

Operating the Cluster — Participant Handout

Ansible for Proxmox VE · Module 05

credativ GmbH

Table of contents

1 One API, one cluster	1
2 <code>module_defaults</code> : write the credentials once	1
3 Storage	2
3.1 The same call with curl	3
4 Users, roles and permissions	4
5 API tokens	5
5.1 Privilege separation	5
5.2 How the failure looks	6
5.3 Which mode to use	6
5.4 The same calls with curl	7
6 Guests	8
6.1 Virtual machines, the minimal case	8
6.2 Installing from a cloud image	8
6.3 Containers	11
6.4 The same calls with curl	12
7 Editing a file on the node directly	12
7.1 The same change through the API	14
7.2 The same calls with curl	14
8 Backup	15
8.1 When no module exists	15
8.2 The same calls with curl	16
9 Verifying what you built	16
10 Further reading	17

1 One API, one cluster

Once the nodes form a cluster, almost everything you configure is cluster-wide. You send an API request to *any* node, and `pmxcfs` replicates the result to all of them. Storage definitions, users, permissions, pools and backup jobs all work this way; only guests and node-local settings are tied to a specific node.

For your playbooks this means: one play against one API endpoint, not a loop over three nodes.

2 `module_defaults`: write the credentials once

Every `community.proxmox` module takes the same four or five API parameters. Repeating them in every task is noise and a source of copy-paste errors.

```

1  ---
2  - name: Configure the cluster
3    hosts: localhost
4    connection: local
5    gather_facts: false
6    module_defaults:
7      group/community.proxmox.proxmox:
8        api_host: "{{ hostvars['pve01'].ansible_host }}"
9        api_user: root@pam
10       api_password: "{{ hostvars['pve01'].pve_root_password }}"
11       validate_certs: false
12    tasks:
13      - name: ...

```

`group/community.proxmox.proxmox` is an *action group*: the collection declares which modules belong to it, and the defaults apply to all of them. Individual tasks can still override any value. Because this play talks only to the API, it runs on `localhost`. There is no need to iterate over hosts at all.

3 Storage

Adding directory storage illustrates the separation of concerns: the directory path on the filesystem is created via SSH, while the cluster-wide storage definition is registered via the Proxmox API.

```

1  - name: Ensure the backup directory exists on every node
2    hosts: pve
3    tasks:
4      - name: Create the directory
5        ansible.builtin.file:
6          path: "{{ pve_backup_path }}"
7          state: directory
8          owner: root
9          group: root
10         mode: "0755"

```

```

1  - name: Register the directory as cluster storage
2    community.proxmox.proxmox_storage:
3      name: pve-backup
4      type: dir
5      dir_options:
6        path: "{{ pve_backup_path }}"
7      content: [backup]
8      nodes: "{{ groups['pve'] }}"
9      state: present

```

content decides what the storage may hold: backup, images, rootdir, iso, vztmpl, snippets. A storage that does not allow backup will refuse backups with an error that does not obviously say so.

The same module handles nfs, cifs, iscsi, zfspool, rbd, cephfs and pbs, each with its own <type>_options dictionary:

```

1  - name: Attach the shared NFS library
2    community.proxmox.proxmox_storage:
3      name: "{{ pve_nfs_storage }}"
4      type: nfs
5      nfs_options:
6        server: "{{ pve_nfs_server }}"
7        export: "{{ pve_nfs_export }}"
8      content: [import, iso, vztmpl]
9      nodes: "{{ groups['pve'] }}"
10     state: present

```

One API call and the share is mounted on every node, below /mnt/pve/<storage name>. Note what content does *not* contain: without images or rootdir, Proxmox VE will never allocate a guest disk there, so the shared library stays a library. That is the difference between the local backup directory from the previous section, which each node owns, and this one, which everybody reads.

3.1 The same call with curl

```

1  TOKEN='PVEAPIToken=root@pam!ansible=xxxxxxxx-xxxx-xxxx-
xxxx-xxxxxxxxxxxx'
2  PVE=https://192.168.0.1:8006/api2/json
3
4  # list the storages (Ansible: community.proxmox.proxmox_storage_info)
5  curl -sk -H "Authorization: $TOKEN" "$PVE/storage" | jq '.data[] |
{storage, type, content}'
6
7  # create a directory storage (Ansible:
community.proxmox.proxmox_storage)
8  curl -sk -H "Authorization: $TOKEN" -X POST "$PVE/storage" \
9    -d storage=pve-backup \
10   -d type=dir \
11   -d path=/var/lib/vz-backup \
12   -d content=backup \
13   -d nodes=pve01,pve02,pve03 | jq

```

API reference: /storage.

i Where that token comes from

Nothing in this workshop creates a token for root@pam, so make one on a node before you try these calls:

```
pveum user token add root@pam ansible --privsep 0
```

The secret is printed once. If you use the automation@pve token from module 05 instead, the cluster-wide endpoints come back **empty rather than failing**: that account holds PVEVMAdmin, which does not include Sys.Audit. An empty result from the Proxmox VE API usually means missing privileges, not missing data.

The module's `dir_options.path` is simply the path parameter; `nfs_options`, `pbs_options` and the others map to that type's parameters in the same way. When a module is missing an option, this page tells you whether the API has it at all.

4 Users, roles and permissions

Proxmox VE separates **authentication realms** from **authorisation**:

Realm	Meaning
@pam	a Linux user that exists on the node (root@pam)
@pve	a user that exists only inside Proxmox VE
LDAP / AD / OIDC	configured realms, for real environments

Permissions are ACL entries: a *path* (`/`, `/vms/9001`, `/storage/pve-backup`), a *subject* (user, group or token) and a *role* (a named set of privileges).

```

1  - name: Ensure the automation group exists
2    community.proxmox.proxmox_group:
3      name: automation
4      comment: Service accounts
5
6  - name: Ensure the service account exists
7    community.proxmox.proxmox_user:
8      name: automation@pve
9      groups: [automation]
10     comment: Ansible service account
11     enable: true
12
13  - name: Grant the group permissions on the whole datacenter
14    community.proxmox.proxmox_access_acl:
15      path: /
16      type: group
17      ugid: automation

```

```
18     roleid: PVEVMAAdmin
19     propagate: 1
```

Built-in roles include PVEVMAAdmin, PVEVMUser, PVEDatastoreAdmin, PVEAuditor and Administrator. If none fits, define your own:

```
1  - name: A role that may only run backups
2    community.proxmox.proxmox_role:
3      roleid: BackupOperator
4      privs:
5        - VM.Backup
6        - Datastore.AllocateSpace
7        - VM.Audit
```

💡 Tip

Grant on the narrowest path that works. / with propagate: 1 is convenient in a lab and almost always too much in production. /vms, /pool/web or /storage/pve-backup are usually what you actually meant.

5 API tokens

A token is a second credential belonging to a user. It can be revoked individually, restricted by its own ACL, and it does not unlock an SSH login.

```
1  - name: Ensure the service account and its token exist
2    community.proxmox.proxmox_user:
3      name: automation@pve
4      groups: [automation]
5      tokens:
6        - tokenid: ansible
7          comment: Created by Ansible
8          privsep: false
9      register: automation_user
```

The module returns the secrets it created in `automation_user.secrets`, keyed by token id. The API returns a token secret **only at creation time**. If you lose it, you delete the token and create a new one.

5.1 Privilege separation

A token is not just a second password for the user. It is a separate identity, `user@realm!` `tokenid`, and it can carry its own ACL entries. `privsep` decides whether it does:

privsep	Proxmox VE calls it	Effective permissions
true	separated privileges	intersection of the user's permissions and the token's own ACL
false	full privileges	identical to the user's

true is the default, both in the API and in proxmox_user. Our playbook sets false explicitly, so the token inherits what automation@pve may do and nothing else has to be granted.

Two consequences follow from the intersection rule, and both cost people time:

A separated token starts with no permissions at all. It does not matter what the user may do: until the token has an ACL entry of its own, the intersection is empty. Grant it like any other subject, with the full token id as ugid:

```

1  - name: Let the token manage guests, and nothing else
2    community.proxmox.proxmox_access_acl:
3      path: /vms
4      type: token
5      ugid: "automation@pve!ansible"
6      roleid: PVEVMAdmin
7      propagate: 1

```

A token can never exceed its user. Granting the token Administrator on / changes nothing if the user does not hold it. That is the point of the intersection: the token is a way to *narrow* an account for one integration, never to widen it.

5.2 How the failure looks

Missing token privileges rarely announce themselves. A write is refused with a permission error, which is at least honest, but a **read comes back empty**:

```

1  # token without Sys.Audit
2  curl -sk -H "Authorization: $TOKEN" "$PVE/cluster/status"
3  # {"data":null}

```

null or [] where you expected data usually means the token lacks the privilege, not that the cluster has nothing to report. Check with pveum acl list before you start debugging the playbook.

5.3 Which mode to use

- **privsep: false** for a service account that exists only for this automation. The account itself is already the scope, and a second layer of ACLs on top buys nothing but confusion. That is our case.
- **privsep: true** when the token hangs off an account that a human also uses, or when one account holds several tokens that should differ in scope — one for inventory, one for guest management.

Either way, set expire (Unix epoch seconds) when the integration is temporary. A token with no expiry outlives the reason it was created for.

⚠ Handling the secret

Do not print it with debug in a shared screen session, and do not commit it in clear text. In this lab it goes into a file with mode 0600; in production it belongs in Ansible Vault or your CI/CD secret store.

Authenticating with the token afterwards:

```
1  module_defaults:
2      group/community.proxmox.proxmox:
3          api_host: "{{ hostvars['pve01'].ansible_host }}"
4          api_user: automation@pve
5          api_token_id: ansible
6          api_token_secret: "{{ pve_api_token_secret }}"
7          validate_certs: false
```

5.4 The same calls with curl

```
1  # create a group          (Ansible: proxmox_group)
2  curl -sk -H "Authorization: $TOKEN" -X POST "$PVE/access/groups" \
3      -d groupid=automation -d comment="Service accounts"
4
5  # create a user          (Ansible: proxmox_user)
6  curl -sk -H "Authorization: $TOKEN" -X POST "$PVE/access/users" \
7      -d userid=automation@pve -d groups=automation -d enable=1
8
9  # create an API token    (Ansible: proxmox_user, tokens:)
10 curl -sk -H "Authorization: $TOKEN" \
11     -X POST "$PVE/access/users/automation@pve/token/ansible" \
12     -d privsep=0 | jq
13 # -> {"data": {"full-tokenid": "automation@pve!ansible", "value":
14     "xxxxxxxx-..."}}
15 # grant a permission    (Ansible: proxmox_access_acl)  note: PUT,
16 not POST
17 curl -sk -H "Authorization: $TOKEN" -X PUT "$PVE/access/acl" \
18     -d path=/ -d groups=automation -d roles=PVEVMAdmin -d propagate=1
```

API reference: /access/users, /access/groups, /access/acl, /access/roles.

The token response is the one place where the API returns a secret, and it returns it exactly once. That is a property of the API, not a limitation of the Ansible module.

6 Guests

6.1 Virtual machines, the minimal case

This creates an empty guest: a disk, some RAM, a network card, no operating system. Useful to see what proxmox_kvm does with nothing else in the way; the section after it builds a guest you can actually log into.

```
1  - name: Ensure an empty VM exists
2    community.proxmox.proxmox_kvm:
3      node: pve02
4      vmid: 9001
5      name: web01
6      cores: 1
7      memory: 512
8      ostype: l26
9      scsihw: virtio-scsi-single
10     scsi:
11       scsi0: "{{ pve_vm_storage }}:1"
12     net:
13       net0: virtio,bridge=vmbr0
14     state: present
```

state accepts present, started, stopped, restarted, absent and a few more. The module identifies the guest by vmid, so a second run reports ok.

Cloning from a template is the common production path, because it gives you a prepared image instead of an empty shell:

```
1  - name: Clone the golden image
2    community.proxmox.proxmox_kvm:
3      node: pve02
4      clone: debian12-template
5      newid: 9010
6      name: web02
7      storage: "{{ pve_vm_storage }}"
8      full: true
9      timeout: 300
```

With a cloud-init enabled template you also set ciuser, sshkeys and ipconfig0, and the guest configures itself on first boot. Template plus cloud-init plus Ansible is what most Proxmox VE automation looks like in practice.

6.2 Installing from a cloud image

Creating an empty VM and then installing an operating system into it is the slow path. A cloud image is a disk that is already installed and carries cloud-init, which configures the guest on first boot from data the hypervisor hands it.


```

1  - name: Ensure the guest exists, installed from the cloud image
2    community.proxmox.proxmox_kvm:
3      node: "{{ pve_vm_node }}"
4      vmid: "{{ pve_vm_id }}"
5      name: "{{ pve_vm_name }}"
6      cores: 1
7      memory: 1024
8      ostype: l26
9      scsihw: virtio-scsi-single
10     scsi:
11         scsi0:      "{{ pve_vm_storage }}:0,import-
from={{ pve_cloud_image_path }}"
12     ide:
13         ide2: "{{ pve_vm_storage }}:cloudinit"
14     net:
15         net0: "virtio,bridge=vmbro"
16     boot: "order=scsi0"
17     serial:
18         serial0: socket
19     ciuser: "{{ pve_cloud_user }}"
20     sshkeys:  "{{ lookup('ansible.builtin.file', '~/.ssh/
id_ed25519.pub') }}"
21     ipconfig:
22         ipconfig0: "ip={{ pve_cloud_vm_ip }},gw={{ pve_cloud_vm_gw }}"
23     timeout: 300
24     state: present

```

⚠ Attach and import in one run: two traps

The storage is mounted lazily. Proxmox VE mounts an NFS storage on first activation, not when you attach it. Importing from it in the same playbook run fails with can't find file on a path that exists as soon as you look at it by hand, which makes the error thoroughly confusing. Activating the storage first solves it:

```

1  - name: Activate the NFS library on the target node and find the image
2    community.proxmox.proxmox_storage_contents_info:
3      node: "{{ pve_vm_node }}"
4      storage: "{{ pve_nfs_storage }}"
5      content: import
6      register: library
7      until: library.proxmox_storage_content | default([])
8            | selectattr('valid', 'search', pve_cloud_image) | list |
count > 0
9      retries: 12
10     delay: 5

```

Waiting for the file to appear with `wait_for` does **not** work, because nothing has asked the node to mount the share.

The import is slower than the default timeout. `proxmox_kvm` waits 30 seconds; reading a few hundred megabytes over NFS and converting it takes longer, especially when multiple nodes import concurrently in a classroom. Set `timeout: 300` on the task.

Four parts of that task are worth reading slowly:

scsi0: "`<storage>:0,import-from=<path>`". The 0 is the size, and 0 means "take it from the source image". `import-from` points at the qcow2 on the NFS library, which every node can read because the storage from the previous section mounted it everywhere. Proxmox VE copies and converts it into a real disk on `local-lvm`; on this lab that takes a few seconds.

ide2: "`<storage>:cloudinit`". A small generated ISO holding the configuration below. Without it the guest boots the image unchanged and ignores everything you set.

ciuser and sshkeys. The user cloud-init creates, and the key it authorises. `cipassword` exists too, but on its own it will not get you in: the Debian cloud image ships with `PasswordAuthentication no`, so a password-only guest is one you can reach on the console and nowhere else.

ipconfig0. Static addressing handed to the guest. `ip=dhcp` works as well when the network has a DHCP server.

```

1  - name: Start it
2    community.proxmox.proxmox_kvm:
3      node: "{{ pve_vm_node }}"
4      vmid: "{{ pve_vm_id }}"

```

```

5     state: started
6
7   - name: Wait until cloud-init has brought up SSH
8     ansible.builtin.wait_for:
9       host: "{{ pve_cloud_vm_ip.split('/')[0] }}"
10      port: 22
11      timeout: 300
12    delegate_to: localhost

```

Import, boot and cloud-init together stay under a minute here. The same guest built from an ISO and an installer takes minutes at best, and needs an answer file that nobody maintains.

i Where the image comes from

The image is a file in the `import/` directory of the NFS library, and it is staged there before the workshop. To fetch one yourself:

```
curl -O https://cloud.debian.org/images/cloud/trixie/latest/debian-13-genericcloud-amd64.qcow2
```

genericcloud is the variant without cloud-provider specifics. There are equivalents for every distribution that matters; the mechanics above do not change.

6.3 Containers

```

1   - name: Download the container template
2     community.proxmox.proxmox_template:
3       node: pve02
4       storage: local
5       content_type: vztpl
6       template: "{{ pve_ct_template }}"
7
8   - name: Ensure the container exists
9     community.proxmox.proxmox:
10      vmid: 9002
11      node: pve02
12      hostname: ct01.example.internal
13      osteamplate: "local:vztpl/{{ pve_ct_template }}"
14      storage: "{{ pve_vm_storage }}"
15      disk: 4
16      cores: 1
17      memory: 512
18      password: "{{ ct_root_password }}"
19      state: present

```

Note the module name: containers are handled by `community.proxmox.proxmox`, without a suffix. It is a historical artefact and it does confuse people.

Do not hard-code the template file name; it carries a version and changes. Ask the node what is on offer and put the answer in a variable:

```
1 pveam update
2 pveam available --section system | grep debian-13
3 # system debian-13-standard_13.x-y_amd64.tar.zst
4 pveam list local # what is already downloaded
```

Your lab runs Proxmox VE 9, so a Debian 13 template is the natural match for a new container, but any of the listed templates works.

6.4 The same calls with curl

```
1 # create a VM (Ansible: proxmox_kvm)
2 curl -sk -H "Authorization: $TOKEN" -X POST "$PVE/nodes/pve02/qemu" \
3     -d vmid=9001 -d name=web01 -d cores=1 -d memory=512 \
4     -d ostype=l26 -d scsihw=virtio-scsi-single \
5     -d scsi0=local-lvm:1 \
6     -d net0=virtio,bridge=vmbro | jq
7
8 # read its configuration (Ansible: proxmox_vm_info)
9 curl -sk -H "Authorization: $TOKEN" "$PVE/nodes/pve02/qemu/9001/config"
10 | jq
11
12 # start it (Ansible: proxmox_kvm, state: started)
13 curl -sk -H "Authorization: $TOKEN" -X POST "$PVE/nodes/pve02/qemu/9001/status/start"
14
15 # clone a template (Ansible: proxmox_kvm, clone:)
16 curl -sk -H "Authorization: $TOKEN" -X POST "$PVE/nodes/pve02/qemu/9000/clone" \
17     -d newid=9010 -d name=web02 -d full=1
```

API reference: `/nodes/{node}/qemu` for VMs and `/nodes/{node}/lxc` for containers. Every parameter of `proxmox_kvm` has the same name here, which makes this page the fastest way to look up what a VM option is actually called.

Each of these returns a UPID. The Ansible module waits for the task; with curl you poll `/nodes/{node}/tasks/{upid}/status` yourself.

7 Editing a file on the node directly

Every automation project reaches a point where no module fits: an option the module does not model, a file no collection knows about, a workaround that has to go in today. Doing it over SSH is legitimate. It is a different trade-off, and one you should make consciously.

A second bridge, written straight into /etc/network/interfaces:

```

1  - name: Add a second bridge the manual way
2    hosts: pve03
3    gather_facts: false
4
5    handlers:
6      - name: Reload network configuration
7        ansible.builtin.command: ifreload -a
8
9    tasks:
10     - name: Keep a pristine copy of the original interface configuration
11       ansible.builtin.copy:
12         src: /etc/network/interfaces
13         dest: /etc/network/interfaces.orig
14         remote_src: true
15         force: false
16         mode: "0644"
17
18     - name: Define vmbr1 in /etc/network/interfaces
19       ansible.builtin.blockinfile:
20         path: /etc/network/interfaces
21         marker: "# {mark} ANSIBLE MANAGED - vmbr1"
22         block: |
23             auto vmbr1
24             iface vmbr1 inet manual
25                 bridge-ports none
26                 bridge-stp off
27                 bridge-fd 0
28         validate: "grep -q 'iface vmbr1' %s"
29         notify: Reload network configuration

```

Three safety measures are doing real work here:

- **force: false** on the copy keeps the original from being overwritten on the second run. A safety net that silently updates itself to the broken state is not a safety net.
- **validate:** runs a command against a temporary copy before the real file is replaced; %s is the temporary path. The check above is minimal. In production, use dedicated syntax validators: visudo -cf %s, nginx -t -c %s, named-checkconf %s.
- **hosts: pve03** limits the blast radius to one node.

⚠ ifreload -a applies immediately

An error affecting vmbro takes the node off the network, including your own SSH session. Never run a networking change against all nodes at once, always keep a console (VDI, IPMI, KVM) available, and test with --check --diff first. This is not paranoia. It is a common way to lose a Proxmox VE node remotely.

7.1 The same change through the API

```
1 - name: The same bridge, through the API
2   community.proxmox.proxmox_node_network:
3     node: pve03
4     state: present
5     iface: vmbro
6     iface_type: bridge
7     bridge_ports: ""
8     autostart: true
```

The module writes to /etc/network/interfaces.new, the same *pending changes* mechanism the GUI uses. The change becomes active on **Apply Configuration**, on reboot, or with ifreload -a. So the API route gives you a review step that the direct edit does not.

	direct edit over SSH	proxmox_node_network
Idempotent	only because blockinfile is	yes
Reviewable before it applies	no, it is live after ifreload	yes, as pending changes
Works without the collection	yes	no
Covers exotic configuration	anything you can write	what the module models
Visible in the GUI	as normal configuration	as pending changes

The rule that follows: **use the module when it covers your case**. When it does not, edit the file, but limit the scope, keep a copy, validate, and have a console open.

7.2 The same calls with curl

```
1 # list the interfaces (Ansible: proxmox_node_network_info)
2 curl -sk -H "Authorization: $TOKEN" "$PVE/nodes/pve03/network" | jq
3   '.data[] | {iface, type, active}'
4
5 # create a bridge (Ansible: proxmox_node_network)
6 curl -sk -H "Authorization: $TOKEN" -X POST "$PVE/nodes/pve03/network" \
7   -d iface=vmbro -d type=bridge -d bridge_ports="" -d autostart=1
```

```

8  # apply the pending changes (the GUI button "Apply Configuration")
9  curl -sk -H "Authorization: $TOKEN" -X PUT "$PVE/nodes/pve03/network"
10
11 # discard the pending changes
12 curl -sk -H "Authorization: $TOKEN" -X DELETE "$PVE/nodes/pve03/network"

```

API reference: `/nodes/{node}/network`.

Note that POST only writes `/etc/network/interfaces.new`; the separate PUT is what applies it, and DELETE throws the pending changes away. That two-step design is the safety net the direct file edit does not have, and the reason the module alone does not activate your change.

8 Backup

```

1  - name: Back up the new VM
2    community.proxmox.proxmox_backup:
3      mode: include
4      vmids: [9001]
5      storage: pve-backup
6      compress: zstd
7      wait: true
8      wait_timeout: 600

```

mode is include, exclude or all. With `wait: true` the task returns when the backup has finished; without it, the task returns as soon as the API has accepted the job, which is rarely what you want in a pipeline.

To add or remove guests from an **existing** scheduled job:

```

1  - name: Ensure the VM is part of the nightly job
2    community.proxmox.proxmox_backup_schedule:
3      vm_id: 9001
4      backup_id: backup-b2adffdc-316e
5      state: present

```

8.1 When no module exists

Creating a scheduled backup job itself is not covered by a module at the time of writing. The fallback is to call the REST API directly:

```

1  - name: Create a nightly backup job
2    ansible.builtin.uri:
3      url: "https://{ pve_api_host }:8006/api2/json/cluster/backup"
4      method: POST
5      validate_certs: false
6      headers:
7        Authorization: "PVEAPIToken={{ pve_api_user }}"

```

```
{{ pve_api_token_id }}={{ pve_api_token_secret }}"
8     body_format: form-urlencoded
9     body:
10        schedule: "02:30"
11        storage: pve-backup
12        mode: snapshot
13        all: 1
14        status_code: [200]
```

You lose idempotency the moment you leave the modules behind, so you add the guard yourself: check for the job first, and run the creation only when it is missing. This is the same discipline as with command, one layer up.

8.2 The same calls with curl

```
1  # run a backup now          (Ansible: proxmox_backup)
2  curl -sk -H "Authorization: $TOKEN" -X POST "$PVE/nodes/pve02/vzdump" \
3      -d vmid=9001 -d storage=pve-backup -d compress=zstd -d mode=snapshot
4  | jq
5
6  # follow the task
7  curl -sk -H "Authorization: $TOKEN" "$PVE/nodes/pve02/tasks/$UPID/
8  status" | jq '.data.status'
9
10 # list the backup jobs      (no Ansible module - this is the uri fallback)
11 curl -sk -H "Authorization: $TOKEN" "$PVE/cluster/backup" | jq
12
13 # create a scheduled job
14 curl -sk -H "Authorization: $TOKEN" -X POST "$PVE/cluster/backup" \
    -d schedule="02:30" -d storage=pve-backup -d mode=snapshot -d all=1
```

API reference: /nodes/{node}/vzdump and /cluster/backup.

wait: true in the Ansible module is exactly the polling loop shown above. Without it, “the backup ran” means “the API accepted the request”.

9 Verifying what you built

```
1  ansible pve01 -m ansible.builtin.command -a 'pvesm status'
2  ansible pve01 -m ansible.builtin.command -a 'pveum user list'
3  ansible pve01 -m ansible.builtin.command -a 'pveum acl list'
4  ansible pve01 -m ansible.builtin.command -a 'qm list'
5  ansible pve01 -m ansible.builtin.command -a 'ls -lh /var/lib/vz-backup/
6  dump/'
```


Every one of these has a matching *_info module if you want the result as data rather than as text: proxmox_storage_info, proxmox_user_info, proxmox_vm_info, proxmox_backup_info.

10 Further reading

The collection: the modules used here

- community.proxmox [collection index](#) (start here and browse; the module list is short enough to read once)
- proxmox_storage
- proxmox_user, proxmox_group, proxmox_role, proxmox_access_acl
- proxmox_kvm (virtual machines), proxmox (containers), proxmox_template
- proxmox_backup, proxmox_backup_schedule
- proxmox_node_network
- [Proxmox API authentication guide](#) (the collection's own page on passwords versus tokens)

Ansible concepts from this module

- [Module defaults](#) (including action groups such as group/community.proxmox.proxmox)
- ansible.builtin.uri (the fallback when no module exists)
- ansible.builtin.blockinfile and its validate: option

Proxmox VE

- [Storage](#): which storage type can hold which content, and why dir behaves differently from lvmthin
- [User Management](#): realms, roles, privileges, API tokens, and the full privilege list for custom roles
- [Backup and Restore](#)
- [Qemu/KVM Virtual Machines](#) and [Linux Containers](#)
- [Network Configuration](#) (bridges, bonds, VLANs, and how pending changes work)
- [API viewer](#): when a module lacks an option, this tells you whether the API has it at all

Offline

```
ansible-doc -l community.proxmox          # the full module list
ansible-doc community.proxmox.proxmox_kvm # every option of the VM module
man pvesm ; man pveum ; man vzdump ; man qm
```