

Ansible for Proxmox VE

credativ hands-on workshop

credativ GmbH

00 • Welcome & the Lab

Alexander Wirt & credativ

Alexander Wirt

- CTO & Co-Owner, credativ (at credativ since 2004)
- Open Source since 1998
- **Debian Developer** since 2003 (formorer)
- Focus: Linux infrastructure, Debian, automation & Proxmox VE
- `alexander.wirt@credativ.de`

credativ

- Open Source consulting & support since 1999
- 24/7/365 enterprise operations & support
- Debian, PostgreSQL, Kubernetes, Ceph, Proxmox VE

- Independent, vendor-neutral engineering

Agenda

			min
01	Ansible in 30 minutes	inventory, ad-hoc, idempotency	30
02	Preparing the nodes	repositories, packages, time	30
03	Forming the cluster	quorum, serial joins	40
	<i>Break</i>		10
04	From playbook to role	defaults, handlers, templates	25
05	Operating the cluster	storage, users, a cloud-init guest	45
06	Dynamic inventory	the plugin, tags as groups	10
07	Where to go next		5

What you leave with

Proxmox VE

- **A cluster you can rebuild:** quorum, joining one node at a time, and why the API answers before the node is a member
- **pmxcfs:** /etc/pve is cluster-wide — root's `authorized_keys` included
- **Storage, users and tokens as state:** realms, privilege separation, NFS that mounts on first use
- The **API viewer:** where every module option comes from

Ansible

- A **working project** of your own: inventory, playbooks, a role, a test suite
- **Idempotency** far enough to read `changed=0` as a test result

- **When a module beats a command**, and what to do when there is none
- The habit of **looking it up**: `ansible-doc`, the module pages

What we leave out

- No general Ansible deep dive: we cover what this setup requires
- No high availability, Ceph, or complex SDN
- No production secret management in the lab (simple variables for speed)

Those are topics for dedicated follow-ups.

Your lab

Machine	Role	Address
desktop	Ansible control node (your desktop)	192.168.0.254
pve01	Proxmox VE node 1 → cluster primary	192.168.0.1
pve02	Proxmox VE node 2	192.168.0.2
pve03	Proxmox VE node 3	192.168.0.3

Log in as root with the password password123 — the same on all three nodes, for SSH and for the web GUI on :8006.

Every participant has an identical, isolated set.

Set up your desktop

Everything is on the share already mounted on your desktop. Run it once:

```
1 ~/Desktop/Workshop/deploy.sh
```

- ~/ansible-proxmox — the project you work in
- ~/ansible-venv — Ansible with the Proxmox collection, put on your PATH
- ~/workshop-slides — these slides
- an SSH key, installed on all three nodes, so nothing asks you for a password

Then open a **new terminal**, or the PATH change is not in effect.

Where the sheets are

In the bookmark bar of your browser, put there by `deploy.sh`:

Handouts, Exercises	one entry per module
Workshop	the page you start on: access, setup, all the PDFs
Material	the same documents as a site, with search
pve01–03, API viewer	the web GUIs of your own nodes

Every module has a handout — the reference, with more detail than these slides — and an exercise sheet.

Checkpoint before we start

On desktop, in a terminal:

```
1 ansible --version           # expect core 2.17 or newer
2 ansible-galaxy collection list community.proxmox
3 ssh root@192.168.0.1 'pveversion'
```

No password and no host key question: `deploy.sh` handled both. If any of these fail, say so now.

If ansible --version looks wrong

Ansible runs from a virtualenv on this desktop, and it should already be on your PATH. If the version is older than 2.17, or `community.proxmox` is missing, you are using the system Ansible instead:

```
1 which ansible                # expect ~/ansible-venv/bin/ansible
2 source ~/ansible-venv/bin/activate # only if it is not
```

① Why a virtualenv: the collection `community.proxmox 2.x` needs a newer proxmoxer than Debian ships, and the packaged collection is missing two modules we use.

You do not have to type everything

~/ansible-proxmox is not an empty directory:

```
1 cd ~/ansible-proxmox
2 ansible-playbook playbooks/00-check.yml
```

- inventory and configuration are **complete**; the addresses are the same for everyone
- every playbook is a **skeleton** with the right task names and structure, and a TODO where the content goes

Writing the files yourself works too. The exercise sheets contain everything either way.

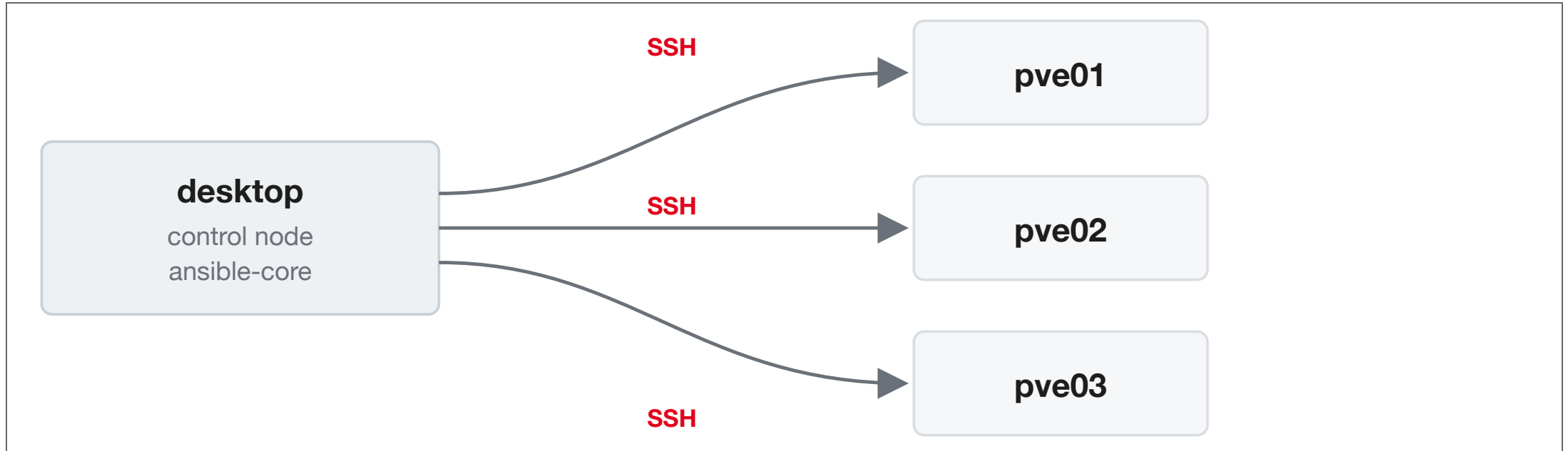
01 • Ansible in 30 Minutes

Learning goals

- Explain what a **control node**, a **managed node** and an **inventory** are
- Run **ad-hoc commands** against all three Proxmox VE nodes
- Describe what **idempotent** means and why it changes how you write things
- Read the output of an Ansible run without guessing

How Ansible works

Agentless, push-based



- Nothing is installed on the managed nodes. Ansible connects over **SSH**.
- The control node runs `ansible-core`; the managed nodes only require Python.
- Playbooks declare the target state, modules execute the necessary changes.

The three things you need

Inventory

Which machines

Module

What to do

Task / Playbook

The plan

Everything else (roles, handlers, variables, templates) is structure on top of these three.

Inventory: your machines, in YAML

```
1 all:
2   children:
3     pve: # group: all Proxmox VE nodes
4       hosts:
5         pve01: {ansible_host: 192.168.0.1}
6         pve02: {ansible_host: 192.168.0.2}
7         pve03: {ansible_host: 192.168.0.3}
8       vars:
9         ansible_user: root
10    pve_primary: # group with exactly one member
11      hosts: {pve01: null}
12    pve_secondary:
13      hosts: {pve02: null, pve03: null}
```

A host may be in several groups. We will use exactly that in module 03.

Check that Ansible reads it the way you meant it:

```
1 ansible-inventory --graph
```

ansible.cfg: stop typing the same flags

```
1 [defaults]
2 inventory      = inventory/hosts.yml
3 host_key_checking = False
4 interpreter_python = auto_silent
5
6 [ssh_connection]
7 pipelining = True
```

⚠ host_key_checking = False disables SSH host key verification. Acceptable in a throwaway lab, **never** in production. There you pre-seed known_hosts.

Doing something

Ad-hoc commands

One module, one run, no file:

```
1 cd ~/ansible-proxmox          # always. ansible.cfg is read from here
2 ansible pve -m ping
3 ansible pve -m ansible.builtin.setup -a 'filter=ansible_distribution*'
4 ansible pve -m ansible.builtin.command -a 'pveversion'
```

- pve is the group from the inventory (all, pve01, pve* also work)
- -m names the module
- -a passes its arguments

Your key is on the nodes, so nothing asks for a password. Where it is not, --ask-pass asks once instead of once per task.

Useful for *looking*. For *changing*, write a playbook.

Run it from the project directory

Ansible reads `ansible.cfg` from the **current working directory**, and that file is what points at your inventory. From anywhere else:

```
1 [WARNING]: No inventory was parsed, only implicit localhost is available
2 [WARNING]: provided hosts list is empty, only localhost is available.
3 [WARNING]: Could not match supplied host pattern, ignoring: pve
```

Three warnings, no error, nothing ran. Almost always a wrong `cd`, not a broken inventory.

Reading the output

A module that returns data:

```
1 pve01 | SUCCESS => {  
2     "ansible_facts": {  
3         "discovered_interpreter_python": "/usr/bin/python3.13"  
4     },  
5     "changed": false,  
6     "ping": "pong"  
7 }
```

SUCCESS means nothing changed, CHANGED means it did. The hosts come back in whatever order they finish — ad-hoc runs them in parallel.

Reading the output

command and shell answer differently: a header line, then raw output.

```
1 pve01 | CHANGED | rc=0 >>  
2 pve-manager/9.2.2/b9984c6d90a4bd80 (running kernel: 7.0.2-6-pve)
```

And when it fails, the return code comes with it:

```
1 pve01 | FAILED | rc=2 >>  
2 Error: Corosync config '/etc/pve/corosync.conf' does not exist - is this  
3 node part of a cluster?
```

Four words to read

Word	Meaning
SUCCESS	already in the desired state, nothing done
CHANGED	Ansible changed something
FAILED	the task ran and did not succeed
UNREACHABLE	Ansible never got to the host

Idempotency

You describe the **target state**, not the steps.

```
1 ansible pve -m ansible.builtin.file \  
2   -a 'path=/root/demo state=directory'
```

- First run → changed
- Second run → ok

A playbook that reports only ok still did its job: it checked every statement and found reality already matching.

command is the last resort

```
1 - name: This runs every single time
2   ansible.builtin.command: pveversion
```

Ansible cannot know what your command does, so it reports changed even for reading a version number. You saw that in exercise 01.



Look for a module first. Nearly everything you would type on a Proxmox VE node has one, or is reachable through the API.

When there really is no module

Then the guard is yours to write:

```
1 - name: Fetch the Debian 13 container template
2   ansible.builtin.command: pveam download local debian-13-standard_13.6-1_amd64.tar.zst
3   args:
4     creates: /var/lib/vz/template/cache/debian-13-standard_13.6-1_amd64.tar.zst
```

- `creates:` — skip the task when that path already exists
- `when:` — a condition of your own
- `changed_when:` / `failed_when:` — decide yourself what counts

The documentation

- Start here: <https://docs.ansible.com/ansible/latest/>
- Playbooks: https://docs.ansible.com/ansible/latest/playbook_guide/index.html
- Ad-hoc commands: https://docs.ansible.com/ansible/latest/command_guide/intro_adhoc.html
- command, with every guard listed:
https://docs.ansible.com/ansible/latest/collections/ansible/builtin/command_module.html

Every module page online has the same content as `ansible-doc <module>` on your desktop, which is offline and matches the version you are actually running.

Exercise 01 — 10 minutes

- ☐ **Know what you are running** — read `ansible.cfg` and the inventory, then check it with `ansible-inventory --graph`
- ☐ **Reach the nodes** — three SUCCESS blocks, and nothing asks for a password
- ☐ **Aim** — one group, then two hosts, then all three. Predict the count before you press return
- ☐ **Run the same thing twice** — first CHANGED, then SUCCESS. The command is not on the sheet; it was on the idempotency slide

Optional block at the end if you get there. → Exercise sheet: *Ansible in 30 minutes*

02 • Preparing the Nodes

Learning goals

- Bring three freshly installed nodes to the state a cluster needs: repositories, packages, name resolution, time
- Write and run the playbook that does it
- Use **variables**, **facts**, **loops** and **handlers** where they belong
- Try a change safely with `--check` `--diff` before you mean it

Why this module exists

Before three Proxmox VE nodes can become a cluster, they need:

Requirement	Why
Working package sources	the enterprise repo fails without a subscription
Name resolution for each other	corosync and pvecm resolve node names
Synchronised time	corosync membership is time-sensitive
The same base state	“it works on pve01” is not a plan

Four requirements, four groups of tasks. That is the playbook.

Playbook structure

A play, in full

playbooks/10-node-prep.yml

```
1 ---
2 - name: Prepare the Proxmox VE nodes
3   hosts: pve           # which machines (a group from the inventory)
4   gather_facts: true   # collect facts first (distribution, IPs, ...)
5   tasks:
6
7     - name: Install the admin tools we want everywhere
8       ansible.builtin.apt:
9         name: "{{ pve_packages }}"
10        state: present
11        update_cache: true
12        cache_valid_time: 3600
```

- name: on every task. It is what you read in the output and in the logs.
- Module arguments are YAML, not a command line

Variables live in group_vars

```
inventory/group_vars/pve.yml
```

```
1 pve_packages:
2   - vim
3   - curl
4   - jq
5   - tmux
```

- Applies to every host in the group pve
- Referenced as { pve_packages }
- Change the list, re-run, done. No playbook edit.

{ } is Jinja

Anything inside { } is an expression, evaluated on the **control node** before the task is sent anywhere.

```
1 name: "{{ pve_packages }}"           # a variable
2 suites: "{{ ansible_distribution_release }}" # a fact
3 loop: "{{ groups['pve'] | difference([inventory_hostname]) }}" # a filter
```

- | pipes a value through a **filter** — here: this group, minus myself
- Quote the value when it starts with {{, or YAML reads the brace as a dict

 Templating · Filters · Jinja itself

Facts: what Ansible found out

gather_facts: true collects a few hundred variables per host before the first task:

```
1 ansible_distribution      = "Debian"
2 ansible_distribution_release = "trixie"      # Debian 13 = Proxmox VE 9
3 ansible_default_ipv4.address = "192.168.0.1"
4 ansible_hostname          = "pve01"
```

Use them instead of hard-coding:

```
1 suites: "{{ ansible_distribution_release }}"
```

Your lab runs Proxmox VE 9 on Debian 13. We use the fact instead of hardcoding trixie.

The four jobs

Loops: one task, many values

```
1 - name: Disable the enterprise repositories
2   ansible.builtin.file:
3     path: "{{ item }}"
4     state: absent
5   loop:
6     - /etc/apt/sources.list.d/pve-enterprise.sources
7     - /etc/apt/sources.list.d/ceph.sources
```

- `loop`: takes a list; the task runs once per entry
- the entry is called `item`
- `loop_control: {label: "{{ item.name }}"}` keeps the output readable when the entries are long

 **Loops** · offline: `ansible-doc -t lookup items`

Repositories

```
1 - name: Disable the enterprise repositories
2   ansible.builtin.file:
3     path: "{{ item }}"
4     state: absent
5   loop:
6     - /etc/apt/sources.list.d/pve-enterprise.sources
7     - /etc/apt/sources.list.d/ceph.sources
8   notify: Update apt cache
9
10 - name: Enable the no-subscription repository
11   ansible.builtin.deb822_repository:
12     name: pve-no-subscription
13     types: [deb]
14     uris: http://download.proxmox.com/debian/pve
15     suites: "{{ ansible_distribution_release }}"
16     components: [pve-no-subscription]
17     signed_by: /usr/share/keyrings/proxmox-archive-keyring.gpg
18   notify: Update apt cache
```

Handlers: react, but only once

```
1 handlers:
2   - name: Update apt cache
3     ansible.builtin.apt:
4       update_cache: true
```

- A handler runs **only if notified**, and **only once** per play
- By default it runs at the **end** of the play

We need the cache *before* installing packages, so we force it:

```
1   - name: Apply pending handlers now
2     ansible.builtin.meta: flush_handlers
```

Name resolution: check, do not configure

It comes from the provisioning, with DNS and /etc/hosts. The playbook confirms it instead of rewriting it:

```
1 - name: Verify that every node resolves the other cluster nodes
2   ansible.builtin.command: "getent hosts {{ item }}"
3   register: pve_resolve
4   changed_when: false
5   failed_when: pve_resolve.rc != 0
6   loop: "{{ groups['pve'] | difference([inventory_hostname]) }}"
```

- `register:` stores the result in a variable — the next line reads its exit code
- `changed_when: false` because reading changes nothing, and command would otherwise report changed every run
- `groups['pve'] | difference([inventory_hostname])`: the inventory at runtime, minus this node

Why not just write /etc/hosts?

Because something else already owns it.

- Proxmox VE resolves its **own** node name through that file
- the provisioning sets it, together with the real domain
- a template that “helpfully” rewrites it replaces the node’s FQDN with whatever domain you invented

Automating a file that another system owns is how you break a platform quietly. Check the state you depend on; configure only what is yours.

Packages, and the clock

```
1 - name: Install the admin tools
2   ansible.builtin.apt:
3     name: "{{ pve_packages }}"
4     state: present
5
6 - name: Check whether the clock is synchronised
7   ansible.builtin.command: timedatectl show -p NTPSynchronized --value
8   register: ntp_state
9   changed_when: false      # this only reads, so never report a change
10  check_mode: false        # ... so it is safe to run it in --check too
11
12 - name: Fail early if time is not in sync
13   ansible.builtin.assert:
14     that: ntp_state.stdout == 'yes'
15     fail_msg: "Time is not synchronised - corosync will be unhappy"
```

What the run looks like

Your `10-node-prep.yml` is still a skeleton — filling it in is the exercise:

```
1 # dry run first: what WOULD change?
2 ansible-playbook playbooks/10-node-prep.yml --check --diff
3
4 # for real
5 ansible-playbook playbooks/10-node-prep.yml
6
7 # again - and now read the recap line
8 ansible-playbook playbooks/10-node-prep.yml
```

```
1 # first run
2 pve01 : ok=8  changed=4  unreachable=0  failed=0  skipped=0
3 # second run
4 pve01 : ok=7  changed=0  unreachable=0  failed=0  skipped=0
```

`changed=0` on the second run is your proof that the playbook is idempotent. One task fewer, too: the handler only runs when something notifies it.

Flags you will use every day

```
1 --check --diff      # dry run, and show the difference
2 --limit pve02       # only this host
3 -v / -vvv          # more output when something is odd
4 --start-at-task "Install the admin tools"
```

--check is not perfect: a task that depends on an earlier change cannot know the future. It still catches most mistakes before they reach three nodes at once.

Exercise 02 — 15 minutes

- ☐ **Fill in the playbook** — replace the four debug placeholders in `playbooks/10-node-prep.yml` with real tasks
- ☐ **See what it would do** — `--check --diff` first, and read the diff
- ☐ **Run it, then run it again** — the second run reports `changed=0`
- ☐ **Two tasks that do not change anything** — find them, and say what would break if they were removed

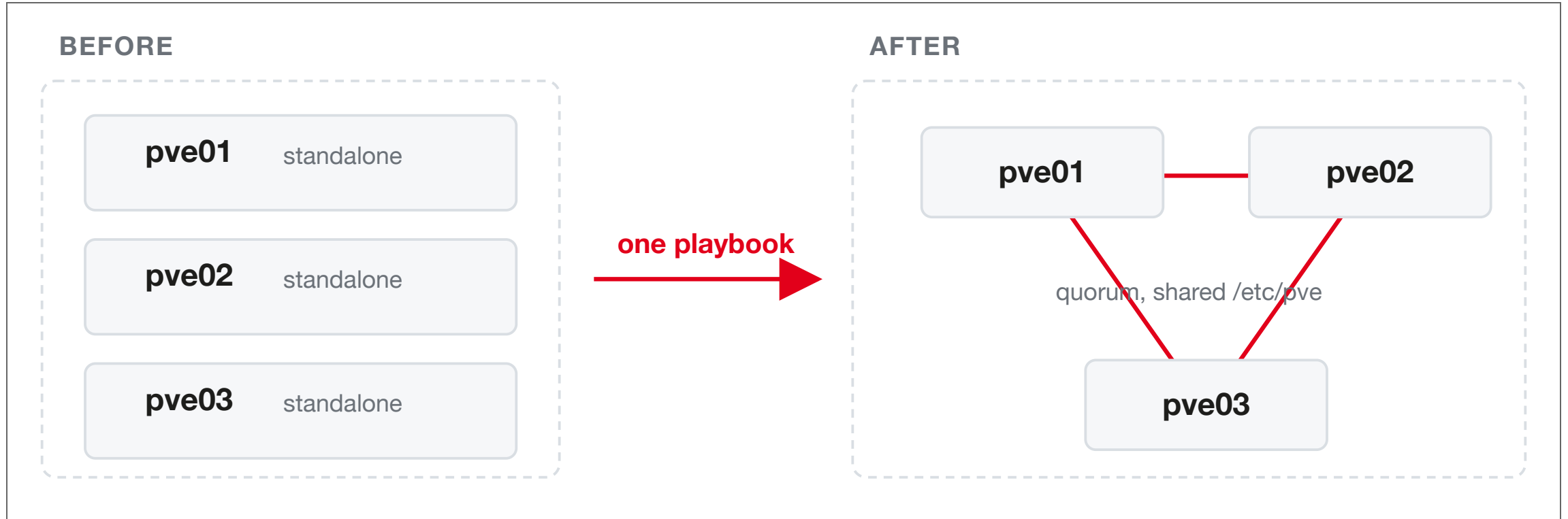
Optional block at the end if you get there. → Exercise sheet: *Preparing the nodes*

03 • Forming the Cluster

Learning goals

- Form a cluster from three separate nodes with `community.proxmox`
- Understand why Proxmox tasks run **on the control node**, not on the node
- Create a cluster and join two nodes, idempotently
- Keep credentials cleanly in variables (and how to handle them in production)

What we are building



One shared configuration filesystem (/etc/pve), one GUI for all three, quorum.

Module vs. command

Raw CLI command

```
1 - ansible.builtin.command:  
2   cmd: pvecm create training  
3   args:  
4     creates: /etc/pve/corosync.conf
```

Works everywhere • you own the guards • parsing output is on you

Dedicated module

```
1 - community.proxmox.proxmox_cluster:  
2   state: present  
3   cluster_name: training
```

Idempotent • check mode • clear errors • needs the collection

We use the collection module. The CLI command path is in the handout as a fallback.

The collection

`community.proxmox` ships around sixty modules plus an inventory plugin.

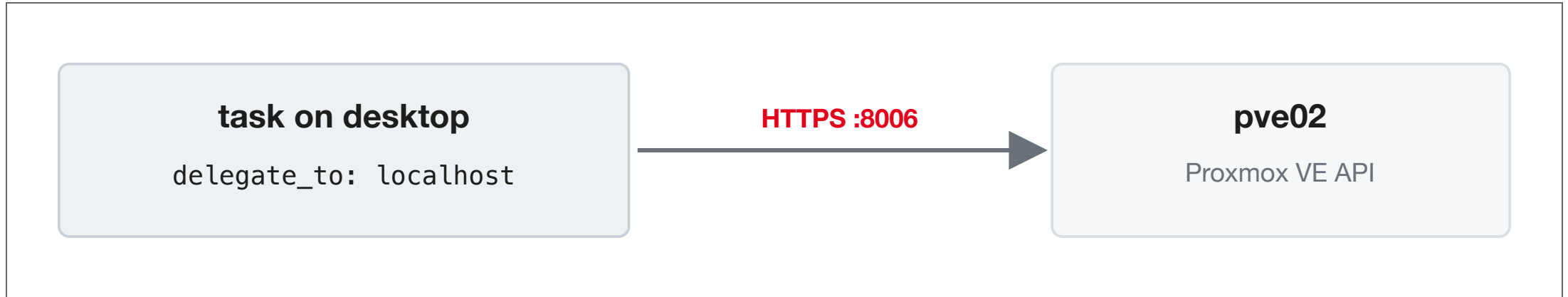
requirements.yml

```
1 ---
2 collections:
3   - name: community.proxmox
4     version: ">=2.0.0"
```

```
1 ansible-galaxy collection install -r requirements.yml
2 pip install --user proxmoxer requests    # or: apt install python3-proxmoxer
```

 The Proxmox modules used to live in `community.general`. They moved to their own collection and were removed from `community.general` in version 11. Old examples on the internet still say `community.general.proxmox_kvm`.

Where does an API module run?



- The module speaks **HTTPS to the API**, so it does not need to run *on* the node
- `proxmoxer` and `requests` must exist where the task runs
- So: run it on the control node with `delegate_to: localhost`

The play still iterates over `pve02` and `pve03`. You keep the per-host variables and only move the execution.

Credentials: lab vs. production

In `inventory/group_vars/pve.yml`:

```
1 pve_cluster_name: training
2 pve_root_password: "password123"
```

- **In this lab:** plain variable to keep things fast and avoid prompt fatigue
- **In production:** keep secrets in Ansible Vault or your CI/CD secret store

```
1 ansible-playbook playbooks/20-cluster.yml
```

The playbook

Play 1: create the cluster

playbooks/20-cluster.yml

```
1 - name: Create the cluster on the primary node
2   hosts: pve_primary
3   gather_facts: false
4   tasks:
5     - name: Ensure the cluster exists
6       community.proxmox.proxmox_cluster:
7         state: present
8         api_host: "{{ ansible_host }}"
9         api_user: root@pam
10        api_password: "{{ pve_root_password }}"
11        validate_certs: false
12        cluster_name: "{{ pve_cluster_name }}"
13        link0: "{{ ansible_host }}"
14        delegate_to: localhost
```

link0 pins corosync to the management network. Be explicit; do not let it guess.

Play 2: join the others, one at a time

```
1 - name: Join the remaining nodes
2   hosts: pve_secondary
3   gather_facts: false
4   serial: 1                      # one node after the other
5   vars:
6     primary: "{{ groups['pve_primary'][0] }}"
7   tasks:
8     - name: Read the join information from the primary
9       community.proxmox.proxmox_cluster_join_info:
10         api_host: "{{ hostvars[primary].ansible_host }}"
11         api_user: root@pam
12         api_password: "{{ pve_root_password }}"
13         validate_certs: false
14         delegate_to: localhost
15         register: join_info
```

Play 2: the join itself

```
1  - name: Join this node to the cluster
2    community.proxmox.proxmox_cluster:
3      state: present
4      api_host: "{{ ansible_host }}"
5      api_user: root@pam
6      api_password: "{{ pve_root_password }}"
7      validate_certs: false
8      master_ip: "{{ hostvars[primary].ansible_host }}"
9      fingerprint: "{{ (join_info.cluster_join.nodelist
10                       | selectattr('name', 'equalto', primary)
11                       | first).pve_fp }}"
12      link0: "{{ ansible_host }}"
13      delegate_to: localhost
```

The fingerprint is how the joining node checks it is talking to the right cluster.

How would you know that?

join_info is whatever the module returned. To find out what is in it, print it:

```
1 - ansible.builtin.debug:
2   var: join_info.cluster_join.nodelist
```

```
1 [
2   {
3     "name": "pve01",
4     "nodeid": "1",
5     "pve_addr": "192.168.0.1",
6     "pve_fp": "47:1F:D4:64:57:2B:0F:B2:5E:0A:87:23:...",
7     "quorum_votes": "1",
8     "ring0_addr": "192.168.0.1"
9   },
10  ...
11 ]
```

var: prints a variable, msg: prints a string you build yourself. Between them and the module's **RETURN VALUES** section in `ansible-doc`, you never have to guess what a registered result contains.

Why serial: 1 matters

Without it, all plays run in parallel and two nodes try to join the same cluster at the same moment.

- corosync has to rewrite its configuration for every new member
- the API of the joining node restarts during the join
- concurrent joins produce a cluster that is *sometimes* fine

serial: 1 turns a race condition into a queue. It costs a few seconds and removes a class of bug that is unpleasant to debug later.

The join is asynchronous

The API accepts the join and returns immediately. The node is *not* a member yet.

```
1  - name: Wait until this node is a quorate cluster member
2    community.proxmox.proxmox_cluster_status_info:
3      api_host: "{{ ansible_host }}"
4      api_user: root@pam
5      api_password: "{{ pve_root_password }}"
6      validate_certs: false
7    delegate_to: localhost
8    register: cl
9    until: cl.cluster_status | default([])
10          | selectattr('type', 'equalto', 'cluster')
11          | selectattr('quorate', 'equalto', true) | list | count > 0
12    retries: 30
13    delay: 5
```

`default([])` matters: the node's API restarts during the join, and without it a single failed call aborts the loop instead of retrying.

Creating the cluster is asynchronous too

```
1 - name: Wait until the primary reports a quorate cluster
2   community.proxmox.proxmox_cluster_status_info:
3     ...
4   register: primary_status
5   until: primary_status.cluster_status | default([])
6         | selectattr('type', 'equalto', 'cluster')
7         | selectattr('quorate', 'equalto', true) | list | count > 0
8   retries: 30
9   delay: 5
```



Skip this and the first join fails with `cluster not ready - no quorum?`. Waiting for port 8006 does not help: pveproxy never went away.

What the run looks like

Your `20-cluster.yml` is still a stub — filling it in is the exercise. This is what it does once it is not:

```
1 ansible-playbook playbooks/20-cluster.yml
```

Then check, from Ansible and from the node:

```
1 ansible pve01 -m ansible.builtin.command -a 'pvecm status'
2 ansible pve01 -m ansible.builtin.command -a 'pvecm nodes'
```

Run the playbook a second time: everything ok, nothing changed.

What quorum means for you

Three nodes, three votes, quorum at 2.

Situation	Quorate	Effect
3 of 3 online	yes	normal operation
2 of 3 online	yes	full operation, no redundancy left
1 of 3 online	no	/etc/pve becomes read-only

A non-quorate node cannot start guests or change configuration. That is a feature: it prevents two halves of a split cluster from both believing they are in charge.

The join rewrites /etc/pve

```
1 /root/.ssh/authorized_keys -> /etc/pve/priv/authorized_keys
```

Root's authorized keys are not a per-node file. They live in pmxcfs, which makes them cluster-wide. A joining node receives the primary's /etc/pve. Keys that existed only on that node are gone afterwards. Your key is on all three nodes since `deploy.sh`, so the join changes nothing for you. On a cluster you build for someone else, it decides whether you can still log in.

Exercise 03 — 20 minutes

- ☐ **Give the playbook the password** — `pve_root_password` in `inventory/group_vars/pve.yml`, and the collection installed
 - ☐ **Fill in the playbook** — no `ansible.builtin.debug` left in `20-cluster.yml`
 - ☐ **Run it** — `serial: 1`, one node at a time. Watch the GUI while it runs
 - ☐ **Prove it, twice** — Quorate: Yes with three nodes, then re-run for `changed=0`
- Optional block at the end if you get there. → Exercise sheet: *Forming the cluster*

04 • From Playbook to Role

Learning goals

- Move `10-node-prep.yml` into a role without changing what the nodes end up as
- Know when node setup has outgrown a single file
- Put values in `defaults/`, logic in `tasks/`, files in `templates/`
- Understand enough about **variable precedence** to stay out of trouble

The problem roles solve

Your `10-node-prep.yml` is 60 lines. It will not stay that way.

- The same tasks are needed in the next project
- Values and logic are mixed in one file
- Two people editing one playbook produce one merge conflict

A role is a **directory layout with conventions**, nothing more.

The layout

```
1 roles/pve_node/
2 |— defaults/main.yml      # values, lowest precedence - meant to be overridden
3 |— vars/main.yml          # values that are NOT meant to be overridden
4 |— tasks/main.yml         # the tasks (no "hosts:", no "- name: play")
5 |— handlers/main.yml      # the handlers
6 |— templates/             # Jinja2 templates (.j2)
7 |— files/                 # files copied verbatim
8 |— meta/main.yml          # dependencies, metadata
```

Ansible loads `main.yml` from each of these automatically. You do not wire anything up yourself.

Before and after

Before

```
1 - name: Prepare the nodes
2   hosts: pve
3   tasks:
4     - name: Disable ...
5       ansible.builtin.file:
6         ...
7   handlers:
8     - name: Update apt cache
9       ...
```

After

site.yml

```
1 - name: Prepare the nodes
2   hosts: pve
3   roles:
4     - pve_node
```

The 60 lines move to
roles/pve_node/tasks/main.yml, unchanged.

defaults/ is the role's interface

```
roles/pve_node/defaults/main.yml
```

```
1 pve_packages: [vim, curl, jq, tmux]
2 pve_repo_component: pve-no-subscription
3 pve_keyring: /usr/share/keyrings/proxmox-archive-keyring.gpg
```

Anyone using your role can override any of these. That is the contract.

`vars/main.yml` is the opposite: values the role needs and callers should not touch. Use it sparingly, because it outranks almost everything.

Variable precedence, the short version

From weakest to strongest, the parts you will actually meet:

1. `roles/x/defaults/main.yml`
2. `inventory/group_vars/`
3. `inventory/host_vars/`
4. `roles/x/vars/main.yml`
5. `vars:` in the play
6. `--extra-vars` on the command line

The working rule: **put values in `defaults/`, override them in `group_vars/`**. That covers almost every case.

Templates

A template renders a file from inventory data and facts. The classic first one is a message of the day:

tasks/main.yml

```
1 - name: Render the message of the day from the inventory
2   ansible.builtin.template:
3     src: motd.j2
4     dest: /etc/motd
5     owner: root
6     group: root
7     mode: "0644"
8   tags: [motd]
```

The template itself

templates/motd.j2

```
1 This node is managed by Ansible (role pve_node).
2
3 node      : {{ inventory_hostname }} ({{ ansible_host }})
4 cluster  : {{ pve_cluster_name | default('not configured') }}
5 peers    : {{ groups['pve'] | difference([inventory_hostname]) | sort | join(', ') }}
```

The logic leaves the YAML. Diffs become readable, and the template can be inspected on its own.

template owns the whole file

That is the difference to `lineinfile` or `blockinfile`, which own a marked region.

	blockinfile	template
Owns	a marked region	the whole file
Foreign changes	survive	are overwritten
Use when	the file has other owners	the file is yours alone

`/etc/motd` is yours. `/etc/hosts` is not – the provisioning writes it, and Proxmox VE resolves its own node name through it. Pick the file before you pick the module.

Tags: running part of a role

A **tag** is a label on a task. Once the tasks carry them, you can run a subset instead of the whole play.

```
1 - name: Disable the enterprise repositories
2   ansible.builtin.file: ...
3   tags: [repos]
```

```
1 ansible-playbook site.yml --list-tags      # what is there to pick from
2 ansible-playbook site.yml --tags repos     # only those
3 ansible-playbook site.yml --skip-tags packages
```

Tags

 A tagged partial run can leave a host in a state that no full run would produce. Tags are for fast iteration while developing, not a substitute for structure.

One more habit: ansible-lint

```
1 pip install --user ansible-lint
2 ansible-lint site.yml roles/
```

It catches the things reviewers would otherwise catch for you: missing name:, shell where command suffices, deprecated syntax, unsafe file permissions.

Exercise 04 — 10 minutes

- ☐ **Give the role its defaults** — `pve_packages` appears exactly once in the whole project when you are done
- ☐ **Move the tasks and the handler** — into `roles/pve_node/`, nothing left behind
- ☐ **Render the message of the day** — a template task carrying tags: `[motd]`
- ☐ **Run it three times** — the first run after the conversion must report `changed=0`, and that is the whole point of the exercise

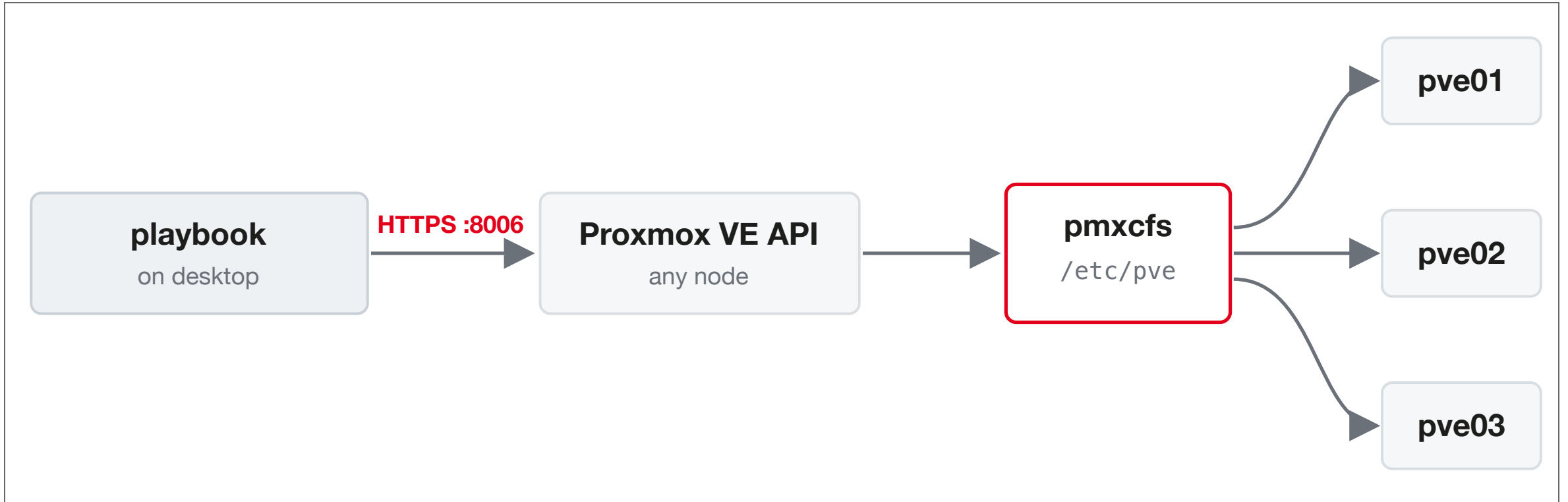
Optional block at the end if you get there. → Exercise sheet: *From playbook to role*

05 • Operating the Cluster

Learning goals

- Add **storage** to the cluster: a local directory and the shared NFS library
- Create **users, roles, permissions** and an **API token**, then use it
- Create a **VM** from a cloud image and **back it up**, from the same playbook
- **Start, stop, migrate and delete** it — and see where that quietly does nothing
- Stop repeating API credentials with `module_defaults`
- See what it costs to **edit a config file on the node** instead

The pattern is always the same



One API call, one cluster-wide change. You address **any** node; pmxcfs distributes the result.

Look it up: the API browser

Every node serves the full API reference, for exactly the version you are running:

```
1 https://192.168.0.1:8006/pve-docs/api-viewer/
```

- searchable tree of every endpoint, with **every parameter and its type**
- the public copy: <https://pve.proxmox.com/pve-docs/api-viewer/>

Use it when a module lacks an option. The page tells you whether the API has it at all — which decides between “file a feature request” and “use `ansible.builtin.uri`”.

Module option, API parameter, same name

```
1 community.proxmox.proxmox_kvm:
2   node: pve02
3   vmid: 9001
4   cores: 1
5   memory: 512
6   scsihw: virtio-scsi-single
```

POST /nodes/{node}/qemu in the API browser lists `vmid`, `cores`, `memory`, `scsihw` — the same names. That correspondence is what makes the browser the fastest way to find out what a guest option is actually called.

Stop repeating yourself: module_defaults

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

Every `community.proxmox` module in this play inherits those four lines.

Storage

Two halves of one job

```
1 # 1 - on the nodes, over SSH
2 - name: Ensure the backup directory exists
3   hosts: pve
4   tasks:
5     - ansible.builtin.file:
6       path: /var/lib/vz-backup
7       state: directory
8       mode: "0755"
```

The other half, over the API

```
1 # 2 - on the cluster, over the API
2 - name: Register the directory as storage
3   community.proxmox.proxmox_storage:
4     name: pve-backup
5     type: dir
6     dir_options:
7       path: /var/lib/vz-backup
8     content: [backup, snippets]
9     nodes: "{{ groups['pve'] }}"
10    state: present
```

The node is a Debian machine **and** a Proxmox VE node.

A second storage, a shared one

The backup directory is local to each node. An NFS library is shared, and it is where images and templates live in a real environment:

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

- one API call, mounted on **every** node under `/mnt/pve/nfs-shared`
- content decides what may live there; no `images`, so nothing gets allocated on it
- this is where the cloud image for the next slide comes from

Users and permissions

Group, user, permission

```
1 - community.proxmox.proxmox_group:
2   name: automation
3   comment: Service accounts
4
5 - community.proxmox.proxmox_user:
6   name: automation@pve
7   groups: [automation]
8   comment: Ansible service account
9   enable: true
10
11 - community.proxmox.proxmox_access_acl:
12   path: /
13   type: group
14   ugid: automation
15   roleid: PVEVMAdmin
16   propagate: 1
```

Realms matter: @pam are Linux users on the node, @pve exist only in Proxmox VE.

API tokens instead of passwords

```
1 - community.proxmox.proxmox_user:
2   name: automation@pve
3   groups: [automation]
4   tokens:
5     - tokenid: ansible
6       comment: Created by Ansible
7       privsep: false
8   register: automation_user
```

- The secret is returned **once**, at creation, in `automation_user.secrets`
- `privsep: false` gives the token the user's permissions; `true` restricts it further with its own ACL

⚠ Never print a secret with debug in a shared session. Write it somewhere safe, ideally straight into an encrypted file.

Then authenticate with the token

```
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
```

A token can be revoked on its own, is scoped by ACL, and never unlocks an SSH session. The root password can do none of those things.

Guests

A guest from a cloud image

```
1 - community.proxmox.proxmox_kvm:
2   node: "{{ pve_vm_node }}"
3   vmid: "{{ pve_vm_id }}"
4   name: "{{ pve_vm_name }}"
5   cores: 1
6   memory: 1024
7   ostype: l26
8   scsihw: virtio-scsi-single
9   scsi:
10    scsi0: "{{ pve_vm_storage }}:0,import-from={{ pve_cloud_image_path }}"
11  ide:
12    ide2: "{{ pve_vm_storage }}:cloudinit"
13  net:
14    net0: virtio,bridge=vmbr0
15  boot: "order=scsi0"
16  timeout: 300          # NFS import is slower than the 30 s default
17  state: present
```

- :0 means “size taken from the source image”
- import-from reads the qcow2 straight off the NFS library
- ide2: ...:cloudinit adds the cloud-init drive

Lazy mounts bite here

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

- Proxmox VE mounts an NFS storage **on first activation**, not when you attach it
- so importing in the same run fails with can't find file
- listing the contents activates it and confirms the image

Waiting for the path does not help: nothing has asked the node to mount it yet.

cloud-init does the rest

```
1 ciuser: "{{ pve_cloud_user }}"
2 sshkeys: "{{ lookup('ansible.builtin.file', '~/.ssh/id_ed25519.pub') }}"
3 ipconfig:
4     ipconfig0: "ip={{ pve_cloud_vm_ip }},gw={{ pve_cloud_vm_gw }}"
```

The guest configures its user, its SSH key and its address on first boot. No installer, no answer file, no template maintenance.



! cpassword alone will not get you in: the Debian cloud image ships with PasswordAuthentication no. Set sshkeys.

Anything else needs a snippet

ciuser, sshkeys, ipconfig is the whole of what Proxmox VE generates. Packages, files, commands come from a cloud-config file on a storage with snippets content:

```
1 - community.proxmox.proxmox_kvm:
2   agent: "enabled=1"
3   cicustom: "vendor={{ pve_backup_storage }}:snippets/qemu-guest-agent.yaml"
```

```
1 #cloud-config
2 package_update: true
3 packages:
4   - qemu-guest-agent
5 runcmd:
6   - [systemctl, start, qemu-guest-agent]
```

⚠ Important

vendor= is applied **in addition** to what Proxmox VE generates. user= **replaces** it — same syntax, and your key is gone.

From nothing to reachable in 30 seconds

```
1 - community.proxmox.proxmox_kvm:
2   node: "{{ pve_vm_node }}"
3   vmid: "{{ pve_vm_id }}"
4   state: started
5
6 - ansible.builtin.wait_for:
7   host: "{{ pve_cloud_vm_ip.split('/')[0] }}"
8   port: 22
9   timeout: 300
10  delegate_to: localhost
```

Import, boot and cloud-init together take well under a minute on this lab.

Containers are one module away

```
1 - community.proxmox.proxmox:
2   vmid: 9002
3   node: pve02
4   hostname: ct01.example.internal
5   otemplate: "local:vztmpl/{{ pve_ct_template }}"
6   storage: "{{ pve_vm_storage }}"
7   disk: 4
8   cores: 1
9   memory: 512
10  password: "{{ ct_root_password }}"
11  state: present
```

The template has to be on the storage first; proxmox_template downloads it. Never hard-code the file name; ask the node which ones exist:

```
1 pveam update && pveam available --section system | grep debian-13
```

Start, stop, delete

state covers the whole life of a guest, not just its creation:

started	stopped	restarted	power
paused	hibernated	suspend to RAM, suspend to disk	
present	absent	the guest exists, or it does not	
template	freeze it as a template to clone from		
current	read the state, change nothing		

```
1 - community.proxmox.proxmox_kvm:
2   vmid: 9001
3   state: stopped
```

No node: — for an existing guest the cluster already knows where it runs. stopped is a graceful ACPI shutdown; force: true pulls the plug.

Running the same thing twice

```
1 TASK [Stop it] ***** changed: [localhost]
2 TASK [Stop it again] ***** ok: [localhost]
```

Declaring a state, not performing an action. Same for started, absent.
absent on a *running* guest neither deletes it nor fails:

```
1 TASK [Remove it while it runs] ***
2 ok: [localhost] => {
3     "changed": false,
4     "msg": "VM 9001 is running. Stop it before deletion or use force=true."
5 }
```

⚠ Important

ok, changed: false, failed=0 in the recap — and the guest is still running. The only sign that nothing happened is the msg. Set state: stopped first, or add force: true and the module stops it for you.

Migration

```
1 - community.proxmox.proxmox_kvm:
2   vmid: 9001
3   node: pve01           # where it should end up
4   migrate: true
5   with_local_disks: true
6   timeout: 600
```

15 seconds on this lab, running guest, no interruption. A second run reports VM 9001 is already on pve01.

Drop `with_local_disks` and the node refuses before it starts:

```
1 can't live migrate attached local disks without with-local-disks option
```

The module does not see that. It polls until `timeout` and then says `Reached timeout while waiting for migrating VM` — ten minutes of waiting for a job that died in the first second.

Note

Why the disks matter: `local-lvm` lives on one node, so a migration has to copy it. On shared storage there is nothing to copy. When a migration hangs, the reason is in the node's task log – GUI, or `pvesh get /nodes/pve01/tasks`.

The other route

Sometimes you really do edit the file

Not everything has a module, and some things are simply faster to write directly. A second bridge, over SSH:

```
1 - name: Add a second bridge
2   hosts: pve
3   tasks:
4     - name: Define vmbr1 in /etc/network/interfaces
5       ansible.builtin.blockinfile:
6         path: /etc/network/interfaces
7         marker: "# {mark} ANSIBLE MANAGED - vmbr1"
8         block: |
9             auto vmbr1
10            iface vmbr1 inet manual
11                bridge-ports none
12                bridge-stp off
13                bridge-fd 0
14       notify: Reload network configuration
```

Applying it is then your problem

```
1 handlers:
2   - name: Reload network configuration
3     ansible.builtin.command: ifreload -a
```

⚠ `ifreload -a` applies the change **immediately**. A mistake in the file for `vmbri0` takes the node off the network, including your own SSH session. Always keep a console (VDI, IPMI) available, and never touch the management interface of all three nodes in one run.

The same thing, through the API

```
1 - community.proxmox.proxmox_node_network:
2   node: pve01
3   state: present
4   iface: vmbr1
5   iface_type: bridge
6   bridge_ports: ""
7   autostart: true
```

- writes to `/etc/network/interfaces.new`, the same **pending** state the GUI shows
- becomes active on *Apply Configuration*, on reboot, or via `ifreload -a`

Both routes end in the same file. Only one of them is idempotent and reviewable before it takes effect.

Which route when

	blockinfile over SSH	proxmox_node_network
Idempotent	you build the guard	yes
Visible in the GUI as pending	no, it is already in the live file	yes
Works without the collection	yes	no
Risk of locking yourself out	high	lower, but real

Use the module when it covers your case. Reach for the file when it does not, and then go carefully: one node at a time, `--limit`, console at hand.

Backup

Run a backup

```
1 - community.proxmox.proxmox_backup:
2   mode: include
3   vmids: [9001]
4   storage: pve-backup
5   compress: zstd
6   wait: true
7   wait_timeout: 600
```

`wait: true` makes the task finish when the backup finishes, not when the API has accepted the request.

What has no module yet

Creating a **scheduled** backup job is not covered by a module today. `proxmox_backup_schedule` only adds guests to a job that already exists.

```
1 - name: Create a nightly backup job
2   ansible.builtin.uri:
3     url: "https://{{ api_host }}:8006/api2/json/cluster/backup"
4     method: POST
5     ...
```

`ansible.builtin.uri` against the REST API works for anything. You lose idempotency, so you add the guard yourself, the same way you do with `command`.

Exercise 05 — 20 minutes

- ☐ **Check the ground** — what storage does the cluster have before you add any
- ☐ **Storage, in two halves** — the directory and the snippet over SSH, the registration and the NFS library over the API
- ☐ **A service account with a token** — group, user, token, ACL
- ☐ **A guest that installs itself** — activate the library, create the VM from the cloud image, start it, wait for SSH. Do not run it yet
- ☐ **The direct route, then run it** — `blockinfile` on `/etc/network/interfaces`, then the whole playbook, twice

Optional block at the end if you get there. → Exercise sheet: *Operating the cluster*

06 • Dynamic Inventory

Learning goals

- Understand why a static inventory goes stale
- Configure the `community.proxmox.proxmox` inventory plugin
- Read the groups and facts it generates
- Build your own groups from **Proxmox tags**

The problem

```
1 pve01: {ansible_host: 192.168.0.1}  
2 web01: {ansible_host: 192.168.0.50}  
3 db01:  {ansible_host: 192.168.0.51}
```

- Someone creates a VM in the GUI → it is not in your inventory
- Someone deletes one → your playbook fails on a host that no longer exists
- Someone renames one → you find out at the worst possible moment

The cluster already knows all of this. Ask it.

The plugin

inventory/lab.proxmox.yml

```
1 plugin: community.proxmox.proxmox
2 url: https://192.168.0.1:8006
3 user: automation@pve
4 token_id: ansible
5 token_secret: "{{ lookup('ansible.builtin.env', 'PVE_TOKEN_SECRET') }}"
6 validate_certs: false
7 want_facts: true
8 exclude_nodes: false
```

❗ The file name **must** end in `.proxmox.yml` or `.proxmox.yaml`. Otherwise the plugin politely ignores it and you debug an empty inventory.

What you get

```
1 ansible-inventory -i inventory/lab.proxmox.yml --graph
```

```
1 @all:
2   |--@proxmox_all_qemu:
3   |   |--web01
4   |--@proxmox_all_running:
5   |   |--web01
6   |--@proxmox_all_stopped:
7   |--@proxmox_nodes:
8   |   |--pve01
9   |   |--pve02
10  |   |--pve03
11  |--@proxmox_pve02_qemu:
12  |   |--web01
```

Groups per node, per type, per status, per pool. Generated, never maintained.

Facts, if you ask for them

With `want_facts: true` every guest carries its Proxmox configuration:

```
1 ansible-inventory -i inventory/lab.proxmox.yml --host web01
```

```
1 {
2   "proxmox_vmid": 9001,
3   "proxmox_node": "pve02",
4   "proxmox_status": "stopped",
5   "proxmox_maxmem": 536870912,
6   "proxmox_tags_parsed": ["web", "staging"]
7 }
```

Tags become groups

Tag a guest in Proxmox VE, group on it in Ansible:

```
1 keyed_groups:
2   - key: proxmox_tags_parsed
3     separator: ""
4     prefix: tag
5
6 groups:
7   webserver: "'web' in (proxmox_tags_parsed | list)"
8   production: "'prod' in (proxmox_tags_parsed | list)"
```

The person creating the VM decides its role, in the GUI, with a tag, and your playbooks follow.

Static and dynamic together

```
1 ansible-playbook site.yml -i inventory/hosts.yml -i inventory/lab.proxmox.yml
```

Or set both in `ansible.cfg`:

```
1 [defaults]
2 inventory = inventory/hosts.yml,inventory/lab.proxmox.yml
```

The nodes stay static, because they have to exist before anything is dynamic. The guests come from the cluster.

Exercise 06 — 8 minutes

- ❑ **Hand the plugin the token** — the secret from `~/pve-token.json` in the environment; `echo ${#PVE_TOKEN_SECRET}` prints 36
- ❑ **Ask the cluster** — the graph shows `proxmox_nodes` with your three nodes, and `web01` from module 05
- ❑ **Let a tag decide the group** — tag the guest, then build `webserver`s from it
- ❑ **Both inventories at once** — static groups and generated groups in the same run

Optional block at the end if you get there. → Exercise sheet: *Dynamic inventory*

07 • Where to Go Next

What you built in 3.5 hours

```
1 ~/ansible-proxmox/
2 |— ansible.cfg
3 |— requirements.yml
4 |— site.yml
5 |— inventory/
6 |   |— hosts.yml           # the three nodes
7 |   |— lab.proxmox.yml    # everything else, from the API
8 |   |— group_vars/
9 |— playbooks/
10 |   |— 20-cluster.yml     # forms the cluster
11 |   |— 30-operations.yml  # storage, users, guests, backup
12 |— roles/pve_node/       # the reusable part
```

Summary

Proxmox VE

1. A cluster is a **state you describe**: quorum, one join at a time, and an API that answers before the node is a member
2. **/etc/pve is cluster-wide**. pmxcfs distributes it — root's `authorized_keys` included
3. **Storage, users and tokens are state too**. Least privilege: a scoped token, not the root password
4. **The API viewer** is where module options come from, and what to reach for when one is missing
Ansible, as far as that needed
5. Describe the target state rather than the steps; **changed=0 is a test result**
6. **A module beats a command** — and when there is none, write the guard yourself

What we did not cover

Topic	Where it belongs
Ceph, HA, SDN, QDevice	<i>Proxmox VE Clustering & Shared Storage</i>
Cloud-init images, golden templates	a VM lifecycle workshop
AWX / Ansible Automation Platform, Semaphore	an Ansible operations workshop
Molecule, CI for roles	an Ansible testing workshop
Ansible for the guests inside the VMs	a general Ansible course

The obvious next step

Your playbooks currently run from your laptop. That works exactly as long as it is only you.

- **Git** first: the repository is the source of truth, not a directory
- **CI** second: run `ansible-lint` and `--check` on every merge request
- **AWX / AAP or Semaphore** third: scheduled runs, an audit trail, credentials that live somewhere other than a developer's home directory

In that order.

Still curious?

The **bonus exercises** go further than the workshop had time for:

- make the backup task idempotent
- scope the API token to a resource pool
- build a scheduled backup job with `ansible.builtin.uri`
- turn the guest into a template and clone from it
- let Proxmox tags drive a play
- write a read-only health check playbook

Solutions included. They assume the state you have right now.

Where to look things up

- Ansible: <https://docs.ansible.com/ansible/latest/>
- The collection: <https://docs.ansible.com/ansible/latest/collections/community/proxmox/>
- Proxmox VE: <https://pve.proxmox.com/pve-docs/>
- The API of your own cluster: `https://<node>:8006/pve-docs/api-viewer/`
- Offline, always correct for your version: `ansible-doc <module>`

Bookmark the third and the fifth. Those are the two you will actually use.

Feedback

- What was too fast, what was too slow?
- What did you expect that did not happen?
- What would you have cut to get more of something else?

Afterwards, and for anything longer: alexander.wirt@credativ.de

Thank you

Questions, discussion, and code:

- All slides, handouts, labs, and reference code in code/ are yours to keep
- credativ GmbH · <https://www.credativ.de>



credativ · Ansible for Proxmox VE

Speaker notes