Ansible

A Complete 10-Chapter Configuration Management Course

Topics covered:

Agentless configuration management vs. Terraform · Inventory & playbooks

Modules, idempotency & handlers · Jinja2 templates & variables

Roles & Ansible Galaxy · Vault secrets management

Dynamic inventory & Molecule testing · Capstone: automating ws1's hardening course

Exercises: 30 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples · builds directly on the site's own Terraform, Cloud Platforms, and Observability courses

Table of Contents

CHAPTER 1 OF 10

What Ansible Is & Why It's Different From Terraform

Ansible

Chapter 1 · What Ansible Is & Why It's Different From Terraform

`tf1-1` named Ansible directly, back at the start of the Terraform course: "still on this site's own bucket list." `cloud1-11` named the real lesson behind configuration drift as "fix the config, not just the console," without yet having the tool to act on it. This course is both of those promises, paid off.

Two Different Jobs — Provisioning vs. Configuration Management

`tf1-1` already drew this line precisely: Terraform **provisions** infrastructure — it creates the VM, the network, the database instance. It has no concept of "log in and install a package." Ansible **configures** infrastructure that already exists — installing software, editing config files, restarting services, on machines Terraform (or something else) has already brought into being. A real pipeline commonly runs Terraform first, to create the servers, then Ansible, to configure them — complementary tools, not competing ones, exactly as `tf1-1` already put it.

Agentless — Ansible's Own Defining Architectural Choice

Older configuration management tools — Puppet, Chef — work by installing a persistent **agent** on every managed node: a daemon that runs continuously, periodically checking in with a central server. Ansible takes a deliberately different approach: it's **agentless**. It connects over plain SSH (or WinRM for Windows), pushes a small amount of code to the target, runs it, and cleans up — nothing persistent left running on the managed host afterward. A new server can be managed the moment it has SSH access; there's no separate bootstrap step to install an agent first.

	ARCHITECTURE	GETTING A NEW HOST MANAGED
Ansible	Agentless — connects over SSH, runs, disconnects	Just needs SSH access + Python
Puppet / Chef	Agent-based — a persistent daemon runs on every node	Requires installing and registering the agent first

Push, Not Pull — A Real Contrast With Kubernetes and Prometheus

`obs1-3` described Prometheus's own pull model — the monitoring system reaches out on its own schedule. Ansible is the reverse: **push-based**. A human, or a CI pipeline, explicitly runs a playbook against a set of target hosts, at a moment they choose. Ansible doesn't sit in the background continuously reconciling state the way `k8s1-2`'s own reconciliation loop does — nothing changes on a managed host until a playbook is actually run against it. This is a genuine, important architectural difference between tools that all get grouped loosely under "infrastructure as code" — one worth keeping in mind now, since `ansible1-4` revisits it directly once idempotency is on the table.

"Fix the Config, Not Just the Console" — Delivering on cloud1-11's Own Lesson

`cloud1-11` named the real discipline behind avoiding configuration drift: change the source-controlled configuration, not the live server directly through a console or an ad-hoc SSH session that leaves no record and gets silently overwritten the next time anything reapplies the "real" config. Ansible is the concrete tool built to apply exactly that discipline to server configuration specifically — the piece Terraform, by its own design, never touches once a server already exists.

Where This Course Is Headed

Inventory, playbooks and modules, variables and Jinja2 templates, real idempotency testing, roles, Vault for secrets, dynamic inventory, and testing with Molecule — building, chapter by chapter, toward a genuine capstone: turning `ws1`'s own hand-run Debian web-server hardening course (HTTPS, UFW, fail2ban, SSH hardening, Apache security headers) into one real, idempotent, version-controlled playbook. The exact discipline this chapter just named, made concrete against content already on this site.

ANSIBLE SHIPS "BATTERIES INCLUDED"

A huge library of built-in modules covers practically every common task out of the box — package management, service control, file operations, user management, and modules for every major cloud provider's own API. Custom scripting is the exception, not the starting point.

AGENTLESS DOESN'T MEAN EFFORTLESS

"No agent to install" doesn't mean "zero setup" — it means a different, usually smaller kind of setup. SSH key management, a working network path to every target, and a Python interpreter available on the managed host are all real prerequisites Ansible still depends on, even without a persistent daemon of its own.

Hands-On Exercises

EXERCISE 1

A team already uses Terraform to provision their servers. Explain, in your own words, what Ansible would add to their pipeline that Terraform structurally cannot do on its own.

View solution

EXERCISE 2

Explain why a newly provisioned server can be managed by Ansible the moment it has SSH access, but would need an extra setup step before Puppet or Chef could manage it.

View solution

EXERCISE 3

Explain, using cloud1-11's own "fix the config, not just the console" lesson, why manually SSHing into a server to fix a problem is a worse long-term practice than fixing the same problem in an Ansible playbook, even though both approaches solve the immediate issue.

View solution

CHAPTER 1 QUICK REFERENCE

- **Terraform provisions** infrastructure; **Ansible configures** infrastructure that already exists — complementary, not competing
- **Agentless** — Ansible connects over SSH, runs, and leaves nothing persistent behind, unlike agent-based tools like Puppet/Chef
- **Push, not pull** — a playbook only changes anything when it's explicitly run, unlike Kubernetes' own continuous reconciliation loop or Prometheus's own scheduled scraping
- Ansible is the concrete tool behind cloud1-11's own "fix the config, not just the console" discipline
- This course builds toward a capstone automating ws1's own hand-run Debian hardening course into a real playbook
- Agentless still has real prerequisites — SSH access, network reachability, a Python interpreter on the target

CHAPTER 2 OF 10

Installing Ansible & Your First Playbook

Ansible

Chapter 2 · Installing Ansible & Your First Playbook

`ansible1-1` established that Ansible is agentless. This chapter makes that concrete — installing it, running your first commands, and writing the smallest playbook that actually does something.

The Control Node — Where Ansible Actually Runs

Ansible itself is installed on exactly one machine — the **control node** — not on every host it manages. This is `ansible1-1`'s agentless architecture made literal: there's no Ansible software running on a managed host at all, only on the control node that initiates outbound SSH connections. Installation is typically `pip install ansible`, and works natively on Linux and macOS; on Windows, the control node itself normally runs inside WSL — Ansible can still *manage* Windows hosts via WinRM, it just doesn't run natively as a control node on Windows.

Ad-Hoc Commands — Ansible Without a Playbook

```
ansible all -i inventory -m ping
ansible webservers -i inventory -a "systemctl status nginx"
```

`-m` specifies a **module** to run — `ping` here is a built-in connectivity-check module, worth distinguishing explicitly from ICMP ping: it confirms Ansible can connect over SSH and that Python is reachable and working on the target, not just that the host responds to a network ping. `-a` passes arguments to the default `command` module when no `-m` is given. Ad-hoc commands are genuinely useful for quick, one-off checks or actions — they don't give you repeatability or a record of what changed, which is exactly what a playbook is for.

Your First Playbook

```yaml
---
- name: Ensure nginx is installed and running
  hosts: webservers
  become: true
  tasks:
    - name: Install nginx
      apt:
        name: nginx
        state: present

    - name: Start nginx
      service:
        name: nginx
        state: started
```

A playbook is a list of **plays**; each play targets a group of hosts (from the inventory `ansible1-3` covers fully) and contains a list of **tasks**. Each task calls exactly one module — `apt`, `service` — with specific arguments describing the desired state, not a literal command to run. `become: true` is Ansible's privilege-escalation flag, the equivalent of running with `sudo`.

Running a Playbook

```
ansible-playbook -i inventory site.yml
```

Each task reports one of three outcomes per host — `ok` (already in the desired state, nothing to do), `changed` (something was actually modified), or `failed` — followed by a **PLAY RECAP** summarizing every host's own results at the end of the run.

ansible.cfg — Configuring Ansible's Own Defaults

```
[defaults]
inventory = ./inventory
remote_user = deploy
host_key_checking = False
```

`ansible.cfg` sets defaults — inventory path, default remote user, SSH behavior — so they don't need to be passed as flags on every single command. `host_key_checking = False` is a common development-convenience setting, worth flagging directly rather than copying blindly (see the warn-box below).

	GOOD FOR	REPEATABLE?
Ad-hoc commands	Quick one-off checks or actions	No — not recorded, not version-controlled
Playbooks	Anything meant to run more than once, or to be reviewed	Yes — a real, repeatable, version-controlled definition

TWO COMMANDS WORTH KNOWING IMMEDIATELY

`ansible --version` confirms the install and shows the config file actually in use. `ansible-doc <module>` shows full documentation and real usage examples for any built-in module, directly in the terminal — often faster than searching online once you know it exists.

HOST_KEY_CHECKING = FALSE IS A REAL SECURITY TRADEOFF, NOT A DEFAULT TO COPY BLINDLY

SSH host key checking protects against connecting to an unexpected host on first contact — a real, meaningful defense against a man-in-the-middle attack. Disabling it is genuinely convenient in a disposable lab or CI environment where hosts are recreated constantly, but it removes that protection outright. Worth a deliberate decision per environment, not a setting copy-pasted into every `ansible.cfg` out of habit.

Hands-On Exercises

EXERCISE 1

Write the ad-hoc command that checks whether every host in a group called "dbservers" is reachable over SSH with Python working, and explain what specifically the ping module confirms that a raw ICMP ping wouldn't.

View solution

EXERCISE 2

Write a playbook that ensures the package "curl" is installed and the service "cron" is started on hosts in the group "all", using the same structure as the chapter's own nginx example.

View solution

EXERCISE 3

A colleague sets host_key_checking = False in the ansible.cfg for a production environment "because it was annoying in testing." Explain what's actually being given up by this setting, and suggest a better approach for the testing environment specifically.

View solution

CHAPTER 2 QUICK REFERENCE

- Ansible installs only on the **control node** — never on managed hosts
- **Ad-hoc commands** (`ansible ... -m ... -a ...`) — quick, one-off, not repeatable
- A **playbook** is a list of plays; each play has hosts and a list of tasks, each task calling one module
- **ansible-playbook -i inventory site.yml** runs a playbook; results are ok/changed/failed per host, summarized in the PLAY RECAP
- **ansible.cfg** sets defaults (inventory, remote_user, SSH behavior) so flags aren't repeated every run
- `ansible-doc <module>` — full docs/examples for any module, without leaving the terminal
- host_key_checking = False is a real, deliberate security tradeoff — not a blanket default

CHAPTER 3 OF 10

Inventory — Telling Ansible What to Manage

Ansible

Chapter 3 · Inventory — Telling Ansible What to Manage

`ansible1-2` used `-i inventory` in every command without explaining what that file actually was. This chapter delivers — the inventory is Ansible's own list of hosts, and how it's organized shapes everything else in this course.

What an Inventory Actually Is

A plain-text file listing the hosts Ansible can manage, optionally organized into named **groups**. Two common formats: **INI-style** — simple, classic — and **YAML**, more structured and consistent with every other Ansible file this course writes.

INI-Style Inventory

```
[webservers]
web1.example.com
web2.example.com

[dbservers]
db1.example.com

[webservers:vars]
http_port=80
```

Bracketed headers define groups; hosts listed underneath belong to that group. The `:vars` suffix attaches variables to every host in that group at once — `http_port` here applies to both `web1` and `web2` without repeating it per host.

YAML-Style Inventory

```
all:
  children:
    webservers:
      hosts:
        web1.example.com:
        web2.example.com:
      vars:
        http_port: 80
    dbservers:
      hosts:
        db1.example.com:
```

The exact same inventory, expressed as YAML — more verbose for a trivial case like this one, but it scales far better once groups nest deeply or carry many variables, and it keeps every file in an Ansible project speaking the same syntax.

Groups of Groups — Nested Groups

```
[production:children]
webservers
dbservers
```

A group can itself be composed of other groups. `production` here contains every host in both `webservers` and `dbservers` — a play targeting `hosts: production` reaches all of them at once, without listing each individual group.

Host Variables and Group Variables

Inline `:vars` works, but `group_vars/` and `host_vars/` directories are the cleaner, more common approach at any real scale — one YAML file per host or group, automatically loaded by Ansible based on the filename matching the group or host name exactly:

```
inventory
group_vars/
  webservers.yml
host_vars/
  web1.example.com.yml
```

No explicit reference to these files is needed anywhere — Ansible loads `group_vars/webservers.yml` automatically for any host in the `webservers` group, purely by matching the filename.

Special Built-In Groups

- **all** — every host in the inventory, always implicitly available, no matter how groups are defined
- **ungrouped** — any host not explicitly placed into a named group

	READABILITY FOR SIMPLE CASES	SCALES TO DEEP NESTING
INI	Very readable, minimal	Gets awkward with many nested groups/vars
YAML	More verbose for a small inventory	Scales cleanly, consistent with playbooks

ANSIBLE-INVENTORY IS THE REAL DEBUGGING TOOL HERE

`ansible-inventory --list -i inventory` (or `--graph` for a tree view) shows exactly how Ansible parsed the inventory — every host, every group, every merged variable — genuinely the fastest way to answer "why isn't this variable applying the way I expect."

THE SAME VARIABLE DEFINED AT MULTIPLE LEVELS IS A COMMON SOURCE OF CONFUSION

A host can belong to several groups at once, and the same variable name can be set in `group_vars/all.yml`, a more specific group's own `group_vars` file, and a host's own `host_vars` file simultaneously. Which one actually wins is governed by Ansible's own variable precedence rules — `ansible1-5` covers this fully — but until then, a variable quietly taking an unexpected value is almost always this, not a bug.

Hands-On Exercises

EXERCISE 1

Write an INI-style inventory with a group "cacheservers" containing two hosts, cache1.example.com and cache2.example.com, with a group variable redis_port set to 6379.

View solution

EXERCISE 2

Write a nested group called "backend" containing both "webservers" and "dbservers" as children, using the INI :children syntax, and explain what hosts would be targeted by a play with hosts: backend.

View solution

EXERCISE 3

A host named web1.example.com is in the "webservers" group, and http_port is set to 80 in group_vars/webservers.yml but also set to 8080 in host_vars/web1.example.com.yml. Explain why this isn't necessarily a bug, and where you'd look to find out which value actually applies.

View solution

CHAPTER 3 QUICK REFERENCE

- Inventory lists hosts, organized into **groups** — INI-style (simple) or YAML (scales better, consistent with playbooks)
- `[group:vars]` (INI) or a `vars:` key (YAML) attaches variables to every host in a group
- `[group:children]` (INI) or nested `children:` (YAML) — a group made of other groups
- **group_vars/<name>.yml** and **host_vars/<name>.yml** — auto-loaded by filename match, the cleaner alternative to inline :vars
- **all** — every host, always implicit; **ungrouped** — hosts in no named group
- `ansible-inventory --list` / `--graph` — see exactly how Ansible parsed the inventory and merged variables
- The same variable set at multiple levels is resolved by precedence rules (ansible1-5), not a bug

CHAPTER 4 OF 10

Playbooks, Tasks & Modules In Depth

Ansible

Chapter 4 · Playbooks, Tasks & Modules In Depth

`ansible1-2` showed a minimal playbook without explaining what actually makes it safe to run more than once. This chapter is where that gets explained properly — modules as the real unit of work, and idempotency, the principle everything else in this course assumes.

Modules — The Atomic Unit of Work

Every task calls exactly one **module** — a small, self-contained unit of code Ansible pushes to the target, executes, and receives a structured result back from. `ansible1-1` 's own "batteries included" claim is concrete here: modules exist for package management (`apt` , `yum` , `dnf`), files (`copy` , `template` , `file`), services (`service` , `systemd`), users, and practically every major cloud provider's own API. Critically, a module describes **desired state** — `state: present` , `state: started` — not a literal command to execute. That single design choice is the foundation idempotency is built on.

Idempotency — The Central Design Principle

An **idempotent** operation produces the same end result no matter how many times it's applied. Run a playbook once, and the system converges toward the desired state. Run the exact same playbook again immediately afterward, and nothing should change — every task reports `ok` , not `changed` , because the state it describes already exists. Contrast this with a naive imperative script doing `echo "some line" >> config_file` — run twice, and the line appears twice. Modules like `lineinfile` / `blockinfile` exist specifically to make this kind of edit idempotent: they check the file's current content first, and only make a change if the desired line genuinely isn't already there.

Ansible's Idempotency vs. Kubernetes' Continuous Reconciliation

`ansible1-1` promised this contrast directly. `k8s1-2` 's own reconciliation loop continuously observes actual state and corrects drift automatically, all the time, without anyone triggering it. Ansible's idempotency is a genuinely different kind of guarantee: a playbook, when run, converges the target toward the desired state — but nothing happens between runs. If something drifts an hour after a playbook finishes, Ansible does nothing about it until that playbook is explicitly run again. Idempotent design makes *repeated* runs safe and

convergent; it does not provide *continuous* enforcement the way Kubernetes' reconciliation loop does. Neither is universally better — Kubernetes needs continuous self-healing for a live running system; Ansible is typically run on a schedule or on demand, for configuration management specifically.

Task Structure In Depth

```
- name: Install a list of packages
  apt:
    name: "{{ item }}"
    state: present
  loop:
    - curl
    - git
    - vim
```

`name` is a human-readable label shown in the run output — always name every task, so a real run's output actually tells you which step did what. `loop` runs the task once per item in a list, with `{{ item }}` substituting the current value each time. Other common task-level keywords: `when` (conditionally skip a task), `register` (capture a task's own result into a variable for a later task to use), and `tags` (run only a selected subset of tasks on a given invocation).

A Non-Idempotent Trap — the command/shell Modules

`command` and `shell` run arbitrary commands directly — a genuinely useful escape hatch when no dedicated module exists for a specific task. They are **not idempotent by default**, though: a shell command that unconditionally does something (appends to a file, restarts a service every time) reports `changed` on every single run, even when nothing meaningful actually needed to change. The fix is explicit: the `creates` / `removes` arguments tell Ansible to only run the command if a given file doesn't (or does) already exist, and `changed_when: false` lets a task explicitly declare its own changed status, restoring honest, idempotent-style reporting even while using this lower-level tool.

	IDEMPOTENT BY DEFAULT?	WHAT IT DESCRIBES
apt / service / file / ...	Yes	Desired state — Ansible figures out whether action is needed
command / shell	No — reports changed every run unless told otherwise	A literal command to execute, every time

CHECK MODE — SEE WHAT WOULD CHANGE, WITHOUT CHANGING ANYTHING

`ansible-playbook --check` runs a playbook in "dry run" mode — every task reports what it *would* do, without actually doing it. Genuinely useful for validating a change before it runs for real, especially given how much of this site's own general guidance leans toward reviewing risky operations before committing to them.

REACHING FOR COMMAND/SHELL WHEN A DEDICATED MODULE ALREADY EXISTS THROWS AWAY IDEMPOTENCY FOR NO REAL BENEFIT

Before writing a `command` / `shell` task, check whether a dedicated module already covers it — `ansible1-2` 's own `ansible-doc -l` lists every built-in module. A dedicated module gives you idempotency (and usually better portability across operating systems) for free; `command` / `shell` should be reserved for the genuine gaps, not used out of habit.

Hands-On Exercises

EXERCISE 1

Explain, using the echo/config-file example from this chapter, exactly why running a non-idempotent task twice produces a different result than running an idempotent module twice.

View solution

EXERCISE 2

A colleague asks why Ansible doesn't just run continuously in the background the way Kubernetes does, to catch configuration drift immediately instead of waiting for the next playbook run. Explain the real architectural reason, referencing this chapter's own comparison.

View solution

EXERCISE 3

A task uses shell: "systemctl restart myapp" with no changed_when or creates argument. Explain what problem this causes when the playbook is run repeatedly, and rewrite it (in words or YAML) so it only reports changed when the restart genuinely happened.

View solution

CHAPTER 4 QUICK REFERENCE

- A **module** is the atomic unit of work — it describes desired state, not a literal command

- **Idempotency** — running the same playbook twice makes no further changes the second time; the foundation this course is built on
- Ansible's idempotency (repeated runs converge safely) is different from Kubernetes' own continuous reconciliation loop (constant, automatic drift correction) — different guarantees for different problems
- **loop** (iterate), **register** (capture output), **when** (conditional), **tags** (selective execution) — common task-level keywords
- **command/shell** are not idempotent by default — use `creates` / `removes` or `changed_when` to fix that
- **ansible-playbook --check** — dry-run mode, shows what would change without changing anything
- Reach for a dedicated module before command/shell — idempotency and portability come free with the former

CHAPTER 5 OF 10

Variables, Facts & Jinja2 Templates

Ansible

Chapter 5 · Variables, Facts & Jinja2 Templates

`ansible1-3`'s own warn-box promised this chapter would cover variable precedence properly. It also used `{{ item }}` and `{{ variable | default(...) }}` without ever naming what that syntax actually is. Both get resolved here — along with the exact templating mechanism this course's own capstone uses to generate real config files.

Variable Precedence — Resolving Chapter 3's Own Promise

Ansible's real, official precedence order has over twenty distinct levels — more detail than is useful for getting started. The practical version most real playbooks actually need, lowest to highest:

1. Role defaults (`defaults/main.yml`) — the lowest priority, meant to be overridden
2. Inventory variables (`group_vars`, `host_vars`, from `ansible1-3`)
3. Playbook variables (a play's own `vars:` block)
4. Task-level variables / `set_fact`
5. **Extra vars** (`-e` on the command line) — always wins, no exceptions

A variable set at a higher level on this list always overrides the same variable set at a lower one. `-e` on the command line is the deliberate escape hatch for "just this one run, use this value, regardless of everything else" — and it always gets the final word.

Facts — Data Ansible Gathers For You

At the start of a play, Ansible automatically **gathers facts** about each managed host — operating system, IP addresses, memory, CPU count, hostname — unless explicitly disabled. These are available as `{{ ansible_facts.xxx }}`, or common ones via a shorthand like `{{ ansible_distribution }}` and `{{ ansible_distribution_version }}`.

```
- name: Only run on Debian-family hosts
  apt:
    name: ufw
    state: present
  when: ansible_os_family == "Debian"
```

Facts are what make a single playbook safely reusable across genuinely different hosts — the capstone's own target, Debian specifically, is exactly the kind of thing a `when:` condition on `ansible_os_family` would check for.

Jinja2 — Ansible's Templating Language

Every `{{ }}` used since `ansible1-2` — variable substitution in a task's arguments, `{{ item }}` in a loop — is **Jinja2**, not special Ansible-only syntax. Jinja2 also supports **filters**: `{{ variable | default('fallback') }}` provides a safe fallback value when a variable might not be defined, avoiding an "undefined variable" error outright — exactly the pattern used unexplained in `ansible1-4`'s own exercise solutions.

The template Module — Templating Real Config Files

`copy` transfers a file byte-for-byte, unchanged. `template` processes a file through Jinja2 *first* — substituting variables, evaluating conditionals and loops — then copies the resulting, fully rendered output.

```
# nginx.conf.j2
server {
    listen {{ http_port }};
    server_name {{ server_name }};
}
```

```
- name: Deploy nginx config from template
  template:
    src: nginx.conf.j2
    dest: /etc/nginx/sites-available/default
  notify: restart nginx
```

This is exactly the mechanism `ansible1-10`'s own capstone uses to generate real Apache vhost and security-header configuration for `ws1`'s own hardening course — a template file, rendered per-host, replacing what was originally typed by hand.

Conditionals and Loops Inside a Template

```
{% if ssl_enabled %}
listen 443 ssl;
{% endif %}

{% for header in security_headers %}
add_header {{ header.name }} "{{ header.value }}";
{% endfor %}
```

`{% if %}` / `{% endif %}` and `{% for %}` / `{% endfor %}` are Jinja2's own template-level conditionals and loops — distinct from the `when:` and `loop:` task keywords, which control whether/how a whole *task* runs, not what appears inside a rendered file. A list of security headers, each with its own name and value, rendered this way is exactly the shape of content `ws1`'s own security-headers chapter covered by hand — now generated.

MODULE	WHAT IT DOES	PROCESSES JINJA2?
copy	Transfers a file exactly as-is	No
template	Renders {{ }}/{% %} first, then transfers the result	Yes

THE DEFAULT FILTER IS A SAFE, COMMON PATTERN

`{{ my_var | default('fallback') }}` substitutes `'fallback'` if `my_var` is undefined, rather than failing the whole task. A small habit worth adopting anywhere a variable might legitimately be absent for some hosts but not others.

FACT GATHERING HAS A REAL COST ON LARGE INVENTORIES

Gathering facts means an extra SSH round trip and a real amount of data collected per host, every single play, by default — on a large inventory, that adds up. `gather_facts: false` set on a specific play is a genuine, worthwhile optimization when that play's own tasks don't actually need any fact data, not a setting to leave on everywhere purely out of habit.

Hands-On Exercises

EXERCISE 1

A variable app_port is set to 8080 in group_vars/webservers.yml and to 9090 via -e app_port=9090 on the command line. State which value wins, and explain why using this chapter's own precedence order.

View solution

EXERCISE 2

Write a task that installs the package "firewalld" only when ansible_os_family is "RedHat", using the when: keyword and a gathered fact.

View solution

EXERCISE 3

Write a small Jinja2 template snippet (as used inside a .j2 file) that renders a line "MaxConnections {{ max_connections }}" only if ssl_enabled is true, and explain why this has to be a {% if %} block rather than the task-level when: keyword.

View solution

CHAPTER 5 QUICK REFERENCE

- Practical variable precedence, low to high: role defaults → inventory vars → playbook vars → task vars/set_fact → **-e (always wins)**
- **Facts** — auto-gathered host data (ansible_os_family, ansible_distribution, ...), usable in when: conditions
- **Jinja2** is the templating language behind every {{ }} — including {{ item }} and the default filter
- **copy** transfers a file as-is; **template** renders Jinja2 first, then transfers the result
- {% if %}/{% for %} are Jinja2 template-level constructs, distinct from the task-level when:/loop: keywords
- template is the exact mechanism the capstone (ansible1-10) uses to generate ws1's own Apache/security-header config
- gather_facts: false is a real, worthwhile optimization when a play's tasks don't need fact data

CHAPTER 6 OF 10

Handlers & Real Idempotency Testing

Ansible

Chapter 6 · Handlers & Real Idempotency Testing

`ansible1-5` used `notify: restart nginx` without explaining it. This chapter closes that gap — and goes further, turning idempotency from something `ansible1-4` merely asserted into something you can actually prove, by running a playbook twice and reading the result honestly.

Handlers — Explaining notify:

```
handlers:
  - name: restart nginx
    service:
      name: nginx
      state: restarted
```

A **handler** is a special task that only runs if something `notify`-ed it, and runs at most once per play, no matter how many tasks notified it. `ansible1-5`'s own template task — `notify: restart nginx` — only actually triggers this handler when the template task itself reports `changed`. If the config file was already correct, the task reports `ok`, and the handler never fires at all. This is `ansible1-4`'s idempotency principle paying off directly: a service restart happens exactly when something genuinely changed, never as a matter of routine.

Why Handlers Run at the End

Handlers run **after every regular task in the play has completed**, not the moment they're notified. If five separate tasks all notify the same handler during one run, it still only fires once, at the very end — avoiding five redundant restarts when a single one is enough. For the rare case where an action genuinely needs to happen immediately, mid-play, rather than deferred — `meta: flush_handlers` exists as an explicit escape hatch, worth knowing exists even if it's rarely needed.

changed vs. ok — Reading Ansible's Own Report Honestly

`ansible1-2` already showed the report format — each task is `ok`, `changed`, or `failed` per host. Put together with `ansible1-4`'s idempotency principle, this gives a concrete, checkable claim: a healthy second run of an idempotent playbook should report **changed=0** in the PLAY RECAP. Not "probably nothing changed" — a specific, verifiable number.

Real Idempotency Testing — Running a Playbook Twice

```
# First run, against a fresh host
PLAY RECAP *****
web1.example.com          : ok=6    changed=4    unreachable=0    failed=0

# Second run, immediately after, no changes made in between
PLAY RECAP *****
web1.example.com          : ok=6    changed=0    unreachable=0    failed=0
```

This is the actual practice, not just a claim to trust: run the playbook once, expect real changes on a fresh host. Run it again, immediately, with nothing else touched. If the second run reports anything other than `changed=0`, that's a genuine bug somewhere in the playbook — most often `ansible1-4`'s own `command` / `shell` trap, a task quietly doing something every single run regardless of whether it needed to.

A Concrete Idempotency Bug, Caught by This Test

Recall `ansible1-4`'s own unfixed example — `shell: "systemctl restart myapp"`, no `changed_when`. Before the fix, running this idempotency test would show `changed=1` on **every single run**, second run included, no matter how many times it's repeated — the exact symptom that reveals the bug. After the fix (a `when:` / `changed_when:` tied to whether the config genuinely changed), a clean second run correctly reports `changed=0`. The fix from that chapter and the verification practice from this one are the same discipline, applied at two different points.

SECOND-RUN RESULT	WHAT IT MEANS
changed=0	The playbook is genuinely idempotent — nothing left to converge
changed>0, identical to the first run	A real idempotency bug — something is running unconditionally every time

--CHECK PLUS --DIFF SHOWS EXACTLY WHAT WOULD CHANGE

`ansible1-4`'s own `--check` shows *that* a task would report changed; adding `--diff` shows the actual line-by-line difference for file-based tasks like `template`. Genuinely useful when a second run reports an unexpected change and you need to see precisely what's different, not just that something is.

A NOTIFY INSIDE A SKIPPED TASK NEVER FIRES — A REAL, EASY-TO-MISS GOTCHA

If a task carrying `notify: restart nginx` is itself skipped by a `when:` condition, the handler is never triggered at all — the `notify` line is still physically present in the file and looks like it should apply, but a skipped task notifies nothing. Worth checking directly when a service unexpectedly fails to restart after a config change that should have triggered it.

Hands-On Exercises

EXERCISE 1

Three separate tasks in one play each notify the same handler, "restart myapp." Explain exactly how many times the handler actually runs, and when, during that one playbook run.

View solution

EXERCISE 2

A playbook's second run reports changed=2 in the PLAY RECAP, identical to the first run's result for those same two tasks. Explain what this tells you, and the specific next step you'd take to investigate.

View solution

EXERCISE 3

A task that copies a config file and notifies a "restart myapp" handler has when: feature_flag_enabled attached to it. On a host where feature_flag_enabled is false, explain what happens to both the task and the handler.

View solution

CHAPTER 6 QUICK REFERENCE

- A **handler** only runs if notified, at most once, at the **end** of the play — regardless of how many tasks notify it
- A task only notifies its handler when that task itself reports **changed** — ansible1-4's idempotency principle, made practical
- A healthy, idempotent playbook reports **changed=0** on an immediate second run — a concrete, checkable claim, not a hope
- Running a playbook twice and checking for changed=0 is the real idempotency test — not just trusting a module's own reputation
- --check --diff shows exactly what would change, line by line, for file-based tasks

- A notify inside a task skipped by when: never fires — a real, easy-to-miss cause of "why didn't this restart"

CHAPTER 7 OF 10

Roles — Organizing Playbooks for Reuse

Ansible

Chapter 7 · Roles — Organizing Playbooks for Reuse

Everything so far — tasks, handlers, variables, templates — has lived in one or two files. That works for a small example; it doesn't scale. This chapter is Ansible's real answer: **roles**, a standard way to package each concern into its own self-contained, reusable unit.

The Problem — One Big Playbook Doesn't Scale

As a playbook grows — more tasks, more handlers, more variables, more templates, all covering genuinely different concerns — it becomes hard to reuse any single piece elsewhere, and hard to find anything in a file that keeps growing. Roles solve this by giving each concern its own directory, structured the same predictable way every time.

The Role Directory Structure

```
roles/
  nginx/
    tasks/
      main.yml
    handlers/
      main.yml
    templates/
      nginx.conf.j2
    vars/
      main.yml
    defaults/
      main.yml
    files/
    meta/
      main.yml
```

- **tasks/main.yml** — the role's own task list, loaded automatically by convention, no explicit reference needed anywhere
- **handlers/main.yml** — handlers scoped to this role, working exactly as `ansible1-6` described

- **templates/** — `.j2` files, referenced by relative filename from the role's own tasks
- **defaults/main.yml** — deliberately the *lowest*-precedence variables per `ansible1-5`'s own order, meant to be overridden by whoever uses the role
- **vars/main.yml** — higher-precedence variables, meant to stay more fixed, internal to the role's own logic
- **files/** — static files for a plain `copy` task
- **meta/main.yml** — role metadata, including dependencies on other roles

Using a Role in a Playbook

```
- hosts: webservers
  roles:
    - nginx
    - firewall
```

Every task in every listed role runs, in order, before any play-level tasks. For finer control over ordering — interleaving a role's tasks with ordinary tasks at a specific point — `include_role` does the same job from inside a normal `tasks:` list:

```
- hosts: webservers
  tasks:
    - include_role:
        name: nginx
```

Ansible Galaxy — A Registry of Shared Roles

`ansible-galaxy install <role>` pulls a pre-built, community-maintained role from the public Galaxy registry — the same "why reinvent this" idea as `tf1-7`'s own public Terraform Registry. `ansible-galaxy init <rolename>` scaffolds the entire directory structure above automatically, so the exact folder names never need to be memorized or typed by hand.

DRY Configuration — Why This Actually Matters Here

Without roles, the capstone's own real target — `ws1`'s HTTPS/Let's Encrypt setup, UFW, fail2ban, SSH hardening, and Apache security headers — would all have to live crammed into one sprawling playbook. With roles, each of those concerns becomes its own independently testable, independently reusable unit. This directly foreshadows how `ansible1-10`'s own capstone is actually structured: one role per `ws1` concern, not one giant file.

	PRECEDENCE (ANSIBLE1-5)	INTENDED USE
defaults/main.yml	Lowest — meant to be overridden	Sensible defaults, freely customizable by whoever uses the role
vars/main.yml	Higher — closer to fixed	Internal values the role's own logic depends on, not meant for casual overriding

ANSIBLE-GALAXY INIT SCAFFOLDS THE WHOLE TREE

Running `ansible-galaxy init nginx` creates every subdirectory shown above, empty and ready to fill in — no need to remember the exact folder names or create them by hand one at a time.

A TYPO'D DIRECTORY NAME FAILS SILENTLY, NOT LOUDLY

Role tasks are auto-loaded purely by convention — there's no explicit "here's where the tasks live" line anywhere in a playbook. A typo like `task/` instead of `tasks/` doesn't raise an error; Ansible simply finds nothing there and treats the role as if it has no tasks at all, silently doing nothing. Worth checking directory spelling directly when a role appears to run successfully but does nothing.

Hands-On Exercises

EXERCISE 1

Sketch the directory structure (folder names only, matching this chapter's own layout) for a new role called "fail2ban" that needs its own tasks, handlers, and one default variable.

View solution

EXERCISE 2

Explain why a role variable meant to let users customize the SSH port should live in defaults/main.yml rather than vars/main.yml, referencing this chapter's own precedence table.

View solution

EXERCISE 3

A role named "ufw" runs successfully with no errors, but none of its intended firewall rules are ever applied. Using this chapter's own warn-box, propose the most likely explanation and how to confirm it.

View solution

CHAPTER 7 QUICK REFERENCE

- A **role** packages tasks/handlers/templates/vars/files for one concern into a standard directory structure
- **tasks/main.yml** auto-loads by convention; **defaults/main.yml** is deliberately lowest precedence, **vars/main.yml** higher
- `roles:` at the play level runs all listed roles first; `include_role` interleaves role tasks anywhere in a task list
- **Ansible Galaxy** — a public registry of shared roles, the same idea as Terraform's own Registry
- `ansible-galaxy init <name>` scaffolds the whole directory tree automatically
- A typo'd directory name (task/ vs tasks/) fails silently — the role just does nothing, with no error
- The capstone (ansible1-10) is structured as one role per ws1 concern — this chapter's own DRY payoff, made concrete

CHAPTER 8 OF 10

Ansible Vault — Secrets Management

Ansible

Chapter 8 · Ansible Vault — Secrets Management

`ansible1-7` gave every concern its own tidy, version-controlled home. But roles and playbooks are naturally full of exactly the values `pipelines1-5` and `crypto1-11` both already warned about — database passwords, API keys, private keys. This chapter is the mechanism that lets those values live in version control safely, following the rule those earlier chapters already established.

The Problem — Roles and Playbooks Are Full of Secrets

Database passwords, API keys, TLS private keys — all naturally want to live as ordinary Ansible variables, in exactly the same files `ansible1-7` just organized. But `pipelines1-5`'s own rule stands regardless of which tool is committing the file: a credential committed to git in plaintext is compromised forever, even if it's later removed, since git history retains it. **Ansible Vault** is the tool for encrypting exactly these values, so they can live safely in version control alongside everything else — encrypted, not absent.

Encrypting a Whole File

```
ansible-vault create secrets.yml    # create + open a new encrypted file
ansible-vault encrypt vars.yml      # encrypt an existing plaintext file in place
ansible-vault edit secrets.yml      # decrypt, open in $EDITOR, re-encrypt on save
ansible-vault view secrets.yml      # read-only decrypt to stdout
```

`edit` never writes the plaintext to disk at any point during the process — it decrypts into memory, opens your editor, and re-encrypts on save. An encrypted file on disk looks genuinely unreadable:

```
$ANSIBLE_VAULT;1.1;AES256
6638643965396563396637 3...
(many more lines of ciphertext)
```

Safe to commit, safe to diff (though the diff itself is meaningless without the password), and structurally no different from any other file in the repository.

Encrypting a Single Variable — encrypt_string

```
ansible-vault encrypt_string 'supersecret' --name 'db_password'
```

```
db_password: !vault |
          $ANSIBLE_VAULT;1.1;AES256
          6638643965396365...
```

This produces a small, self-contained encrypted block that can be pasted directly into an otherwise-plaintext vars file. Often preferable to encrypting the whole file: the rest of the file — variable names, structure, non-sensitive values — stays readable in a normal diff, with only the genuinely sensitive value itself hidden.

Providing the Vault Password to Run a Playbook

```
ansible-playbook site.yml --ask-vault-pass
ansible-playbook site.yml --vault-password-file ~/.vault_pass.txt
```

`--ask-vault-pass` prompts interactively; `--vault-password-file` points at a file (or an executable script that outputs the password) — the practical option for CI, where nothing can be typed interactively. Running a playbook that references encrypted variables without supplying the password fails cleanly, with a clear error — not a silent, wrong value quietly used instead.

Vault IDs — Multiple Passwords for Multiple Environments

```
ansible-vault encrypt --vault-id dev@prompt dev_secrets.yml
ansible-vault encrypt --vault-id prod@~/.vault_pass_prod prod_secrets.yml
```

A light touch, worth naming: **vault IDs** let genuinely different environments use genuinely different vault passwords. Someone who only has the `dev` password structurally cannot decrypt production secrets — a real, meaningful security boundary between environments, not just a naming convention.

Directly Extending pipelines1-5/crypto1-11's Own Rule

`pipelines1-5` named "a committed credential is compromised forever" as the reason never to commit secrets in plaintext; `crypto1-11` covered key management practice more broadly. Ansible Vault is the concrete mechanism that lets Ansible-managed secrets follow that exact rule while still living in version

control the same way everything else in this course does — `ansible1-7`'s own roles, and the same discipline `tf1`'s own state-file material already applied to Terraform's own sensitive values.

	DIFF READABILITY	GRANULARITY
Encrypt whole file	Diff is opaque ciphertext, no visible structure	All-or-nothing — every value in the file is hidden
encrypt_string	Surrounding structure/keys stay readable in diffs	Per-value — only the sensitive value itself is hidden

NEVER STORE THE VAULT PASSWORD ANYWHERE NEAR THE REPO IT PROTECTS

A password manager, a dedicated secrets system, or — for CI specifically — a protected pipeline secret injected at run time, exactly the CI-secrets pattern `pipelines1-5` already covered. The vault password sitting in the same repository (or worse, the same commit history) as the encrypted data it protects defeats the entire point.

LOSING THE VAULT PASSWORD MEANS LOSING THE SECRETS, PERMANENTLY

There's no backdoor, no "forgot password" reset — Ansible Vault's own encryption is genuinely strong, which means losing the password is equivalent to losing the underlying secrets outright. The password's own safekeeping matters exactly as much as the encryption itself.

Hands-On Exercises

EXERCISE 1

Write the command that would encrypt an existing plaintext file named db_vars.yml in place, and explain what the file looks like on disk immediately afterward.

View solution

EXERCISE 2

A vars file has ten ordinary configuration variables and one API key. Explain, using this chapter's own compare-table, why encrypt_string would likely be the better choice here over encrypting the whole file.

View solution

EXERCISE 3

A CI pipeline needs to run a playbook that references vault-encrypted variables, with no human available to type a password interactively. Explain which vault option is appropriate here, and where the password itself should actually be stored.

View solution

CHAPTER 8 QUICK REFERENCE

- **ansible-vault create/encrypt/edit/view** — encrypt/decrypt whole files; edit never writes plaintext to disk
- **encrypt_string** — encrypt one variable inline, keeping the rest of a file's diff readable
- **--ask-vault-pass** (interactive) vs. **--vault-password-file** (CI-friendly) — how a playbook gets the password to decrypt
- **Vault IDs** let different environments use genuinely different passwords — a real security boundary, not just a label
- Ansible Vault is the concrete mechanism letting secrets follow pipelines1-5/crypto1-11's own never-commit-plaintext rule while still living in version control
- Never store the vault password near the repo it protects — a password manager or a protected CI secret, never the same commit history
- Losing the vault password permanently loses the secrets — there is no recovery mechanism

CHAPTER 9 OF 10

Dynamic Inventory & Testing with Molecule

Ansible

Chapter 9 · Dynamic Inventory & Testing with Molecule

Two remaining practical topics before the capstone: keeping an inventory accurate automatically, and actually testing a role rather than trusting it by inspection. Both echo themes this course has already built toward — `obs1-3`/`obs1-10`'s own static-vs-dynamic discovery material, applied here to hosts, and `ansible1-6`'s own manual idempotency-testing habit, automated.

The Problem With Static Inventory at Real Scale

`ansible1-3`'s static INI/YAML inventory is fine for a small, stable set of hosts. It breaks down the exact same way `obs1-3`'s own static Prometheus scrape configuration did: cloud instances autoscale, get replaced, get new IPs — a hand-maintained inventory file quietly goes stale, listing hosts that no longer exist and missing ones that do.

Dynamic Inventory — Querying the Source of Truth Directly

```
ansible-playbook -i aws_ec2.yml site.yml
```

A **dynamic inventory** is a plugin that queries a live source — a cloud provider's API, a Kubernetes cluster — at run time, generating the inventory on the fly instead of reading a static file. `aws_ec2.yml` above isn't a host list; it's a plugin configuration:

```
plugin: amazon.aws.aws_ec2
regions:
  - us-east-1
filters:
  tag:Environment: production
keyed_groups:
  - key: tags.Role
    prefix: role
```

This generates groups automatically *from* the cloud provider's own tags — any EC2 instance tagged `Role=webserver` lands in a `role_webserver` group automatically, no one maintaining that membership by

hand. This is genuinely the same idea as `obs1-10`'s own ServiceMonitor label-matching — declare what to select, let the platform's own live state populate the result — just applied to hosts instead of scrape targets.

Kubernetes as a Dynamic Inventory Source

The `kubernetes.core.k8s` inventory plugin generates an inventory directly from pods or services running in a cluster, pairing directly with `obs1-10`'s own Kubernetes material — the same "don't hand-maintain a list of things that are inherently ephemeral" lesson, applied one layer further.

Testing Ansible with Molecule

`ft1-1`'s own principle — test behavior, not implementation — applies here too. **Molecule** is the standard tool for testing Ansible roles: it spins up an isolated test instance (commonly via Docker), applies the role, and runs verification checks, all automatically, with no real staging server required.

```
molecule init role my_role
molecule test
```

`molecule init role` scaffolds a testable role with its own `molecule/` directory — `converge.yml` applies the role under test, `verify.yml` checks the result. `molecule test` runs the full cycle automatically: create a test instance → converge (apply the role) → an idempotence check → verify → destroy.

Molecule's Idempotence Check — Automating Chapter 6's Own Practice

`ansible1-6` taught running a playbook twice by hand and checking for `changed=0`. Molecule's own **idempotence** step in `molecule test` does exactly this, automatically — applying the role a second time and failing the test outright if anything reports `changed`. The same discipline from that chapter, now a real, repeatable, CI-runnable check rather than a manual habit someone has to remember to perform.

	ACCURACY OVER TIME	SETUP EFFORT
Static inventory	Goes stale as hosts scale/change	Minimal — a plain file
Dynamic inventory	Always reflects the live platform's own current state	Requires plugin config, API credentials

MOLECULE TEST BELONGS IN CI

Exactly the kind of check that fits `tf1-10`'s and `pipelines1`'s own CI material — running `molecule test` automatically on every role change catches a broken or non-idempotent role before it's ever applied to a real host, the same "catch it before it ships" discipline this course has already applied elsewhere.

A DYNAMIC INVENTORY QUERY IS A REAL NETWORK DEPENDENCY, NOT A FREE ABSTRACTION

Every run against a live API has real cost and latency, and depends on that API being reachable and correctly authenticated — a genuine new failure mode static inventory never had (a plain file can't have an API outage). Worth being aware of, and caching where appropriate, rather than assuming a dynamic inventory query is free.

Hands-On Exercises

EXERCISE 1

Explain why a static inventory file listing EC2 instances by IP address would become inaccurate within days on a team using autoscaling groups, and how a dynamic inventory plugin avoids that problem.

View solution

EXERCISE 2

Explain what molecule test's idempotence step is actually doing, and how it relates directly to the manual "run it twice, check changed=0" practice from ansible1-6.

View solution

EXERCISE 3

A team's dynamic inventory query starts failing intermittently because the cloud API it depends on occasionally times out. Explain why this is a genuinely new category of failure compared to a static inventory file, and one mitigation worth considering.

View solution

CHAPTER 9 QUICK REFERENCE

- Static inventory goes stale as hosts autoscale/get replaced — the same problem obs1-3's own static scrape config had
- **Dynamic inventory** plugins query a live source (cloud API, Kubernetes) at run time, generating groups automatically from tags/labels
- The `kubernetes.core.k8s` plugin applies the same idea directly to a Kubernetes cluster, pairing with obs1-10
- **Molecule** tests roles in isolation — create, converge, idempotence check, verify, destroy — automatically
- Molecule's own idempotence step automates ansible1-6's own manual "run twice, expect changed=0" practice
- molecule test belongs in CI — catching a broken role before it ever touches a real host

- A dynamic inventory query is a real network dependency — it can fail in ways a static file never could

CHAPTER 10 OF 10

Capstone: Automating ws1's Web Server Hardening

Ansible

Chapter 10 · Capstone — Automating ws1's Web Server Hardening

The final chapter. `ws1`'s own web-server hardening course was built entirely by hand — typing commands over SSH, editing files directly. This capstone turns five of its eight chapters into one real, idempotent, version-controlled Ansible project — the exact "fix the config, not just the console" discipline `ansible1-1` opened this whole course with, finally made concrete.

The Target — ws1's Own Hardening Course, Automated

Five of `ws1`'s eight chapters, each becoming its own role, per `ansible1-7`'s own DRY principle: HTTPS via Let's Encrypt, firewall via UFW, fail2ban, SSH hardening, and Apache security headers.

```
roles/
  firewall/
  ssh_hardening/
  https/
  fail2ban/
  security_headers/
```

```
- name: Harden the web server
  hosts: webservers
  become: true
  roles:
    - firewall
    - ssh_hardening
    - https
    - fail2ban
    - security_headers
```

Role 1 — firewall (UFW)

```
  - name: Allow SSH
    community.general.ufw:
      rule: allow
      port: "22"
      proto: tcp

  - name: Allow HTTPS
    community.general.ufw:
      rule: allow
      port: "443"
      proto: tcp

  - name: Enable UFW, deny by default
    community.general.ufw:
      state: enabled
      policy: deny
```

Role 2 — ssh_hardening

```
  - name: Disable root login
    lineinfile:
      path: /etc/ssh/sshd_config
      regexp: '^#?PermitRootLogin'
      line: 'PermitRootLogin no'
    notify: restart sshd
```

`lineinfile` — `ansible1-4`'s own example of a genuinely idempotent edit, checking the file's current content before deciding whether to change anything.

Role 3 — https (Let's Encrypt)

```
  - name: Obtain certificate via certbot
    command: certbot --apache -d "{{ domain_name }}" --non-interactive --agree-tos -m "{{ admin_email }}
    args:
      creates: "/etc/letsencrypt/live/{{ domain_name }}/fullchain.pem"
```

`certbot` is invoked via `command`, since no dedicated module covers this exact flow — but the `creates` argument is `ansible1-4`'s own fix for `command`/`shell`'s non-idempotency trap, applied for real: the task only actually runs if the certificate doesn't already exist.

Role 4 — fail2ban

```
- name: Deploy fail2ban jail config
  template:
    src: jail.local.j2
    dest: /etc/fail2ban/jail.local
  notify: restart fail2ban
```

`ansible1-5`'s own `template` module, doing exactly what it was introduced to do — rendering a real config file per host, with `ansible1-6`'s own handler restarting the service only when that file genuinely changed.

Role 5 — security_headers (Apache)

```
# security_headers.conf.j2
{% for header in security_headers %}
Header always set {{ header.name }} "{{ header.value }}"
{% endfor %}
```

```
security_headers:
  - { name: "X-Frame-Options", value: "SAMEORIGIN" }
  - { name: "X-Content-Type-Options", value: "nosniff" }
```

`ansible1-5`'s own `{% for %}` loop example, no longer a teaching snippet — this is the actual mechanism generating `ws1`'s own Apache security-header configuration, driven by a plain, editable list of headers rather than typed by hand into a config file.

Secrets — Vault-Protecting admin_email and Anything Sensitive

```
admin_email: !vault |
          $ANSIBLE_VAULT;1.1;AES256
          3662386163...
```

`ansible1-8`'s own `encrypt_string`, applied to `admin_email` (and any API token a real deployment might also need) — the rest of the vars file stays fully readable, only the sensitive value itself is hidden.

Verifying Idempotency — Running the Whole Playbook Twice

```
# First run
web1.example.com : ok=15  changed=9   unreachable=0  failed=0

# Second run, immediately after
web1.example.com : ok=15  changed=0   unreachable=0  failed=0
```

`ansible1-6`'s own practice, applied to the whole capstone: `changed=0` on the second run is the actual proof this playbook is safe to re-run on a schedule, as a real, ongoing defense against configuration drift — not a claim taken on faith.

PIECE	CHAPTER
Role structure, one per ws1 concern	**ansible1-7**
lineinfile idempotent edit	**ansible1-4**
command + creates for certbot	**ansible1-4**
template + Jinja2 for-loop	**ansible1-5**
notify/handlers, restart-on-change	**ansible1-6**
Vault-encrypted admin_email	**ansible1-8**
Second-run idempotency verification	**ansible1-6**
Provisioning/configuring division of labor	**ansible1-1, tf1-1**

STILL OUT OF SCOPE, HONESTLY

This capstone deliberately covers five of `ws1`'s eight chapters — ClamAV and Monitoring & Maintenance are left out, to keep the capstone focused rather than exhaustive. No Molecule tests were written for these specific roles (`ansible1-9` covered the technique; applying it here is a genuine next step, not done in this chapter). This capstone uses a plain, static inventory — a deliberate choice for a small, fixed set of known servers, not `ansible1-9`'s own dynamic inventory, which solves a different problem. And every role here is Debian-specific, matching `ws1`'s own scope exactly — none of it is portable to a RedHat-family host without real additional `when:` conditions.

Hands-On Exercises

EXERCISE 1

Write the fail2ban role's own task (using the template module, matching the chapter's own pattern) that deploys jail.local and notifies a handler named "restart fail2ban" only when the file's content actually changes.

View solution

EXERCISE 2

Explain why the certbot task uses command with a creates argument instead of running unconditionally on every playbook run, referencing ansible1-4's own idempotency material directly.

View solution

EXERCISE 3

A colleague asks why admin_email needs Vault protection when it's "just an email address, not really a secret." Give an honest answer, and name one genuinely sensitive value this capstone's own roles would need to protect the same way in a real deployment.

View solution

CHAPTER 10 QUICK REFERENCE — ANSIBLE COMPLETE

- Five ws1 chapters (HTTPS, UFW, fail2ban, SSH hardening, Apache security headers) become five roles, one playbook
- lineinfile and command+creates deliver ansible1-4's own idempotent-edit and idempotent-escape-hatch patterns for real
- template + Jinja2 {% for %} generates real Apache config from a plain list of headers, not hand-typed text
- notify/handlers restart services only when something genuinely changed — ansible1-6's own principle, applied
- Vault protects admin_email and any real secrets, staying fully readable elsewhere in the vars file
- changed=0 on a second run is the actual, checkable proof this playbook is safe to re-run on a schedule
- Honest scope note: ClamAV/Monitoring left out, no Molecule tests written here, static (not dynamic) inventory, Debian-only
- **The full Ansible course — 10 chapters — is now complete.**