

INF 345 — Fundamentals of DevOps

Lecture 7: Configuration Management & Ansible Basics

Adil Akhmetov · Lesson 7



Recap — Lesson 6 (Container Security)

- Why is `ENV API_TOKEN=...` in a Containerfile a leak? (anyone who can pull the image can read it with `docker inspect` / `docker history`)
- What does `--cap-drop ALL --read-only` buy you? (a compromised process can't use root's powers or change the image's files)
- Why pin `golang:1.23-alpine` instead of `golang:latest`? (`latest` moves — the same Containerfile builds a different image next week)

Today's agenda

- ☐ Why configuration management exists: drift and snowflake servers
- ☐ Declarative vs imperative — and what idempotent really means
- ☐ Ansible's model: control node, inventory, modules, SSH
- ☐ Ad-hoc commands
- ☐ Playbooks: plays, tasks, variables, templates, handlers
- ☐ Dry runs and debugging
- ☐ Straight into today's practice: your first playbook

The scenario

You run 30 servers. Each was set up by hand over SSH, by different people, over two years. A security fix means changing one line in a config file on all of them. Server 12 has a different version of the file. Server 19 has a manual hotfix nobody wrote down. Server 27 was rebuilt last month and is missing three packages.

Containers solved this for the app. Today: the machines underneath — and everything that isn't a container.

Block 1

Why configuration management

Drift and snowflakes

- **Configuration drift:** servers that started identical slowly diverge — one-off fixes, manual installs, forgotten experiments.
- **Snowflake server:** unique, undocumented, impossible to rebuild. Nobody dares to touch it.
- The fix is the same idea as a Containerfile: the server's desired state is **written down, in git**, and a tool makes reality match it.
- That's **configuration management** — and with everything in git, it's **infrastructure as code**.

Rule of thumb: if you SSH in and change something by hand, it isn't done until it's in the playbook.

Imperative vs declarative

Imperative — *how*: a list of commands

```
apt-get install -y nginx
echo "server_tokens off;" >> /etc/nginx/nginx.conf
systemctl restart nginx
```

Run it twice → the line is appended twice, nginx restarts for nothing.

Declarative — *what*: the end state

```
- ansible.builtin.apt:
  name: nginx
  state: present
- ansible.builtin.lineinfile:
  path: /etc/nginx/nginx.conf
  line: "server_tokens off;"
```

Run it twice → the second run sees it's already true and does nothing.

Idempotent: running it once or ten times gives the same result. That's what makes it safe to run against 30 servers you're not sure about.

The tools

Tool	Model	Agent on servers?	Language
Ansible	push, over SSH	no	YAML
Puppet	pull, from a server	yes	Puppet DSL
Chef	pull	yes	Ruby
Salt	push or pull	usually	YAML

- Ansible needs only **SSH and Python** on the managed machine — nothing to install, nothing running between runs.
- It's what Red Hat ships for RHEL automation — **RH294, your Lab 02**.

Block 2

How Ansible works

Control node, managed nodes

- **Control node:** where you run `ansible` — your laptop, a CI runner. Linux or macOS (Windows → WSL).
- **Managed nodes:** the servers. Need SSH + Python, nothing else.
- **Inventory:** the list of managed nodes, in groups.
- **Modules:** units of work — `apt` , `copy` , `template` , `service` , `user` ... Ansible copies a module to the node over SSH, runs it, reads the JSON result, deletes it.
- **Playbook:** YAML that says which modules to run on which hosts.

```
pipx install ansible-core      # or: pip install --user ansible-core
ansible --version
```

Inventory

inventory.ini

```
[web]
web1.example.com
web2.example.com ansible_port=2222

[db]
db1.example.com

[prod:children]
web
db

[web:vars]
http_port=8080
```

- **Groups** in `[brackets]` ; a host can be in many.
- `:children` — groups of groups.
- Variables per host (same line) or per group (`[group:vars]`).
- Special variables change *how* Ansible connects: `ansible_host` , `ansible_user` , `ansible_port` , `ansible_connection=local` (no SSH at all).
- `all` always exists.

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

Ad-hoc commands

```
ansible all -m ansible.builtin.ping                # can I reach them all?
ansible web -m ansible.builtin.setup -a "filter=ansible_distribution*" # facts
ansible web -m ansible.builtin apt -a "name=nginx state=present" --become
ansible db -m ansible.builtin.command -a "uptime"    # read-only one-off
```

- `-m` module, `-a` its arguments, `--become` = sudo.
- `ping` isn't ICMP: it logs in over SSH and checks Python works.
- Good for questions ("which kernel is everyone on?") and emergencies.
- Anything you'll need twice belongs in a **playbook**.

Quick check

`ansible all -m ping` returns *"Failed to connect to the host via ssh"* for `localhost`. You're running Ansible on that same machine. What's missing?

`ansible_connection=local` — without it, Ansible uses SSH for every host, even localhost.

Block 3

Playbooks

Anatomy of a playbook

```
- name: Web servers          # a PLAY: which hosts, what to do
  hosts: web
  become: true               # run tasks with sudo

  vars:
    http_port: 8080

  tasks:                    # TASKS run top to bottom, on every host in parallel
    - name: Install nginx
      ansible.builtin.apt:  # the module (fully qualified name)
        name: nginx
        state: present     # declarative: "present", not "install"

    - name: Start and enable nginx
      ansible.builtin.service:
        name: nginx
        state: started
        enabled: true
```

Running it

```
$ ansible-playbook site.yml

TASK [Install nginx] *****
changed: [web1]
ok: [web2]                      ← already installed – nothing to do

PLAY RECAP *****
web1 : ok=3  changed=2  unreachable=0  failed=0
web2 : ok=3  changed=0  unreachable=0  failed=0
```

- **ok** = already in the desired state. **changed** = Ansible changed it.
- Run it again: everything is `ok` , `changed=0` . That's idempotence you can see.
- A host that fails stops; the others carry on.

Modules, not shell

```
# x always "changed", no idea what happened  
- ansible.builtin.shell: mkdir -p /opt/app
```

```
# ✓ checks first, changes only if needed  
- ansible.builtin.file:  
  path: /opt/app  
  state: directory  
  mode: "0755"
```

- `shell` / `command` run anything — so Ansible can't know if it changed something. It reports `changed` every run.
- A module knows its own state: directory exists? right mode? → `ok` .
- The everyday set: `file` , `copy` , `template` , `lineinfile` , `apt` / `dnf` , `service` , `user` .
- `command` is fine for read-only checks — never for changing state.

Variables and templates

templates/nginx.conf.j2 (Jinja2)

```
server {  
    listen {{ http_port }};  
    server_name {{ inventory_hostname }};  
}
```

```
- ansible.builtin.template:  
  src: templates/nginx.conf.j2  
  dest: /etc/nginx/conf.d/app.conf
```

- `{{ var }}` is filled in from play `vars:`, inventory, facts...
- One template, every server gets its own rendered file.
- **Precedence:** extra vars on the command line win over everything: `ansible-playbook site.yml -e http_port=9090`.
- Never hardcode a value you might change — make it a variable.

Handlers: restart only when needed

```
tasks:
  - name: Nginx config
    ansible.builtin.template:
      src: templates/nginx.conf.j2
      dest: /etc/nginx/conf.d/app.conf
    notify: Restart nginx

handlers:
  - name: Restart nginx
    ansible.builtin.service:
      name: nginx
      state: restarted
```

The handler runs **once, at the end of the play**, and **only if** a task that notifies it reported `changed`. Config unchanged → no restart → no dropped connections.

Before you run it for real

```
ansible-playbook site.yml --syntax-check # YAML and structure
ansible-playbook site.yml --check --diff # dry run: what WOULD change, as a diff
ansible-playbook site.yml --limit web1   # one host first
ansible-playbook site.yml -v             # more output (-vvv: SSH-level detail)
```

`--check --diff` against production is how you find drift: every "changed" it predicts is a server that no longer matches the code.

Quick check

You run your playbook twice. The second run says `changed=1` , always on the same task. What's the most likely cause?

That task isn't declarative — usually a `shell / command` task, which reports "changed" every time because Ansible can't check its state. Replace it with the module that does the job.

Block 4

Debugging checklist

When it doesn't work

Symptom	Likely cause
skipping: no hosts matched	hosts: doesn't match a group in the inventory, or the wrong inventory is used
Failed to connect ... via ssh	Wrong ansible_host /user/key — or localhost without ansible_connection=local
Missing sudo password / permission denied	Task needs become: true
Destination does not exist (lineinfile)	Add create: true
Second run still changed	A shell / command task, or content that differs each run

Block 5

Today's practice

Today's practice — your first playbook

Practice 07 gives you `ansible.cfg`, a page and a template. You write:

1. `inventory.ini` — group `web` → `localhost`, local connection
2. `site.yml` — one play that, under `/tmp/inf345-p07/app` :
 - creates the directory (`file`)
 - copies `index.html` (`copy`)
 - renders `app.conf` (`template`)
 - ensures `APP_ENV=production` in `.env` (`lineinfile`)

All values as **variables**. No `shell` .

```
ansible-inventory --graph
ansible-playbook site.yml --syntax-check
ansible-playbook site.yml          # changed=4
ansible-playbook site.yml          # changed=0
ansible-playbook site.yml -e app_port=9090
```

Graded automatically: inventory, syntax, the files it creates, the template and the variable override, idempotence, and real modules only.

By the end of this lesson, you should be able to

- ☐ Explain configuration drift, and why declarative + idempotent fixes it
- ☐ Write an inventory with groups, host vars and a local connection
- ☐ Use ad-hoc commands to check and change hosts
- ☐ Write a playbook with variables, a template and a handler
- ☐ Prove it's idempotent — and dry-run it with `--check --diff`

Before next lecture

- ☐ Finish today's practice if you didn't wrap it up in session — same Maru flow as always
- ☐ **Lab 02 (RHA RH294)** is due **Wed, Nov 4, 23:59** — its first chapters are today's material; see "Red Hat labs" in Maru
- ☐ Lab 01 (DO188) closed on Sunday — your final Lab 01 grade is in "My grades"

Bring a laptop with Ansible working (or Docker) — next week builds on today's playbook.

Next lecture

Playbooks, roles & variables — organising real playbooks: roles, group and host vars, and where each variable should live.