Ansible Basics

Mastering Infrastructure Automation from the Ground Up

MODULE OVERVIEW

What We'll Cover Today

01

Introduction to Ansible

Understanding automation and Ansible's role in modern infrastructure

02

YAML Fundamentals

Mastering the language that powers Ansible configurations

03

Installation & Setup

Getting your Ansible environment ready for action

04

Core Components

Inventory, modules, playbooks, and templates

05

Practical Application

Running ad-hoc commands and writing effective playbooks

The Automation Challenge

Imagine managing 50 web servers. Each needs a security patch installed tonight. You could SSH into each one manually, typing the same commands 50 times. That's 50 chances for typos, 50 times to wait for connections, and hours of your evening gone.

Or you could write a simple Ansible playbook and let automation handle it in minutes while you grab coffee. That's the power we're unlocking today.

GETTING STARTED

What is Ansible?

Ansible is an open-source automation tool that simplifies configuration management, application deployment, and task automation across your infrastructure. Unlike traditional tools, it's **agentless**—meaning you don't need to install software on the machines you're managing.

Think of Ansible as your infrastructure's remote control. Instead of manually configuring each server, you describe the desired state in simple, human-readable files, and Ansible makes it happen.

Key Advantages

- No agents required—uses SSH
- Simple YAML syntax
- Idempotent operations
- Massive module library
- Strong community support

How Ansible Works

Ansible operates from a control node (your laptop or a dedicated server) and communicates with managed nodes using SSH. It pushes small Python modules to perform tasks, executes them, and removes them when done. No permanent footprint, no background daemons—just clean, efficient automation.

YAML: Ansible's Language

Before diving into Ansible, we need to understand YAML (YAML Ain't Markup Language). It's the human-friendly format Ansible uses for all its configuration files. If you've ever been frustrated by complex XML or JSON, you'll appreciate YAML's simplicity.

SYNTAX GUIDE

YAML Basics Refresher

Key-Value Pairs

```
name: web_server
port: 80
enabled: true
```

The foundation of YAML—simple attribute assignments

Lists

```
packages:
  - nginx
  - postgresql
  - redis
```

Use dashes to create ordered collections

Nested Structures

```
database:
  host: db.example.com
  port: 5432
  credentials:
    user: admin
```

Indentation creates hierarchy—use spaces, not tabs

YAML Pro Tips

- ### Indentation Matters

  Use 2 spaces per level—consistent spacing is critical

- ### Quotes Are Optional

  Use them for strings with special characters or to force string type

- ### Comments Start with #

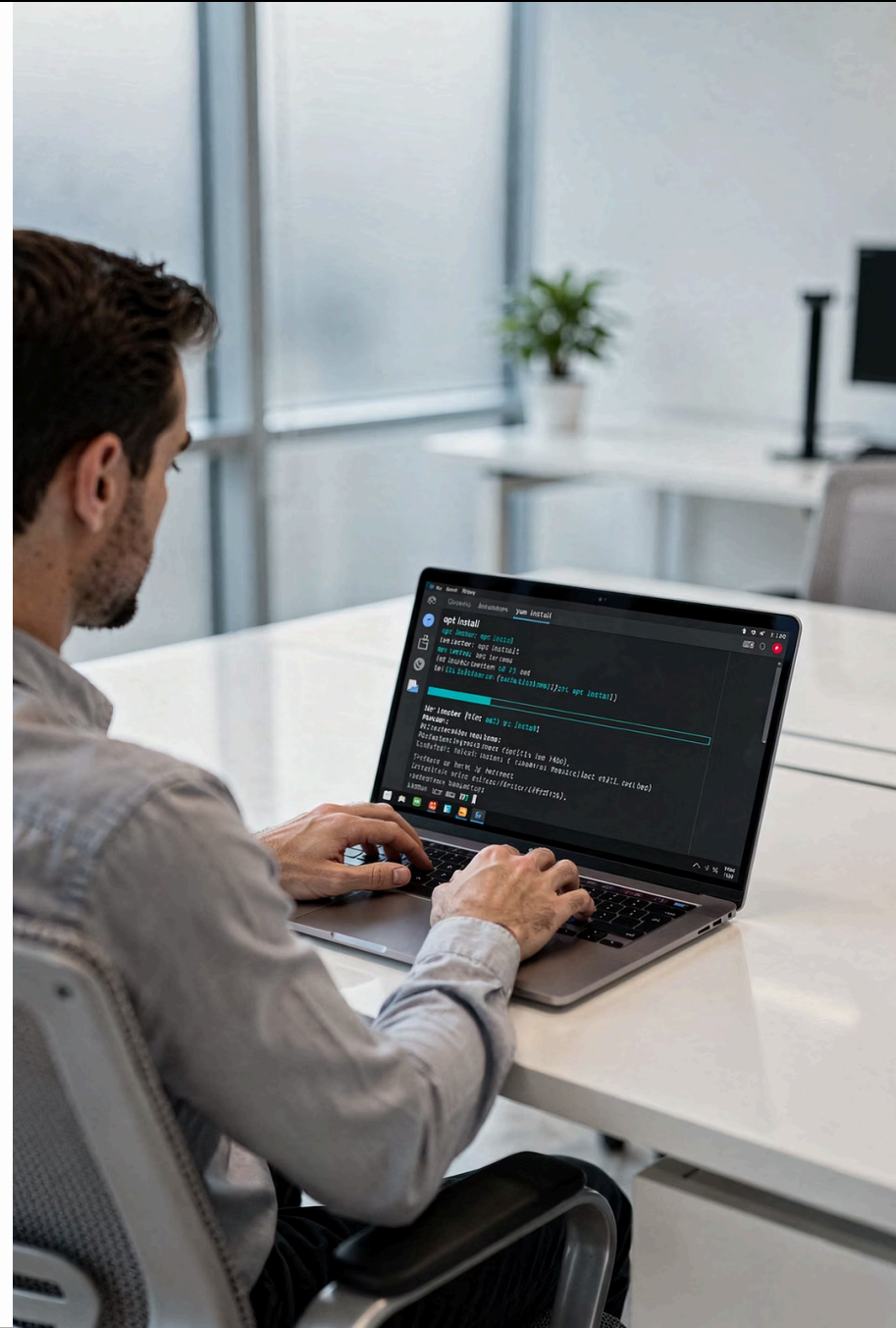
  Document your configurations for future you

Common YAML Mistake

Mixing tabs and spaces will break your files. Configure your editor to convert tabs to spaces automatically. This single tip will save you hours of debugging!

Installing Ansible

Getting Ansible up and running is straightforward. Since it's written in Python, we'll use Python's package manager. The control node (where you run Ansible) needs to be Linux, macOS, or WSL on Windows. Managed nodes can be almost anything with SSH access.



SETUP GUIDE

Installation Steps

Update System

```
sudo apt update
sudo apt upgrade -y
```

Ensure your package manager is current

Install Python & Pip

```
sudo apt install python3 python3-pip -y
```

Ansible requires Python 3.6 or newer

Install Ansible

```
pip3 install ansible
```

Or use your distribution's package manager

Verify Installation

```
ansible --version
```

You should see version info and Python path

Configuring SSH Access

Ansible communicates with managed nodes via SSH. For seamless automation, set up SSH key-based authentication so you don't need to enter passwords repeatedly.

```
# Generate SSH key pair
ssh-keygen -t rsa -b 4096

# Copy public key to managed nodes
ssh-copy-id user@node1.example.com
ssh-copy-id user@node2.example.com

# Test passwordless login
ssh user@node1.example.com
```

Once configured, Ansible can connect automatically without prompting for credentials.

Security Best Practice

Create a dedicated Ansible user on managed nodes with sudo privileges. Use sudo visudo to add:

```
ansible_user ALL=(ALL) NOPASSWD: ALL
```

This enables privilege escalation without password prompts during automation.

The Ansible Inventory

The inventory is your infrastructure's address book. It tells Ansible which machines exist, how to connect to them, and how they're organized. You can use a simple text file or dynamic sources that query cloud providers in real-time.

CONFIGURATION

Creating Your First Inventory

Let's build a basic inventory file for a typical web application infrastructure:

```
# /etc/ansible/hosts or ./inventory.ini

[webservers]
web1.example.com ansible_host=192.168.1.10
web2.example.com ansible_host=192.168.1.11
web3.example.com ansible_host=192.168.1.12

[databases]
db1.example.com ansible_host=192.168.1.20
db2.example.com ansible_host=192.168.1.21

[loadbalancers]
lb1.example.com ansible_host=192.168.1.30

[production:children]
webservers
databases
loadbalancers
```

Groups help you target multiple servers at once. The `:children` syntax creates parent groups that contain other groups.

Inventory Variables

Host Variables

Assign specific values to individual hosts:

```
web1.example.com ansible_port=2222 ansible_user=deploy
```

Group Variables

Apply settings to all hosts in a group:

```
[webservers:vars]
ansible_user=www-data
http_port=8080
```

Real-world example: Your dev servers might use different SSH ports or usernames than production. Inventory variables let you handle these differences without duplicating playbooks.

Ansible Modules: The Building Blocks

Modules are Ansible's units of work—reusable scripts that perform specific tasks. Ansible ships with over 3,000 modules for everything from installing packages to managing cloud resources. You don't write these; you just call them with the right parameters.

Essential Modules You'll Use Daily

apt / yum / dnf

Install, update, or remove software packages on Linux systems. Choose the module that matches your distribution's package manager.

copy

Transfer files from your control node to managed nodes. Perfect for deploying configuration files or application assets.

service / systemd

Start, stop, restart, or enable services. Ensures your applications are running and configured to start at boot.

user

Create and manage user accounts, including passwords, groups, and home directories across your infrastructure.

command / shell

Execute arbitrary commands on remote systems. Use when no specialized module exists for your task.

mysql_db / postgresql_db

Manage database operations including creation, deletion, and backup of databases and users.

MODULE EXAMPLE

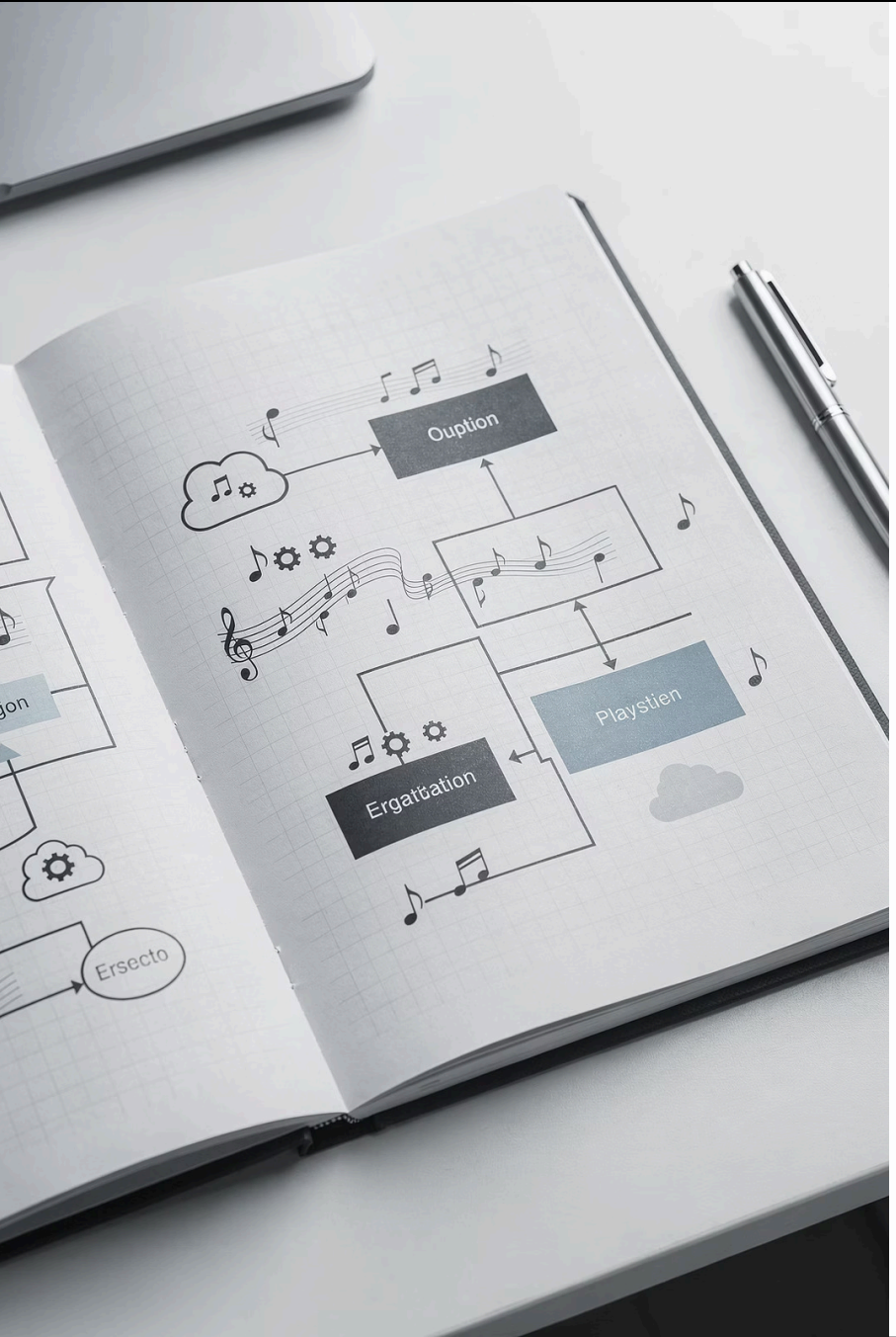
Using Modules: A Practical Example

Let's say you need to ensure Nginx is installed and running on all web servers. Here's how the `apt` and `service` modules work together:

```
# Install nginx package
- name: Install nginx web server
  apt:
    name: nginx
    state: present
    update_cache: yes

# Ensure nginx is running
- name: Start and enable nginx service
  service:
    name: nginx
    state: started
    enabled: yes
```

The `state` parameter is idempotent—run it 100 times and you'll get the same result. Nginx will be installed and running, but Ansible won't reinstall it unnecessarily.



Ansible Playbooks: Orchestrating Automation

Playbooks are where the magic happens. They're YAML files that define a series of tasks to execute on your infrastructure. Think of them as recipes: a list of ingredients (modules) and steps (tasks) that produce a desired outcome.

Anatomy of a Playbook

```
---
- name: Configure web servers
  hosts: webservers
  become: yes

  tasks:
    - name: Install nginx
      apt:
        name: nginx
        state: present

    - name: Copy site config
      copy:
        src: files/nginx.conf
        dest: /etc/nginx/nginx.conf
      notify: restart nginx

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

Play Definition

Names the play, specifies target hosts, and sets privilege escalation

Tasks Section

Ordered list of actions to perform using modules

Handlers

Special tasks that only run when notified—perfect for service restarts

Playbook Best Practices

Name Everything

Every play and task should have a descriptive name. When you run the playbook, these names appear in the output, making it easy to track progress and debug issues.

Use Variables

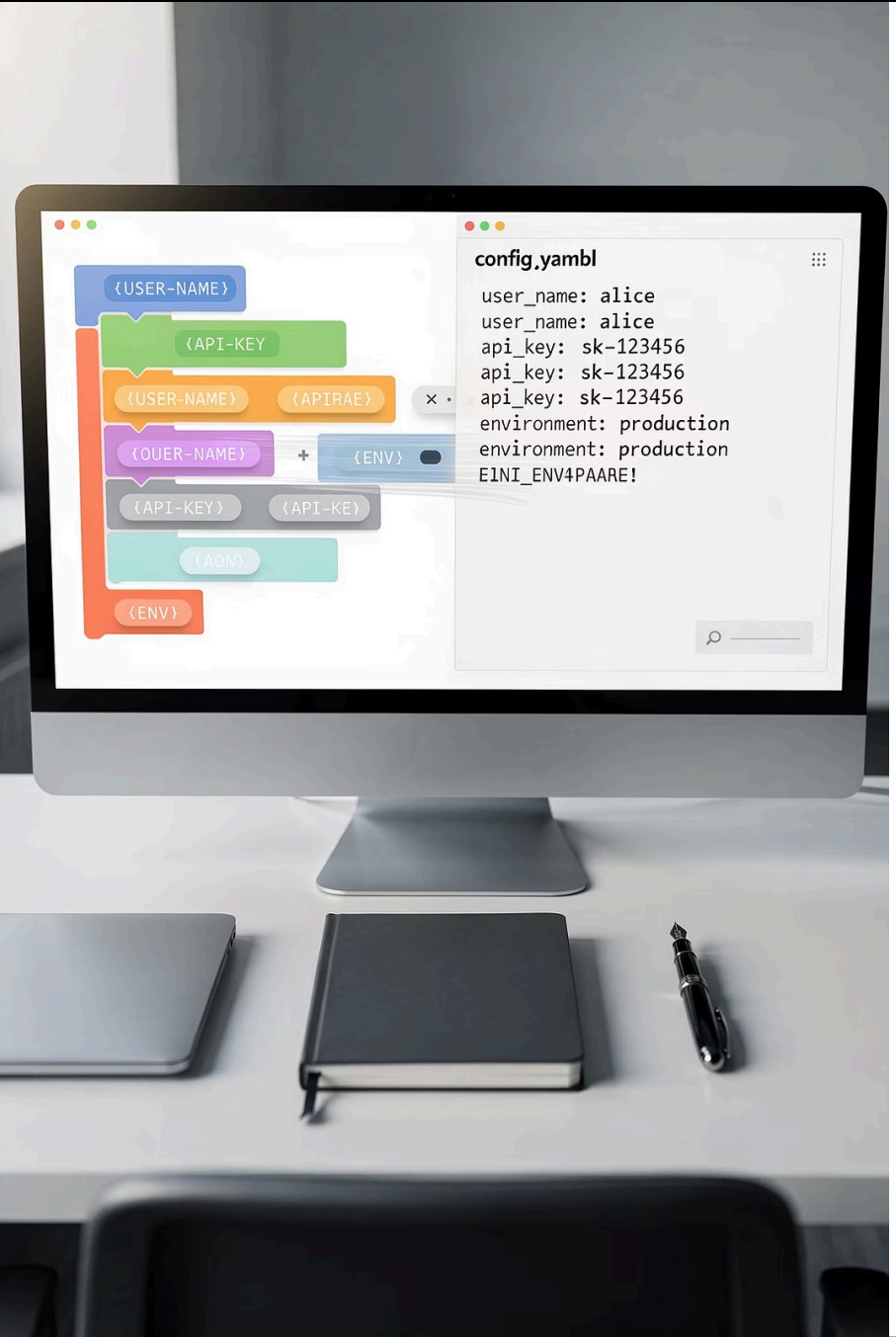
Don't hardcode values. Use variables for anything that might change between environments—ports, paths, versions, etc. This makes playbooks reusable.

Leverage Handlers

Services should restart only when their configuration changes. Handlers ensure restarts happen once at the end, even if multiple tasks trigger them.

Test in Stages

Use the `--check` flag for dry runs. Start with a single host before rolling out to production. Small iterations prevent big disasters.



Jinja2 Templates: Dynamic Configuration

Templates let you create configuration files with variable content. Instead of maintaining separate config files for dev, staging, and production, you maintain one template with placeholders. Ansible fills in the values based on your variables.

TEMPLATE SYNTAX

Creating a Jinja2 Template

Here's a practical example—a Nginx configuration template that adjusts based on environment:

Template File (nginx.conf.j2)

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

    root {{ document_root }};

    location / {
        proxy_pass http://{{ backend_host }}:{{ backend_port }};
        proxy_set_header Host $host;
    }

    {% if enable_ssl %}
    listen 443 ssl;
    ssl_certificate {{ ssl_cert_path }};
    ssl_certificate_key {{ ssl_key_path }};
    {% endif %}
}
```

Variables (vars.yml)

```
http_port: 80
server_name: api.example.com
document_root: /var/www/html
backend_host: 127.0.0.1
backend_port: 3000
enable_ssl: true
ssl_cert_path: /etc/ssl/cert.pem
ssl_key_path: /etc/ssl/key.pem
```

Ansible replaces `{{ variable }}` with actual values at runtime.

Jinja2 Template Features

Variable Substitution

Use `{{ variable_name }}` to insert variable values directly into your templates.

Conditional Logic

Use `{% if condition %}` blocks to include or exclude content based on variables.

Loops

Use `{% for item in list %}` to generate repetitive configuration sections dynamically.

Filters

Transform variables with filters like `{{ name | upper }}` or `{{ price | round(2) }}`.

Using Templates in Playbooks

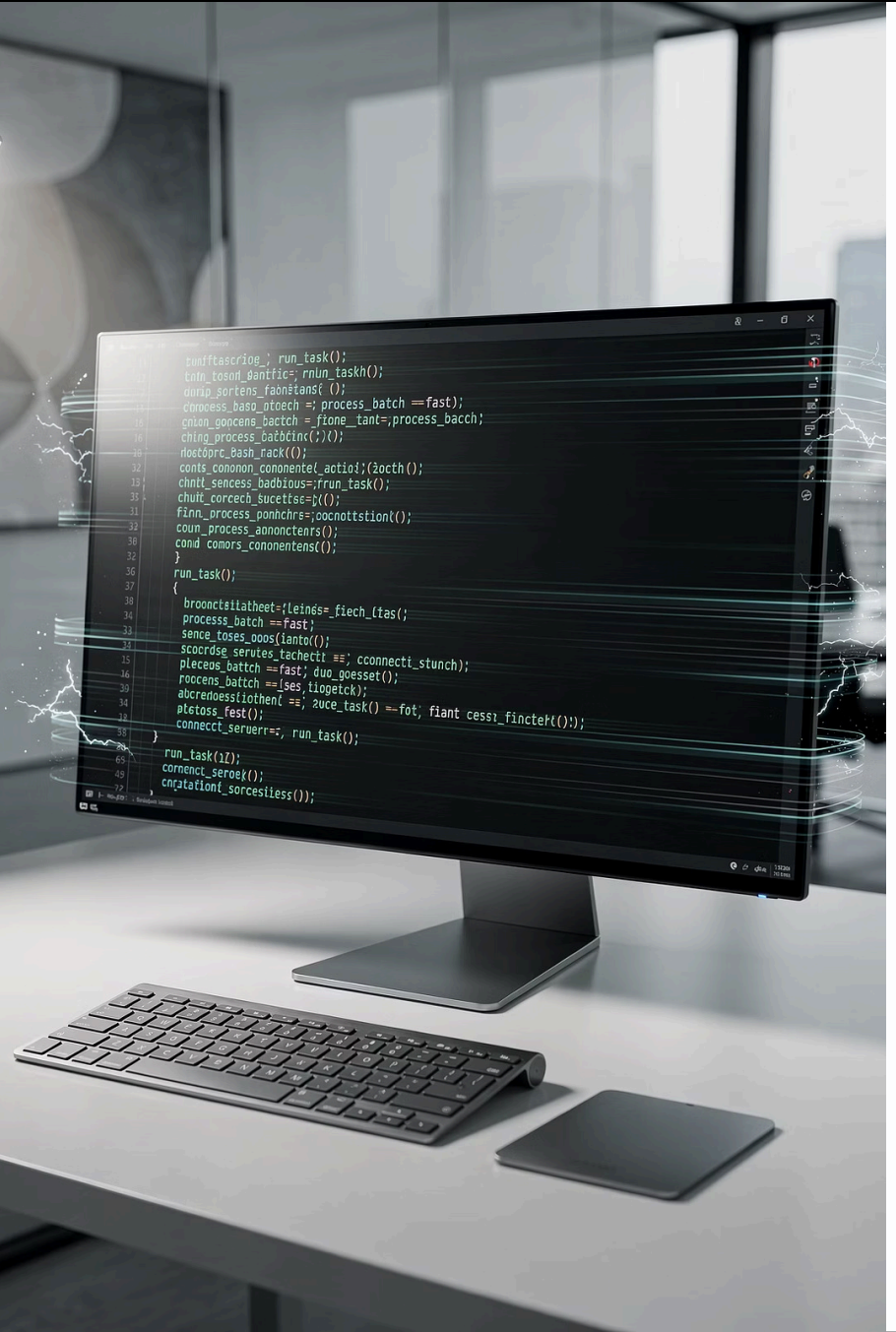
The `template` module processes your Jinja2 files and deploys them to managed nodes:

```
---
- name: Deploy nginx configuration
 hosts: webservers
 become: yes

 vars:
 http_port: 80
 server_name: "{{ inventory_hostname }}"
 backend_port: 3000
 enable_ssl: true

 tasks:
 - name: Generate nginx config from template
 template:
 src: templates/nginx.conf.j2
 dest: /etc/nginx/sites-available/default
 owner: root
 group: root
 mode: '0644'
 notify: reload nginx

 handlers:
 - name: reload nginx
 service:
 name: nginx
 state: reloaded
```



Ad-hoc Commands: Quick Wins

Not every task needs a full playbook. Ad-hoc commands let you execute one-off tasks across your infrastructure instantly from the command line. Perfect for quick checks, emergency fixes, or exploring what's possible.

Ad-hoc Command Examples

Check Connectivity

```
ansible webservers -m ping
```

Verify all web servers are reachable

Install Package

```
ansible all -m apt -a "name=htop state=present" -b
```

Install htop on all servers (requires sudo)

Gather Facts

```
ansible databases -m setup -a "filter=ansible_memory_mb"
```

Check memory on database servers

Copy File

```
ansible webservers -m copy -a "src=/tmp/file dest=/tmp/file"
```

Distribute a file to all web servers

Restart Service

```
ansible loadbalancers -m service -a "name=nginx state=restarted" -b
```

Restart nginx on load balancers

Run Command

```
ansible all -m shell -a "df -h | grep /dev/sda"
```

Check disk usage across infrastructure

COMMAND STRUCTURE

Understanding Ad-hoc Command Syntax

```
ansible [pattern] -m [module] -a "[module options]" [flags]
```

Pattern	Which hosts to target (group name, hostname, or `all`)
Module	The module to execute (defaults to `command` if omitted)
Arguments	Parameters to pass to the module, in quotes
Flags	`-b` for sudo, `-K` to prompt for password, `-v` for verbose

Quick tip: Ad-hoc commands are great for learning. Experiment with different modules to understand what they do before incorporating them into playbooks.

Building Complete Playbooks

Now let's combine everything we've learned. Real-world playbooks integrate multiple modules, use variables and templates, and handle complex deployment scenarios. Here's a complete example that deploys a web application.

COMPLETE EXAMPLE

Full Playbook: Web App Deployment

```
---
- name: Deploy web application
  hosts: webservers
  become: yes

  vars:
    app_name: myapp
    app_version: 1.2.3
    app_port: 3000

  tasks:
    - name: Update package cache
      apt:
        update_cache: yes
        cache_valid_time: 3600

    - name: Install required packages
      apt:
        name:
          - nginx
          - nodejs
          - npm
        state: present

    - name: Create application user
      user:
        name: "{{ app_name }}"
        system: yes
        shell: /bin/false

    - name: Create app directory
      file:
        path: "/opt/{{ app_name }}"
        state: directory
        owner: "{{ app_name }}"
        mode: '0755'

    - name: Copy application files
      copy:
        src: "dist/{{ app_name }}-{{ app_version }}.tar.gz"
        dest: "/opt/{{ app_name }}/"
      notify: restart app

    - name: Deploy nginx config from template
      template:
        src: templates/nginx-app.conf.j2
        dest: "/etc/nginx/sites-available/{{ app_name }}"
      notify: reload nginx

    - name: Enable nginx site
      file:
        src: "/etc/nginx/sites-available/{{ app_name }}"
        dest: "/etc/nginx/sites-enabled/{{ app_name }}"
        state: link
      notify: reload nginx

    - name: Ensure app service is running
      systemd:
        name: "{{ app_name }}"
        state: started
        enabled: yes

  handlers:
    - name: restart app
      systemd:
        name: "{{ app_name }}"
        state: restarted

    - name: reload nginx
      service:
        name: nginx
        state: reloaded
```

Your Ansible Journey Starts Now

You've learned the fundamentals of Ansible—from YAML syntax to writing complete playbooks. You understand inventory management, core modules, Jinja2 templates, and both ad-hoc commands and structured automation.

Next steps:

- Set up your own Ansible control node
- Create an inventory of test machines
- Write simple playbooks for common tasks
- Experiment with different modules
- Build templates for your configurations

Remember: automation is a journey. Start small, iterate often, and gradually build complexity. Every manual task you automate is time saved and errors prevented.

Key Takeaways

- Ansible is agentless and uses SSH
- YAML is human-readable and structured
- Modules are reusable task units
- Playbooks orchestrate automation
- Templates enable dynamic configs
- Start with ad-hoc commands to learn