variable in the new `cluster_initial.yml` file, as shown in the following examples:

This file contains the final version of the various service
requests used throughout the tutorial in TUTORIAL.md.
#--
#--
cluster_initial:
#
#--
service: cluster_initial
create
std_name: none
req_details:
ontap_aggr:
- hostname: "{{ cluster_name }}"
disk_count: 24
name: n01_aggr1
nodes: "{{ cluster_name }}-01"

Show example

Example `cluster_initial.yml` file:

```
- name: Validate required inputs
  ansible.builtin.assert:
    that:
    - service is defined

- name: Include data files
  ansible.builtin.include_vars:
    file:   "{{ data_file_name }}.yml"
  loop:
  - common-site-stds
  - user-inputs
  - cluster-platform-stds
  - vserver-common-stds
  loop_control:
    loop_var:    data_file_name

- name: Initial cluster configuration
  set_fact:
    raw_service_request:
     service:          cluster_initial
     operation:         create
     std_name:           none
     req_details:

      ontap_aggr:
      - hostname:                   "{{ cluster_name }}"
        disk_count:                 24
        name:                       n01_aggr1
        nodes:                      "{{ cluster_name }}-01"
        raid_type:                  raid4

      - hostname:                   "{{ peer_cluster_name }}"
        disk_count:                 24
        name:                       n01_aggr1
        nodes:                      "{{ peer_cluster_name }}-01"
        raid_type:                  raid4

      ontap_license:
      - hostname:                   "{{ cluster_name }}"
        license_codes:
        - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
        - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
```

```
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA

- hostname:                  "{{ peer_cluster_name }}"
  license_codes:
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
```

```
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
    - XXXXXXXXXXXXXXAAAAAAAAAAAAAA

ontap_motd:
- hostname:                     "{{ cluster_name }}"
  vserver:                      "{{ cluster_name }}"
  message:                      "New MOTD"

- hostname:                     "{{ peer_cluster_name }}"
  vserver:                      "{{ peer_cluster_name }}"
  message:                      "New MOTD"

ontap_interface:
- hostname:                     "{{ cluster_name }}"
  vserver:                      "{{ cluster_name }}"
  interface_name:               ic01
  role:                         intercluster
  address:                      10.0.0.101
  netmask:                      255.255.255.0
  home_node:                    "{{ cluster_name }}-01"
  home_port:                    e0c
  ipspace:                      Default
  use_rest:                     never

- hostname:                     "{{ cluster_name }}"
  vserver:                      "{{ cluster_name }}"
  interface_name:               ic02
  role:                         intercluster
  address:                      10.0.0.101
  netmask:                      255.255.255.0
  home_node:                    "{{ cluster_name }}-01"
  home_port:                    e0c
```

```
    ipspace:                          Default
    use_rest:                         never

  - hostname:                         "{{ peer_cluster_name }}"
    vserver:                          "{{ peer_cluster_name }}"
    interface_name:                   ic01
    role:                             intercluster
    address:                          10.0.0.101
    netmask:                          255.255.255.0
    home_node:                        "{{ peer_cluster_name }}-01"
    home_port:                        e0c
    ipspace:                          Default
    use_rest:                         never

  - hostname:                         "{{ peer_cluster_name }}"
    vserver:                          "{{ peer_cluster_name }}"
    interface_name:                   ic02
    role:                             intercluster
    address:                          10.0.0.101
    netmask:                          255.255.255.0
    home_node:                        "{{ peer_cluster_name }}-01"
    home_port:                        e0c
    ipspace:                          Default
    use_rest:                         never

  ontap_cluster_peer:
  - hostname:                         "{{ cluster_name }}"
    dest_cluster_name:                "{{ peer_cluster_name }}"
    dest_intercluster_lifs:           "{{ peer_lifs }}"
    source_cluster_name:              "{{ cluster_name }}"
    source_intercluster_lifs:         "{{ cluster_lifs }}"
    peer_options:
      hostname:                       "{{ peer_cluster_name }}"
```

- **Option 2**: Use a Jinja template to define the request:

  You can also use the following Jinja template format to get the `raw_service_request` value.

  `raw_service_request: "{{ cluster_initial }}"`

4. Perform the initial cluster configuration for the first cluster:

```
ansible-playbook -i inventory/hosts site.yml -e
cluster_name=<Cluster_01>
```

Verify that there are no errors before proceeding.

5. Repeat the command for the second cluster:

```
ansible-playbook -i inventory/hosts site.yml -e
cluster_name=<Cluster_02>
```

Verify that there are no errors for the second cluster.

When you scroll up towards the beginning of the Ansible output you should see the request that was sent to the framework, as shown in the following example:

Show example

```
TASK [Show the raw_service_request]
*******************************************************************
***************************************
ok: [localhost] => {
    "raw_service_request": {
        "operation": "create",
        "req_details": {
            "ontap_aggr": [
                {
                    "disk_count": 24,
                    "hostname": "Cluster_01",
                    "name": "n01_aggr1",
                    "nodes": "Cluster_01-01",
                    "raid_type": "raid4"
                }
            ],
            "ontap_license": [
                {
                    "hostname": "Cluster_01",
                    "license_codes": [
                        "XXXXXXXXXXXXXXXAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
```

```
                            "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                            "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                            "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                            "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                            "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                            "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                            "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                            "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                            "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                            "XXXXXXXXXXXXXXAAAAAAAAAAAAA"
                        ]
                    }
                ],
                "ontap_motd": [
                    {
                        "hostname": "Cluster_01",
                        "message": "New MOTD",
                        "vserver": "Cluster_01"
                    }
                ]
            },
            "service": "cluster_initial",
            "std_name": "none"
        }
    }
```

6. Log in to each ONTAP instance and verify that the request was successful.

Step 2: Configure the intercluster LIFs

You can now configure the intercluster LIFs by adding the LIF definitions to the `cluster_initial` request and defining the `ontap_interface` microservice.

The service definition and the request work together to determine the action:

- If you provide a service request for a microservice that is not in the service definitions, the request is not executed.
- If you provide a service request with one or more microservices defined in the service definitions, but omitted from the request, the request is not executed.

The `execution.yml` playbook evaluates the service definition by scanning the list of microservices in the order listed:

- If there is an entry in the request with a dictionary key matching the `args` entry contained in the microservice definitions, the request is executed.
- If there is no matching entry in the service request, the request is skipped without error.

Steps

1. Navigate to the `cluster_initial.yml` file that you created previously and modify the request by adding the following lines to the request definitions:

```
ontap_interface:
- hostname:                    "{{ cluster_name }}"
  vserver:                     "{{ cluster_name }}"
  interface_name:              ic01
  role:                        intercluster
  address:                     <ip_address>
  netmask:                     <netmask_address>
  home_node:                   "{{ cluster_name }}-01"
  home_port:                   e0c
  ipspace:                     Default
  use_rest:                    never

- hostname:                    "{{ cluster_name }}"
  vserver:                     "{{ cluster_name }}"
  interface_name:              ic02
  role:                        intercluster
  address:                     <ip_address>
  netmask:                     <netmask_address>
  home_node:                   "{{ cluster_name }}-01"
  home_port:                   e0c
  ipspace:                     Default
  use_rest:                    never

- hostname:                    "{{ peer_cluster_name }}"
  vserver:                     "{{ peer_cluster_name }}"
  interface_name:              ic01
  role:                        intercluster
  address:                     <ip_address>
  netmask:                     <netmask_address>
  home_node:                   "{{ peer_cluster_name }}-01"
  home_port:                   e0c
  ipspace:                     Default
  use_rest:                    never

- hostname:                    "{{ peer_cluster_name }}"
  vserver:                     "{{ peer_cluster_name }}"
  interface_name:              ic02
  role:                        intercluster
  address:                     <ip_address>
  netmask:                     <netmask_address>
  home_node:                   "{{ peer_cluster_name }}-01"
  home_port:                   e0c
  ipspace:                     Default
  use_rest:                    never
```

2. Run the command:

```
ansible-playbook -i inventory/hosts  site.yml -e
cluster_name=<Cluster_01> -e peer_cluster_name=<Cluster_02>
```

3. Log in to each instance to check if the LIFs have been added to the cluster:

Show example

```
Cluster_01::> net int show
  (network interface show)
            Logical    Status     Network            Current
Current Is
Vserver     Interface  Admin/Oper Address/Mask       Node
Port    Home
----------- ---------- ---------- ------------------ -------------
------- ----
Cluster_01
            Cluster_01-01_mgmt up/up 10.0.0.101/24   Cluster_01-01
e0c     true
            Cluster_01-01_mgmt_auto up/up 10.101.101.101/24
Cluster_01-01 e0c true
            cluster_mgmt up/up    10.0.0.110/24      Cluster_01-01
e0c     true
5 entries were displayed.
```

The output shows that the LIFs were **not** added. This is because the `ontap_interface` microservice still needs to be defined in the `services.yml` file.

4. Verify that the LIFs were added to the `raw_service_request` variable.

Show example

The following example shows that the LIFs have been added to the request:


```
            "ontap_interface": [
                {
                    "address": "10.0.0.101",
                    "home_node": "Cluster_01-01",
                    "home_port": "e0c",
                    "hostname": "Cluster_01",
                    "interface_name": "ic01",
                    "ipspace": "Default",
                    "netmask": "255.255.255.0",
                    "role": "intercluster",
                    "use_rest": "never",
                    "vserver": "Cluster_01"
                },
                {
                    "address": "10.0.0.101",
                    "home_node": "Cluster_01-01",
                    "home_port": "e0c",
                    "hostname": "Cluster_01",
                    "interface_name": "ic02",
                    "ipspace": "Default",
                    "netmask": "255.255.255.0",
                    "role": "intercluster",
                    "use_rest": "never",
                    "vserver": "Cluster_01"
                },
                {
                    "address": "10.0.0.101",
                    "home_node": "Cluster_02-01",
                    "home_port": "e0c",
                    "hostname": "Cluster_02",
                    "interface_name": "ic01",
                    "ipspace": "Default",
                    "netmask": "255.255.255.0",
                    "role": "intercluster",
                    "use_rest": "never",
                    "vserver": "Cluster_02"
                },
                {
                    "address": "10.0.0.126",
                    "home_node": "Cluster_02-01",
                    "home_port": "e0c",
                    "hostname": "Cluster_02",
```

```
                        "interface_name": "ic02",
                        "ipspace": "Default",
                        "netmask": "255.255.255.0",
                        "role": "intercluster",
                        "use_rest": "never",
                        "vserver": "Cluster_02"
                    }
                ],
```

5. Define the `ontap_interface` microservice under `cluster_initial` in the `services.yml` file.

   Copy the following lines to the file to define the microservice:

```
         - name: ontap_interface
           args: ontap_interface
           role: na/ontap_interface
```

6. Now that the `ontap_interface` microservice has been defined in the request and the `services.yml` file, run the request again:

```
ansible-playbook -i inventory/hosts  site.yml -e
cluster_name=<Cluster_01> -e peer_cluster_name=<Cluster_02>
```

7. Log in to each ONTAP instance and verify that the LIFs have been added.

Step 3: Optionally, configure multiple clusters

If required, you can configure multiple clusters in the same request. You must provide variable names for each cluster when you define the request.

Steps

1. Add an entry for the second cluster in the `cluster_initial.yml` file to configure both clusters in the same request.

   The following example displays the `ontap_aggr` field after the second entry is added.

```
ontap_aggr:
 - hostname:                     "{{ cluster_name }}"
   disk_count:                   24
   name:                         n01_aggr1
   nodes:                        "{{ cluster_name }}-01"
   raid_type:                    raid4

 - hostname:                     "{{ peer_cluster_name }}"
   disk_count:                   24
   name:                         n01_aggr1
   nodes:                        "{{ peer_cluster_name }}-01"
   raid_type:                    raid4
```

2. Apply the changes for all other items under `cluster_initial`.
3. Add cluster peering to the request by copying the following lines to the file:

```
 ontap_cluster_peer:
 - hostname:                     "{{ cluster_name }}"
   dest_cluster_name:            "{{ cluster_peer }}"
   dest_intercluster_lifs:       "{{ peer_lifs }}"
   source_cluster_name:          "{{ cluster_name }}"
   source_intercluster_lifs:     "{{ cluster_lifs }}"
   peer_options:
     hostname:                   "{{ cluster_peer }}"
```

4. Run the Ansible request:

```
ansible-playbook -i inventory/hosts -e cluster_name=<Cluster_01>
site.yml -e peer_cluster_name=<Cluster_02> -e
cluster_lifs=<cluster_lif_1_IP_address,cluster_lif_2_IP_address>
-e peer_lifs=<peer_lif_1_IP_address,peer_lif_2_IP_address>
```

Step 4: Initial SVM configuration

At this stage in the procedure, you configure the SVMs in the cluster.

Steps

1. Update the `svm_initial` request in the `tutorial-requests.yml` file to configure an SVM and SVM peer relationship.

   You must configure the following:

   - The SVM

   - The SVM peer relationship
   - The SVM interface for each SVM
2. Update the variable definitions in the `svm_initial` request definitions. You must modify the following variable definitions:
   - `cluster_name`
   - `vserver_name`
   - `peer_cluster_name`
   - `peer_vserver`

   To update the definitions, remove the **'{}'** after `req_details` for the `svm_initial` definition and add the correct definition.

3. Create a file in the `logic-tasks` folder for the service request. For example, create a file called `svm_initial.yml`.

   Copy the following lines to the file:

```
- name: Validate required inputs
  ansible.builtin.assert:
    that:
    - service is defined

- name: Include data files
  ansible.builtin.include_vars:
    file:   "{{ data_file_name }}.yml"
  loop:
  - common-site-stds
  - user-inputs
  - cluster-platform-stds
  - vserver-common-stds
  loop_control:
    loop_var:    data_file_name

- name: Initial SVM configuration
  set_fact:
    raw_service_request:
```

4. Define the `raw_service_request` variable.

   You can use one of the following options to define the `raw_service_request` variable for `svm_initial` in the `logic-tasks` folder:

   - **Option 1**: Manually define the `raw_service_request` variable.

     Open the `tutorial-requests.yml` file using an editor and copy the content from line 179 to line

222. Paste the content under the `raw service request` variable in the new `svm_initial.yml` file, as shown in the following examples:

```
svm_initial:
  service:          svm_initial
  operation:        create
  std_name:         none
  req_details:

    ontap_vserver:
    - hostname:                   "{{ cluster_name }}"
      name:                       "{{ vserver_name }}"
      root_volume_aggregate:      n01_aggr1

    - hostname:                   "{{ peer_cluster_name }}"
      name:                       "{{ peer_vserver }}"
      root_volume_aggregate:      n01_aggr1
```

Show example

Example `svm_initial.yml` file:

```
- name: Validate required inputs
  ansible.builtin.assert:
    that:
    - service is defined

- name: Include data files
  ansible.builtin.include_vars:
    file:   "{{ data_file_name }}.yml"
  loop:
  - common-site-stds
  - user-inputs
  - cluster-platform-stds
  - vserver-common-stds
  loop_control:
    loop_var:    data_file_name

- name: Initial SVM configuration
  set_fact:
    raw_service_request:
     service:          svm_initial
     operation:        create
     std_name:         none
     req_details:

      ontap_vserver:
      - hostname:                    "{{ cluster_name }}"
        name:                        "{{ vserver_name }}"
        root_volume_aggregate:       n01_aggr1

      - hostname:                    "{{ peer_cluster_name }}"
       name:                        "{{ peer_vserver }}"
       root_volume_aggregate:       n01_aggr1

      ontap_vserver_peer:
      - hostname:                    "{{ cluster_name }}"
        vserver:                     "{{ vserver_name }}"
        peer_vserver:                "{{ peer_vserver }}"
        applications:                snapmirror
        peer_options:
          hostname:                  "{{ peer_cluster_name }}"

      ontap_interface:
```

```
      - hostname:                  "{{ cluster_name }}"
        vserver:                   "{{ vserver_name }}"
        interface_name:            data01
        role:                      data
        address:                   10.0.0.200
        netmask:                   255.255.255.0
        home_node:                 "{{ cluster_name }}-01"
        home_port:                 e0c
        ipspace:                   Default
        use_rest:                  never

      - hostname:                  "{{ peer_cluster_name }}"
        vserver:                   "{{ peer_vserver }}"
        interface_name:            data01
        role:                      data
        address:                   10.0.0.201
        netmask:                   255.255.255.0
        home_node:                 "{{ peer_cluster_name }}-01"
        home_port:                 e0c
        ipspace:                   Default
        use_rest:                  never
```

- **Option 2**: Use a Jinja template to define the request:

  You can also use the following Jinja template format to get the `raw_service_request` value.

  ```
  raw_service_request: "{{ svm_initial }}"
  ```

5. Run the request:

   ```
   ansible-playbook -i inventory/hosts -e cluster_name=<Cluster_01> -e
   peer_cluster_name=<Cluster_02> -e peer_vserver=<SVM_02>  -e
   vserver_name=<SVM_01> site.yml
   ```

6. Log in to each ONTAP instance and validate the configuration.
7. Add the SVM interfaces.

   Define the `ontap_interface` service under `svm_initial` in the `services.yml` file and run the request again:

```
ansible-playbook -i inventory/hosts -e cluster_name=<Cluster_01> -e
peer_cluster_name=<Cluster_02> -e peer_vserver=<SVM_02>  -e
vserver_name=<SVM_01> site.yml
```

8. Log in to each ONTAP instance and verify that the SVM interfaces have been configured.

Step 5: Optionally, define a service request dynamically

In the previous steps, the `raw_service_request` variable is hard-coded. This is useful for learning, development, and testing. You can also dynamically generate a service request.

The following section provides an option to dynamically produce the required `raw_service_request` if you do not want to integrate it with higher level systems.

- If the `logic_operation` variable is not defined in the command, the `logic.yml` file does not import any file from the `logic-tasks` folder. This means the `raw_service_request` must be defined outside of Ansible and provided to the framework on execution.
- A task file name in the `logic-tasks` folder must match the value of the `logic_operation` variable without the .yml extension.
- The task files in the `logic-tasks` folder dynamically define a `raw_service_request`. The only requirement is that a valid `raw_service_request` be defined as the last task in the relevant file.

How to dynamically define a service request

There are multiple ways to apply a logic task to dynamically define a service request. Some of these options are listed below:

- Using a Ansible task file from the `logic-tasks` folder
- Invoking a custom role that returns data suitable for converting to a `raw_service_request` varaible.
- Invoking another tool outside of the Ansible environment to provide the required data. For example, a REST API call to Active IQ Unified Manager.

The following example commands dynamically define a service request for each cluster using the `tutorial-requests.yml` file:

```
ansible-playbook -i inventory/hosts -e cluster2provision=Cluster_01
-e logic_operation=tutorial-requests site.yml
```

```
ansible-playbook -i inventory/hosts -e cluster2provision=Cluster_02
-e logic_operation=tutorial-requests site.yml
```

Step 6: Deploy the ONTAP day 0/1 solution

At this stage you should have already completed the following:

- Reviewed and modified all files in `playbooks/inventory/group_vars/all` according to your requirements. There are detailed comments in each file to help you make the changes.
- Added any required task files to the the `logic-tasks` directory.
- Added any required data files to the `playbook/vars` directory.

Use the following commands to deploy the ONTAP day 0/1 solution and verify the health of your deployment:

> At this stage, you should have already decrypted and modified the `vault.yml` file and it must be encrypted with your new password.

- Run the ONTAP day 0 service:

```
ansible-playbook -i playbooks/inventory/hosts playbooks/site.yml -e
logic_operation=cluster_day_0 -e service=cluster_day_0 -vvvv --ask-vault
-pass <your_vault_password>
```

- Run the ONTAP day 1 service:

```
ansible-playbook -i playbooks/inventory/hosts playbooks/site.yml -e
logic_operation=cluster_day_1 -e service=cluster_day_0 -vvvv --ask-vault
-pass <your_vault_password>
```

- Apply cluster wide settings:

```
ansible-playbook -i playbooks/inventory/hosts playbooks/site.yml -e
logic_operation=cluster_wide_settings -e service=cluster_wide_settings
-vvvv --ask-vault-pass <your_vault_password>
```

- Run health checks:

```
ansible-playbook -i playbooks/inventory/hosts playbooks/site.yml -e
logic_operation=health_checks -e service=health_checks -e
enable_health_reports=true -vvvv --ask-vault-pass <your_vault_password>
```

Customize the ONTAP day 0/1 solution

To customize the ONTAP day 0/1 solution for your requirements, you can add or change Ansible roles.

Roles represent the microservices within the Ansible framework. Each microservice performs one operation. For example, ONTAP day 0 is a service that contains multiple microservices.

Add Ansible roles

You can add Ansible roles to customize the solution for your environment. Required roles are defined by service definitions within the Ansible framework.

A role must meet the following requirements to be used as a microservice:

- Accept a list of arguments in the `args` variable.
- Use the Ansible "block, rescue, always" structure with certain requirements for each block.
- Use a single Ansible module and define a single task within the block.
- Implement every available module parameter according to the requirements detailed in this section.

Required microservice structure

Each role must support the following variables:

- `mode`: If mode is set to `test` the role attempts to import the `test.yml` which shows what the role does without actually executing it.

  > It is not always possible to implement this because of certain interdependencies.

- `status`: The overall status of playbook execution. If the value is not set to `success` the role is not executed.
- `args` : A list of role specific dictionaries with keys that match the role parameter names.
- `global_log_messages`: Gathers log messages during playbook execution. There is one entry generated each time the role is executed.
- `log_name`: The name used to refer to the role within the `global_log_messages` entries.
- `task_descr`: A brief description of what the role does.
- `service_start_time`: The timestamp used to track the time each role is executed.
- `playbook_status`: The status of the Ansible playbook.
- `role_result`: The variable that contains role output and is included in each message within the `global_log_messages` entries.

Example role structure

The following example provides the basic structure of a role that implements a microservice. You must change the variables in this example for your configuration.

Show example

Basic role structure:

```
- name:  Set some role attributes
  set_fact:
    log_name:     "<LOG_NAME>"
    task_descr:   "<TASK_DESCRIPTION>"

-  name: "{{ log_name }}"
   block:
      -  set_fact:
            service_start_time: "{{ lookup('pipe', 'date
+%Y%m%d%H%M%S') }}"

      -  name: "Provision the new user"
         <MODULE_NAME>:

#--------------------------------------------------------------
            # COMMON ATTRIBUTES

#--------------------------------------------------------------
            hostname:            "{{
clusters[loop_arg['hostname']]['mgmt_ip'] }}"
            username:            "{{
clusters[loop_arg['hostname']]['username'] }}"
            password:            "{{
clusters[loop_arg['hostname']]['password'] }}"

            cert_filepath:       "{{ loop_arg['cert_filepath']
| default(omit) }}"
            feature_flags:       "{{ loop_arg['feature_flags']
| default(omit) }}"
            http_port:           "{{ loop_arg['http_port']
| default(omit) }}"
            https:               "{{ loop_arg['https']
| default('true') }}"
            ontapi:              "{{ loop_arg['ontapi']
| default(omit) }}"
            key_filepath:        "{{ loop_arg['key_filepath']
| default(omit) }}"
            use_rest:            "{{ loop_arg['use_rest']
| default(omit) }}"
            validate_certs:      "{{ loop_arg['validate_certs']
| default('false') }}"
```

```
            <MODULE_SPECIFIC_PARAMETERS>

#--------------------------------------------------------------
            # REQUIRED ATTRIBUTES

#--------------------------------------------------------------
            required_parameter:     "{{ loop_arg['required_parameter']
}}"

#--------------------------------------------------------------
            # ATTRIBUTES w/ DEFAULTS

#--------------------------------------------------------------
            defaulted_parameter:    "{{ loop_arg['defaulted_parameter']
| default('default_value') }}"

#--------------------------------------------------------------
            # OPTIONAL ATTRIBUTES

#--------------------------------------------------------------
            optional_parameter:     "{{ loop_arg['optional_parameter']
| default(omit) }}"
         loop:    "{{ args }}"
         loop_control:
            loop_var:   loop_arg
         register:   role_result

   rescue:
      -  name: Set role status to FAIL
         set_fact:
            playbook_status:   "failed"

   always:
      -  name: add log msg
         vars:
            role_log:
               role: "{{ log_name }}"
               timestamp:
                  start_time: "{{service_start_time}}"
                  end_time: "{{ lookup('pipe', 'date +%Y-%m-
%d@%H:%M:%S') }}"
               service_status: "{{ playbook_status }}"
               result: "{{role_result}}"
         set_fact:
            global_log_msgs:   "{{ global_log_msgs + [ role_log ] }}"
```

Variables used in the example role:

- `<NAME>`: A replaceable value that must be provided for each microservice.
- `<LOG_NAME>`: The short form name of the role used for logging purposes. For example, `ONTAP_VOLUME`.
- `<TASK_DESCRIPTION>`: A brief description of the what the microservice does.
- `<MODULE_NAME>`: The Ansible module name for the task.

> The top level `execute.yml` playbook specifies the `netapp.ontap` collection. If the module is part of the `netapp.ontap` collection, there is no need to fully specify the module name.

- `<MODULE_SPECIFIC_PARAMETERS>`: Ansible module parameters that are specific to the module used to implement the microservice. The following list describes types of parameters and how they should be grouped.
  - Required parameters: All required parameters are specified with no default value.
  - Parameters that have a default value specific to the microservice (not the same as a default value specified by the module documentation).
  - All remaining parameters use `default(omit)` as the default value.

Using multi-level dictionaries as module parameters

Some NetApp provided Ansible modules use multi-level dictionaries for module parameters (for example, fixed and adaptive QoS policy groups).

Using `default(omit)` alone does not work when these dictionaries are used, especially when there is more than one and they are mutually exclusive.

If you need to use multi-level dictionaries as module parameters, you should split the functionality into multiple microservices (roles) so that each one is guaranteed to supply at least one second-level dictionary value for the relevant dictionary.

The following examples show fixed and adaptive QoS policy groups split across two microservices.

The first microservice contains fixed QoS policy group values:

```
fixed_qos_options:
  capacity_shared:           "{{
loop_arg['fixed_qos_options']['capacity_shared']         | default(omit)
}}"
  max_throughput_iops:       "{{
loop_arg['fixed_qos_options']['max_throughput_iops']     | default(omit)
}}"
  min_throughput_iops:       "{{
loop_arg['fixed_qos_options']['min_throughput_iops']     | default(omit)
}}"
  max_throughput_mbps:       "{{
loop_arg['fixed_qos_options']['max_throughput_mbps']     | default(omit)
}}"
  min_throughput_mbps:       "{{
loop_arg['fixed_qos_options']['min_throughput_mbps']     | default(omit)
}}"
```

The second microservice contains the adaptive QoS policy group values:

```
adaptive_qos_options:
  absolute_min_iops:         "{{
loop_arg['adaptive_qos_options']['absolute_min_iops'] | default(omit) }}"
  expected_iops:             "{{
loop_arg['adaptive_qos_options']['expected_iops']     | default(omit) }}"
  peak_iops:                 "{{
loop_arg['adaptive_qos_options']['peak_iops']         | default(omit) }}"
```