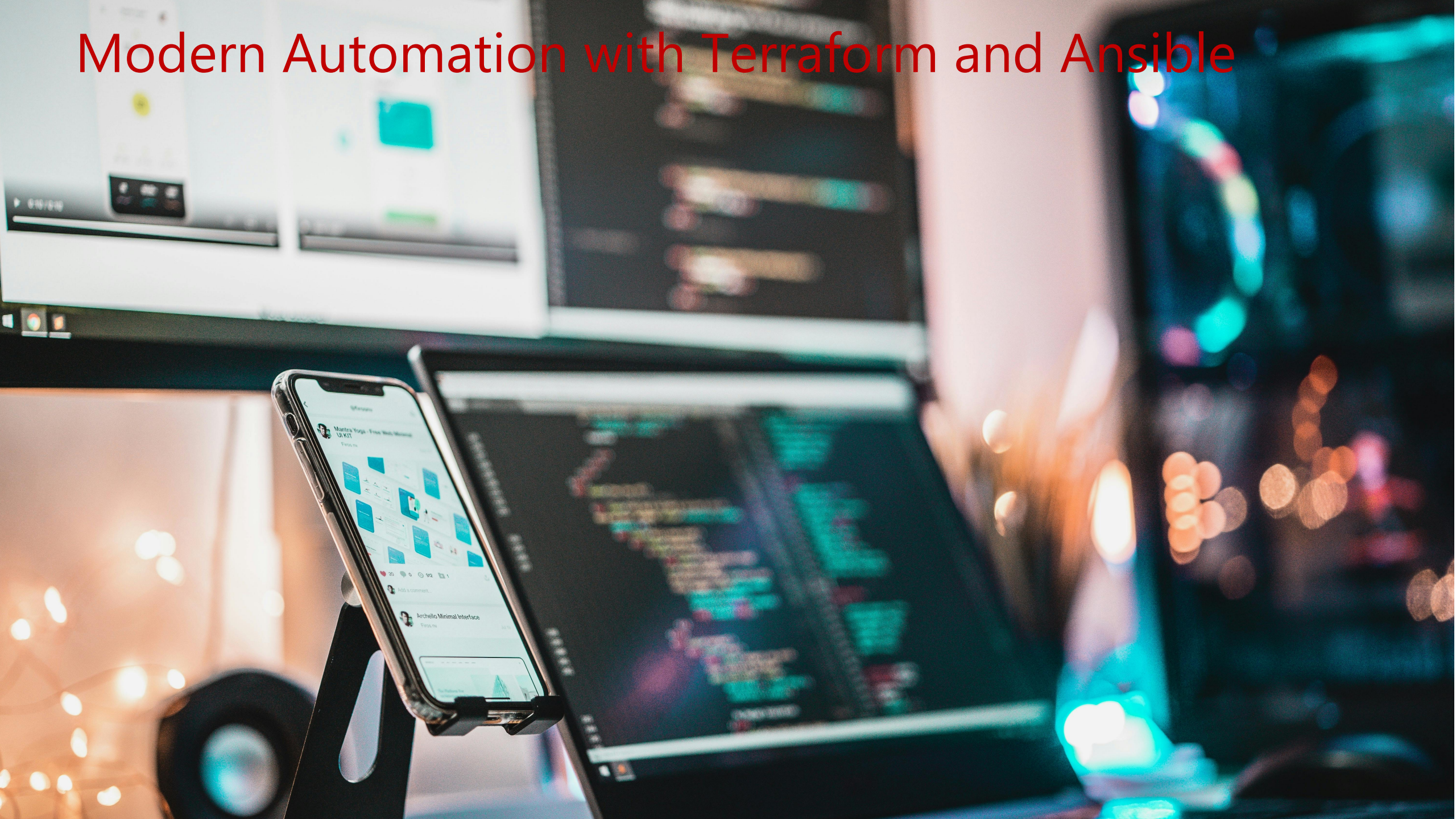
Modern Automation with Terraform and Ansible



TEKsystems Global Services

WORKFORCE DEVELOPMENT

Ansible Ad-hoc

An Ansible ad hoc command uses the `/usr/bin/ansible` command-line tool to automate a single task on one or more managed nodes. ad hoc commands are quick and easy, but they are not reusable.

Why learn about ad hoc commands first? ad hoc commands demonstrate the simplicity and power of Ansible. The concepts you learn will port over directly to the playbook language.

POP QUIZ: DISCUSSION

What are some use-cases for ad-hoc mode?

POP QUIZ: DISCUSSION

What are some use-cases for ad-hoc mode?

- Copy files

POP QUIZ: DISCUSSION

What are some use-cases for ad-hoc mode?

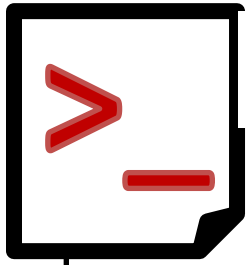
- Copy files
- Manage packages, users, groups

POP QUIZ: DISCUSSION

What are some use-cases for ad-hoc mode?

- Copy files
- Manage packages, users, groups
- Reboot servers

Ansible ad-hoc



Scripted Ad-hoc

```
ansible -m copy -a "src=master.gitconfig dest=~/.gitconfig" localhost

ansible -m homebrew -a "name=bat state=latest" localhost

ansible -i hosts all -m ping -u ubuntu
```

Ansible ad-hoc

ad hoc commands are great for tasks you repeat rarely. For example, if you want to power off all the machines in your lab for Christmas vacation, you could execute a quick one-liner in Ansible without writing a playbook. An ad hoc command looks like this:

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

Example of installing packages on nodes in webservers group.

```
$ ansible webservers -m yum -a "name=apache state=present"
```

Lab: Ansible ad-hoc

YAML

Playbooks are written in YAML.
YAML is used because it is easier for humans to read and write than other data formats like XML and JSON.
It is a format widely used by many tools (Kubernetes, Docker compose, Machine Learning, etc.)

YAML Basics

For Ansible, nearly every YAML file starts with a list. Each item in the list is a list of key/value pairs, commonly called a “hash” or a “dictionary”. So, we need to know how to write lists and dictionaries in YAML.

There’s another small quirk to YAML. All YAML can optionally begin with --- and end with …. This is part of the YAML format and indicates the start and end of a document.

YAML Basics

All members of a list are lines beginning at the same indentation level starting with a "- " (a dash and a space):

Example:

```
---
# A list of tasty fruits
- Apple
- Orange
- Strawberry
- Mango
...
```

YAML Basics

A dictionary is represented in a simple key: value form (the colon must be followed by a space):

Example:

```
---
# An employee record
martin:
  name: Martin D'vloper
  job: Developer
  skill: Elite
```

YAML Basics

More complicated data structures are possible, such as lists of dictionaries, dictionaries whose values are lists or a mix of both:

Example:

```
---
# Employee records
- martin:
    name: Martin D'vloper
    job: Developer
    skills:
      - python
      - perl
      - pascal
- tabitha:
    name: Tabitha Bitumen
    job: Developer
    skills:
      - lisp
      - fortran
      - erlang
```

YAML Basics

Values can span multiple lines using | or >.

Spanning multiple lines using a “Literal Block Scalar” | will include the newlines and any trailing spaces.

Using a “Folded Block Scalar” > will fold newlines to spaces; it’s used to make what would otherwise be a very long line easier to read and edit.

In either case the indentation will be ignored.

YAML Basics

Example:

```
include_newlines: |
        exactly as you see
        will appear these three
        lines of poetry

fold_newlines: >
        this is really a
        single line of text
        despite appearances
```

Ansible Playbook

Ansible Playbooks offer a repeatable, reusable, simple configuration management and multi-machine deployment system, one that is well suited to deploying complex applications.

If you need to execute a task with Ansible more than once

- Write a playbook
- Put it under source control.

Then you can use the playbook to push out new configurations or confirm the configuration of remote systems.

Ansible Playbook

Playbooks can:

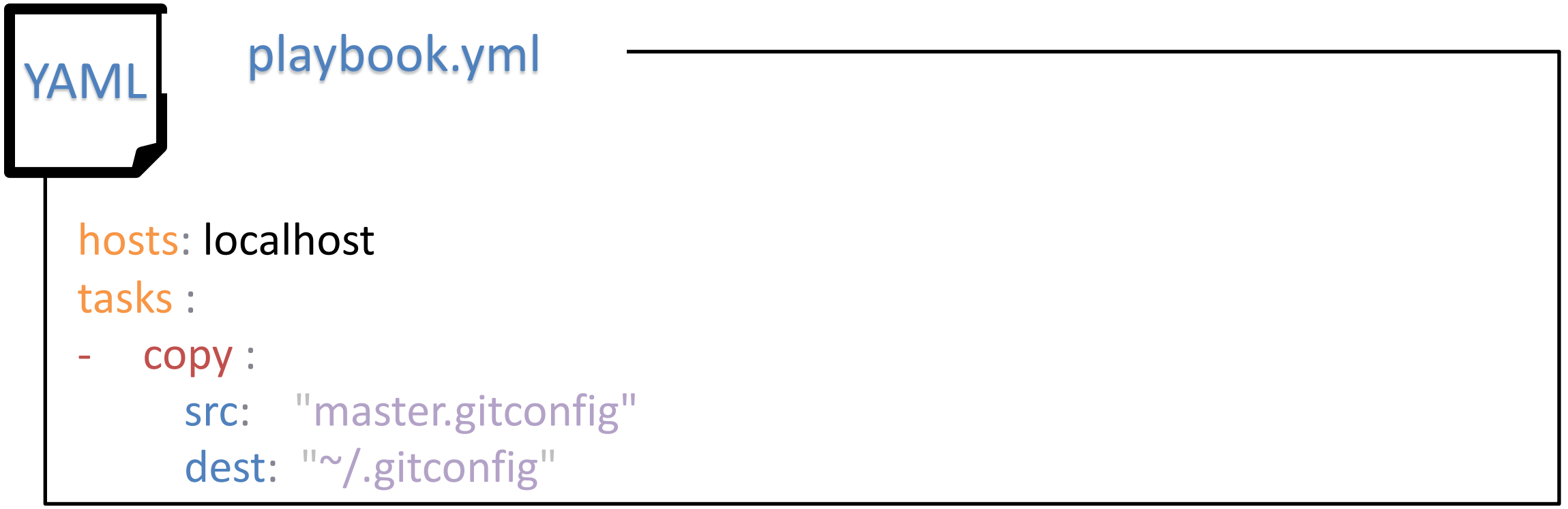
- Declare configurations
- Orchestrate steps of any manual ordered process, on multiple sets of machines, in a defined order
- Launch tasks synchronously or asynchronously

Ansible Playbook

A playbook is composed of one or more ‘plays’ in an ordered list.

The terms ‘playbook’ and ‘play’ are sports analogies. Each play executes part of the overall goal of the playbook, running one or more tasks. Each task calls an Ansible module.

Playbook



Playbook

playbook.yml

```
hosts: localhost
tasks :
-    copy :
        src:    "master.gitconfig"
        dest:  "~/.gitconfig"
```

```
ansible-playbook playbook.yml
```

Ansible Playbook Execution

A playbook runs in order from top to bottom. Within each play, tasks also run in order from top to bottom.

Playbooks with multiple ‘plays’ can orchestrate multi-machine deployments, running one play on your webservers, then another play on your database servers, then the third play on your network infrastructure, and so on.

Ansible Playbook Tips

At a minimum, each play defines two things:

- The managed nodes to target, using a pattern
- At least one task to execute

- Ansible creates a <playbook>.retry playbook for hosts where it failed. You can execute the <playbook>.retry playbook and it will try to run it ONLY on the hosts that failed.
- Limit: Used to run Ansible playbook only on hosts you specify. Great for testing on one host.
- Whitespace: Ansible is like Python, it requires correct indentation. (ansible lint, syntax-check, etc.)

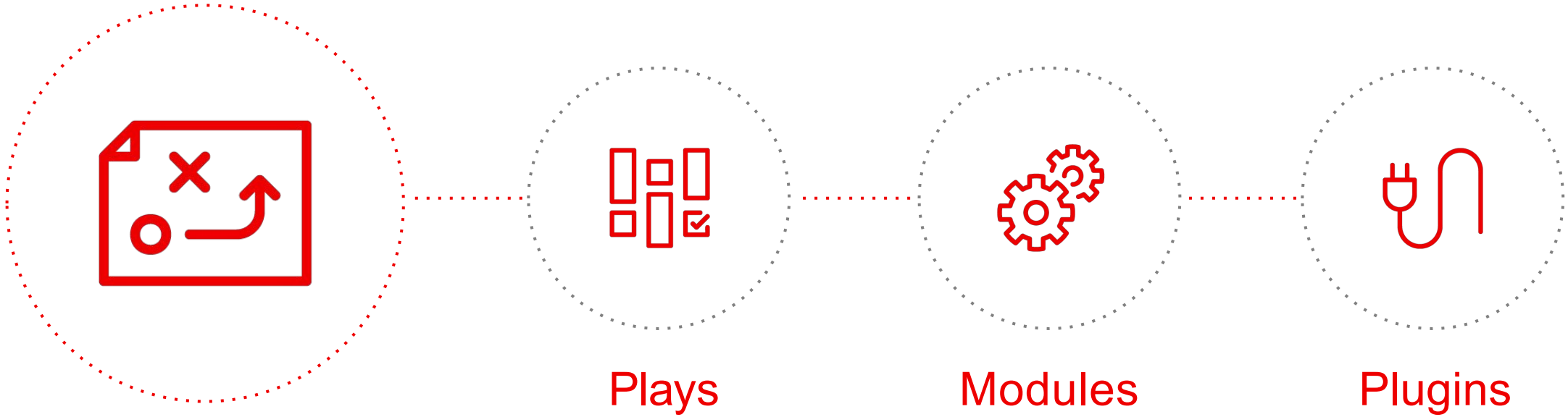
Playbooks

Simple playbook to install and configure Apache web server

```
---
- name: install and start apache
  hosts: web
  become: yes

  tasks:
    - name: httpd package is present
      yum:
        name: httpd
        state: latest

    - name: latest index.html file is present
      template:
        src: files/index.html
        dest: /var/www/html/

    - name: httpd is started service:
        name: httpd state: started
```

Playbooks

What makes up an Ansible playbook?



Ansible Plays

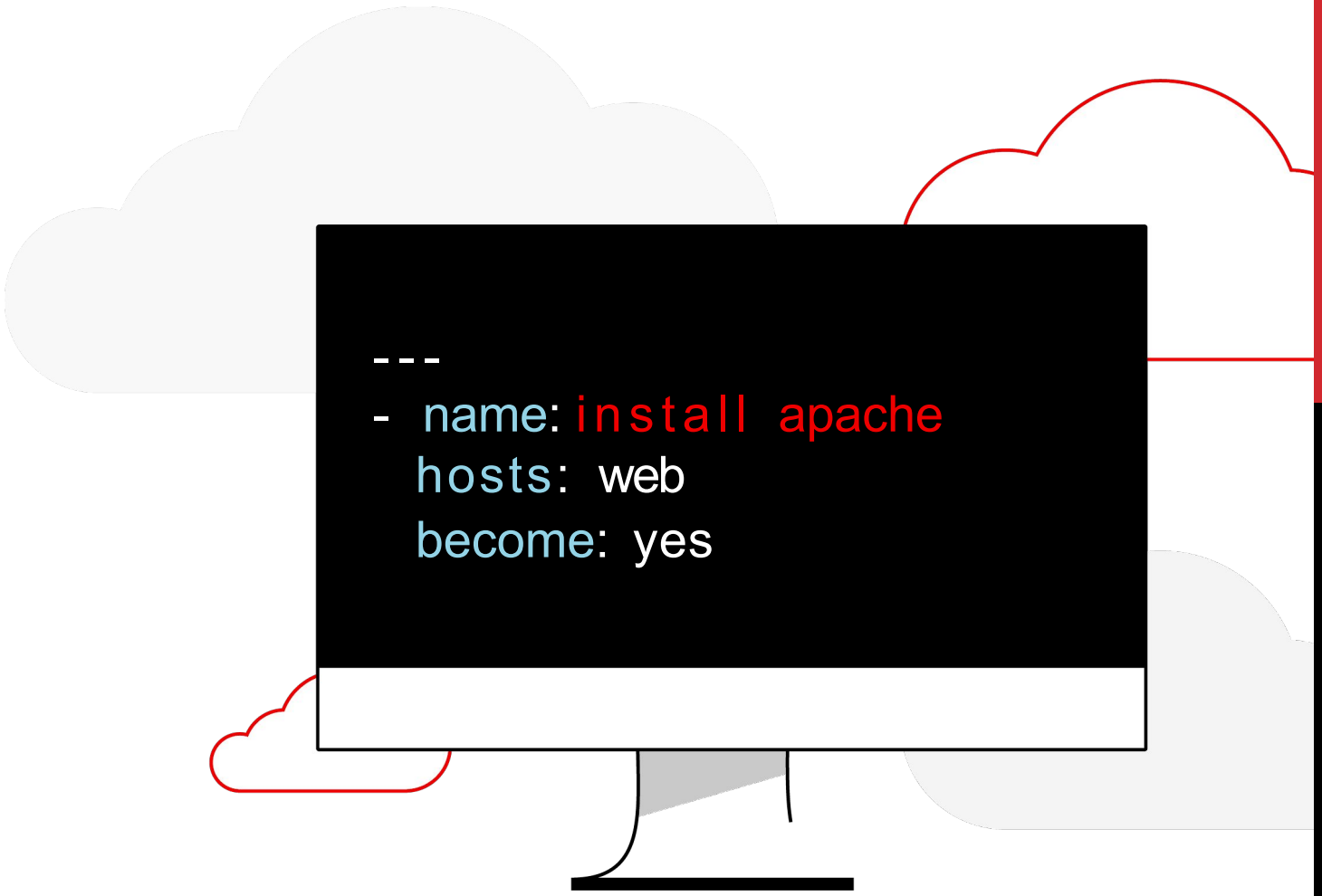
What am I automating?

What are they?

Top level specification for a group of tasks. Will tell that play which hosts it will execute on and control behavior such as fact gathering or privilege level.

Building blocks for playbooks

Multiple plays can exist within an Ansible playbook that execute on different hosts.



Ansible Modules

The “tools in the toolkit”

What are they?

Parametrized components with internal logic, representing a single step to be done.

The modules “do” things in Ansible.

Language

Usually Python, or Powershell for Windows setups. But can be of any language.

Ansible Plugins

The “extra bits”

What are they?

Plugins are pieces of code that augment Ansible’s core functionality. Ansible uses a plugin architecture to enable a rich, flexible, and expandable feature set.

More Complete Playbook

```yaml
- name: Install Docker
  hosts: tag_role_k8s_master:tag_role_k8s_member
  remote_user: ubuntu
  gather_facts: no
  handlers:
    - include: roles/handlers/main.yml
  tasks:
    - package:
        name: "docker-ce"
        state: latest
    - user:
        name: "{{ ansible_ssh_user }}"
        groups: docker
        append: yes
      notify: restart docker
```

Name of Play

More Complete Playbook

```yaml
- name: Install Docker
  hosts: tag_role_k8s_master:tag_role_k8s_member
  remote_user: ubuntu
  gather_facts: no
  handlers:
    - include: roles/handlers/main.yml
  tasks:
    - package:
        name: "docker-ce"
        state: latest
    - user:
        name: "{{ ansible_ssh_user }}"
        groups: docker
        append: yes
      notify: restart docker
```

Hosts to run Play on

More Complete Playbook

```
- name: Install Docker
  hosts: tag_role_k8s_master:tag_role_k8s_member
  remote_user: ubuntu
  gather_facts: no
  handlers:
    - include: roles/handlers/main.yml
  tasks:
    - package:
        name: "docker-ce"
        state: latest
    - user:
        name: "{{ ansible_ssh_user }}"
        groups: docker
        append: yes
      notify: restart docker
```

User to run Play as

More Complete Playbook

```yaml
- name: Install Docker
  hosts: tag_role_k8s_master:tag_role_k8s_member
  remote_user: ubuntu
  gather_facts: no
  handlers:
    - include: roles/handlers/main.yml
  tasks:
    - package:
        name: "docker-ce"
        state: latest
    - user:
        name: "{{ ansible_ssh_user }}"
        groups: docker
        append: yes
      notify: restart docker
```

Define what to do after package is installed.

More Complete Playbook

```yaml
- name: Install Docker
  hosts: tag_role_k8s_master:tag_role_k8s_member
  remote_user: ubuntu
  gather_facts: no
  handlers:
    - include: roles/handlers/main.yml
  tasks:
    - package:
        name: "docker-ce"
        state: latest
    - user:
        name: "{{ ansible_ssh_user }}"
        groups: docker
        append: yes
      notify: restart docker
```

Call the defined handler

More Complete Playbook

```yaml
- name: Install Docker
  hosts: tag_role_k8s_master:tag_role_k8s_member
  remote_user: ubuntu
  gather_facts: no
  handlers:
    - include: roles/handlers/main.yml
  tasks:
    - package:
        name: "docker-ce"
        state: latest
    - user:
        name: "{{ ansible_ssh_user }}"
        groups: docker
        append: yes
      notify: restart docker
```

Tasks to run
- install Docker

More Complete Playbook

```yaml
- name: Install Docker
  hosts: tag_role_k8s_master:tag_role_k8s_member
  remote_user: ubuntu
  gather_facts: no
  handlers:
    - include: roles/handlers/main.yml
  tasks:
    - package:
        name: "docker-ce"
        state: latest
    - user:
        name: "{{ ansible_ssh_user }}"
        groups: docker
        append: yes
      notify: restart docker
```

Create user account

Ansible Variables

Ansible uses variables to manage differences between systems. With Ansible, you can execute tasks and playbooks on multiple different systems with a single command.

- Create variables with YAML syntax, including lists and dictionaries
- Define these variables

Ansible Variables

Variables can be defined in many locations:

- Playbooks
- Inventory
- re-usable files or roles
- command line.

You can also create variables during a playbook run by registering the return value or values of a task as a new variable.

Ansible Variable Valid Names

A variable name can only contain:

- Letters
- Numbers
- Underscores

A variable name cannot begin with a number, but can begin with an underscore.

POP QUIZ: DISCUSSION

Are these valid variable names?

- foo

POP QUIZ: DISCUSSION

Are these valid variable names?

- foo - valid

POP QUIZ: DISCUSSION

Are these valid variable names?

- app.port

POP QUIZ: DISCUSSION

Are these valid variable names?

- app.port - invalid

POP QUIZ: DISCUSSION

Are these valid variable names?

- *ssl_key

POP QUIZ: DISCUSSION

Are these valid variable names?

- *ssl_key - invalid

POP QUIZ: DISCUSSION

Are these valid variable names?

- _web_root

POP QUIZ: DISCUSSION

Are these valid variable names?

- _web_root - valid

Simple Variable

Defining simple variables

```
remote_install_path: /opt/my_app_config
```

After you define a variable, use Jinja2 syntax to reference it. Jinja2 variables use double curly braces.

```
ansible.builtin.template:
  src: foo.cfg.j2
  dest: '{{ remote_install_path }}/foo.cfg'
```

Variable Quotation

If you start a value with {{ foo }}, you must quote the whole expression to create valid YAML syntax. If you do not quote the whole expression, the YAML parser cannot interpret the syntax - it might be a variable or it might be the start of a YAML dictionary.

This example will error.

```
- hosts: app_servers
  vars:
    app_path: {{ base_path }}/22
```

ERROR! Syntax Error while loading YAML

Variable Quotation

If you start a value with {{ foo }}, you must quote the whole expression to create valid YAML syntax. If you do not quote the whole expression, the YAML parser cannot interpret the syntax - it might be a variable or it might be the start of a YAML dictionary.

Fix the issue by quoting the entire expression.

```
- hosts: app_servers
  vars:
    app_path: "{{ base_path }}/22"
```

Ansible Modules

Modules are the main building blocks of Ansible playbooks. Although we do not generally speak of “module plugins”, a module is a type of plugin.

Common modules:

- Working with files: copy, archive, unarchive, get_url
- user, group
- ping
- service
- yum, apt, package
- template

Ansible Modules

Common modules:

- Lineinfile
  - Manipulate text in files
    - Add alias for hosts
    - Supports regex
    - Idempotent
- Shell/command
- Script module
- Debug module

Lab: Ansible playbook fundamentals

Defining Variables - Play

You can define different variables for each individual host, or set shared variables for a group of hosts in your inventory. You can also define variables in a play.

```
- hosts: webservers
  vars:
    http_port: 80
```

NOTE: When you define variables in a play, they are only visible to tasks executed in that play.

Defining Variables – External File

You can define variables in reusable variables files and/or in reusable roles. When you define variables in reusable variable files, the sensitive variables are separated from playbooks.
This separation enables storing playbooks in source control and even share them without exposing passwords or other sensitive data.

Defining Variables – External File

This example shows how you can include variables defined in an external file:

```
- hosts: all
  remote_user: root
  vars:
    favcolor: blue
  vars_files:
    - /var/external_vars.yml
```

Defining Variables – External File

The contents of each variables file is a simple YAML dictionary.
For example:

```
---
var1: myfavorite
var2: mysecondfavorite
```

Defining Variables – Command Line

You can define variables when you run your playbook by passing variables at the command line.

--extra-vars (-e)

```
ansible-playbook playbook.yml -extra-vars "red"
```

Defining Variables – Command Line

If you have a lot of special characters, use a JSON or YAML file containing the variable definitions.

```
ansible-playbook release.yml -extra-vars
"@some_var_file.json"
```

Variable Precedence

Ansible does apply variable precedence, and you might have a use for it. Here is the order of precedence from least to greatest (the last listed variables override all other variables):

1. Command line values (for example, -u my_user, these are not variables)
2. role defaults (defined in role/defaults/main.yml)
3. Inventory file or script group vars
4. Inventory group_vars/all
5. Playbook group_vars/all
6. inventory group_vars/*
7. playbook group_vars/*
8. inventory file or script host vars
9. inventory host_vars/*
10. playbook host_vars/*

Variable Precedence

11. host facts / cached set_facts
12. play vars
13. play vars_prompt
14. play vars_files
15. role vars (defined in role/vars/main.yml)
16. block vars (only for tasks in block)
17. task vars (only for the task)
18. include_vars
19. set_facts /registered vars
20. role (and include_role params)
21. include params
22. extra vars (for example –e "user=my_user") – always wins precedence.

Working With Variables - List

A list variable combines a variable name with multiple values. The multiple values can be stored as an itemized list or in square brackets [], separated with commas.

Defining variables as lists:

You can define variables with multiple values using YAML lists. For example:

Region:

- northeast
- southeast
- midwest

Working With Variables - List

Referencing list variables:

When you use variables defined as a list (also called an array), you can use individual, specific fields from that list. The first item in a list is item 0, the second item is item 1.

For example, to select the first element in the array:

```
region: "{{ region[0] }}"
```

POP QUIZ: DISCUSSION

What does this expression return?

```
region: "{{ region[0] }}"
```

POP QUIZ: DISCUSSION

What does this expression return?

```
region: "{{ region[0] }}"
```

- A: northeast
- B: southeast
- C: midwest

POP QUIZ: DISCUSSION

What does this expression return?

```
region: "{{ region[0] }}"
```

- A: northeast
- B: southeast
- C: midwest

Working With Variables - Dictionary

A dictionary stores the data in key-value pairs. Usually, dictionaries are used to store related data, such as the information contained in an ID or a user profile.

You can define more complex variables using YAML dictionaries. A YAML dictionary maps keys to values. For example:

```
region:
  field1: one
  field2: two
```

Working With Variables - Dictionary

When you use variables defined as a key:value dictionary (also called a hash), you can use individual, specific fields from that dictionary using either bracket notation or dot notation:

```
foo['field']
foo.field
```

Dot notation can cause problems because some keys collide with attributes and methods of python dictionaries.

PROTIP: Use bracket notation in most cases.

Registering Variables

You can create variables from the output of an Ansible task with the task keyword `register`. You can use registered variables in any later tasks in your play.

```
- hosts: web_servers
  tasks:
    - name: Register shell command output as variable
      shell: /usr/bin/foo
      register: foo_result
      ignore_errors: true

    - name: Run shell command using output from
previous task
      shell: /usr/bin/bar
      when: foo_result.rc == 5
```

Registering Variables

Registered variables are stored in memory. You cannot cache registered variables for use in future plays. Registered variables are only valid on the host for the rest of the current playbook run.

If a task fails or is skipped, Ansible still registers a variable with a failure or skipped status, unless the task is skipped based on tags.

Querying Nested Data

Many registered variables (and facts) are nested YAML or JSON data structures. You cannot access values from these nested data structures with the simple {{ foo }} syntax. You must use either bracket notation or dot notation. For example, to reference an IP address from your facts using the bracket notation:

```
{{ ansible_facts["eth0"]["ipv4"]["address"] }}
```

Optional Variables

By default, Ansible requires values for all variables in a templated expression. However, you can make specific variables optional.

For example, you might want to use a system default for some items and control the value for others. To make a variable optional, set the default value to the special variable omit:

Optional Variables

In this example, the default mode for the files `/tmp/foo` and `/tmp/bar` is determined by the umask of the system.

Ansible does not send a value for `mode`. Only the third file, `/tmp/baz`, receives the *mode=0444* option.

```
- name: Touch files with optional mode
  ansible.builtin.file:
    dest: "{{ item.path }}"
    state: touch
    mode: "{{ item.mode | default(omit) }}"
  loop:
    - path: /tmp/foo
    - path: /tmp/bar
    - path: /tmp/baz
      mode: "0444"
```

Handlers

Sometimes you want a task to run only when a change is made on a machine. For example, you may want to restart a service if a task updates the configuration of that service, but not if the configuration is unchanged. Ansible uses handlers to address this use case. Handlers are tasks that only run when notified.

Handlers

```
---
- name: Verify Apache installation
  hosts: web
  vars:
    http_port: 80
    max_clients: 200
  tasks::
    - name: Install Apache
    - yum:
    - name: httpd
...
    - name: Apache config
      template:
        src: httpd.j2
        dest: /etc/httpd.conf
      notify:
      - Restart Apache
  handlers:
    - name: Restart Apache
      service:
        name: httpd
        state: restarted
```

Notify Handlers

Tasks can instruct one or more handlers to execute using the `notify` keyword. The `notify` keyword can be applied to a task and accepts a list of handler names that are notified on a task change.

```
tasks:
    name: Template file
    template:
      src: template.j2
      dest: /etc/foo.conf
    notify:
    - Restart apache
    - Restart memcached
```

Notify Handlers

Tasks can instruct one or more handlers to execute using the `notify` keyword. The `notify` keyword can be applied to a task and accepts a list of handler names that are notified on a task change.

```
tasks:
    notify:
    - Restart apache
    - Restart memcached
handlers:
  - name: Restart memcached
    service:
      name: memcached
      state: restarted
  - name: Restart Apache
    service:
      name: apache
      state: restarted
```

Specifying Handlers

Handlers must be named in order for tasks to be able to notify them using the notify keyword.

Alternately, handlers can utilize the listen keyword. Using this handler keyword, handlers can listen on topics that can group multiple handlers as follows:

Specifying Handlers

```
tasks::
    - name: Restart everything
      command: echo "this task will restart the web services"
      notify: "restart web services"

handlers:
  - name: Restart memcached
    service:
      name: memcached
      state: restarted
    listen: "restart web services"

  - name: Restart Apache
    service:
      name: httpd
      state: restarted
    listen: "restart web services"
```

POP QUIZ: DISCUSSION

How have you used handlers?

POP QUIZ: DISCUSSION

When should handlers be used?

- When a configuration file is updated, and the service needs to be restarted.
- When a new release is rolled out

Asynchronous Actions & Polling

Ansible runs tasks synchronously, holding the connection to the remote node open until the action is completed. This means within a playbook; each task blocks the next task.

Subsequent tasks will not run until the current task completes.

Challenges:

- Slow
- Long running tasks block all subsequent tasks

Asynchronous Actions & Polling

Playbooks support asynchronous mode and polling, with a simplified syntax.

You can use asynchronous mode in playbooks to avoid connection timeouts or to avoid blocking subsequent tasks. The behavior of asynchronous mode in a playbook depends on the value of poll.

Asynchronous Actions & Polling

For long running tasks, connections to the host can timeout.

If you want to set a longer timeout limit for a certain task in your playbook, use `async` with `poll`.

Ansible will still block the next task in your playbook, waiting until the async task either completes, fails or times out. However, the task will only time out if it exceeds the timeout limit you set with the `async` parameter.

Asynchronous Actions & Polling

To avoid timeouts on a task, specify its maximum runtime and how frequently you would like to poll for status:

```
---
tasks:
    - name: Long running task (15 sec), wait for up to 45 sec, poll every 5 sec
      command: /bin/sleep 15
      async: 45
      poll: 5
```

- `DEFAULT_POLL_INTERVAL`
  The default polling value is 15 seconds.

  There is no default for the async time limit. If you omit the `async` keyword the tasks run synchronously.

  Default async job cache file: `~/.ansible_async`

Asynchronous Actions & Polling

If you want to run multiple tasks in a playbook concurrently, use `async` with `poll` set to 0.

When you set `poll: 0`, Ansible starts the task and immediately moves on to the next task without waiting for a result.

Each async task runs until it either completes, fails or times out (runs longer than its async value). The playbook run ends **without checking back on async tasks**.

Asynchronous Actions & Polling

Playbook with asynchronous task:

```
---
tasks:
    - name: Long running task, allow for 45 sec, fire and forget
      command: /bin/sleep 15
      async: 45
      poll: 0
```

Be careful! Operations that require a lock (yum, apt, etc.) should not be run using `async` if you intend to run other commands later in the playbook on them.

When running with `poll: 0`, Ansible will not automatically cleanup the async job cache file. It will need to be cleaned up manually using the `async_status` module with `mode: cleanup`.

Asynchronous Actions & Polling

Check the status of an `async` task

```
- name: async task
  yum:
    name: docker-io
    state: present
  async: 1000
  poll: 0
  register: yum_sleeper

- name: Check status of async task
  async_status:
    jid: "{{ yum_sleeper.ansible_job_id }}"
  register: job_result
  until: job_result.finished
  retries: 100
  delay: 10
```

Check And Diff Mode

Ansible provides two modes of execution that validate tasks:

- Check mode
  - Ansible runs without making any changes on remote systems.
- Diff mode
  - Ansible provides before-and-after comparisons.

Check And Diff Mode

Check mode is just a simulation. It will not generate output for tasks that use conditionals based on registered variables (results of prior tasks). However, it is great for validating configuration management playbooks that run on one node at a time.

To run an entire playbook in `check` mode:

```
ansible-playbook foo.yml --check
```

Check And Diff Mode

It is also possible to specify that a task always or never runs in check mode regardless of command-line argument.

```
tasks:
  - name: Always change system
    command: /bin/change_stuff --even-in-check-mode
    check_mode: no


- name: Never change system
    lineinfile:
    line: "important config"
    dest: /path/to/config.conf
    state : present
  check_mode: yes
  register: changes_to_important_config
```

Check And Diff Mode

Running single tasks with `check_mode` can be useful for testing Ansible modules, either to test the module itself or to test the conditions under which it would make changes.

Combining `check_mode` and `register` provides even more detail on potential changes.

Check And Diff Mode

It is possible to skip a task or ignore errors when using `check_mode` by specifying `ansible_check_mode` boolean

```
tasks:
  - name: Skip in check mode
    git:
      repo: ssh://git@github.com/me/hello-world.git
      dest: /home/me/hello-world
    when: not ansible_check_mode

  - name: Ignore errors in check mode
    git:
      repo: ssh://git@github.com/me/hello-world.git
      dest: /home/me/hello-world
    ignore_errors: "{{ ansible_check_mode }}"
```

Check And Diff Mode

The `--diff` option for `ansible-playbook` can be used with `--check` or alone.

When you run in diff mode, any module that supports diff mode reports the changes made or, if used with `--check`, the changes that would have been made.

Diff mode is most common in modules that manipulate files (for example, the `template` module) but other modules might also show 'before and after' information (for example, the `user` module).

Rolling Updates

Batch size:

By default, Ansible will try to manage all the machines referenced in a play in parallel. For a rolling update use case, you can define how many hosts Ansible should manage at a single time by using the `serial` keyword:

Rolling Updates

Example playbook utilizing `serial` keyword.

```
- name: test play
  hosts: webservers
  serial: 2
  gather_facts: False

  tasks:
  - name: task one
    command: hostname
  - name: task two
    command: hostname
```

With 4 hosts in the group 'webservers', 2 would complete the play before moving onto the next 2 hosts.

Rolling Updates

In the previous example, if we had 4 hosts in the group 'webservers', 2 would complete the play before moving on to the next 2 hosts:

```
PLAY [webservers] ****************************************

TASK [task one] ******************************************
changed: [web2]
changed: [web1]

TASK [task two] ******************************************
changed: [web1]
changed: [web2]


PLAY [webservers] ****************************************
```

Rolling Updates

Now that web1 & web2 are complete Ansible continues with web3 & web4

```
PLAY [webservers] ***************************************

TASK [task one] *****************************************
changed: [web3]
changed: [web4]

TASK [task two] *****************************************
changed: [web3]
changed: [web4]

PLAY RECAP **********************************************
web1 : ok=2 changed=2 unreachable=0 failed=0
web2 : ok=2 changed=2 unreachable=0 failed=0
web3 : ok=2 changed=2 unreachable=0 failed=0
web4 : ok=2 changed=2 unreachable=0 failed=0
```

Rolling Updates

The `serial` keyword can also be specified as a percentage, which will be applied to the total number of hosts in a play, in order to determine the number of hosts per pass:

```
- name: test play
  hosts: webservers
  serial: 30%
```

If the number of hosts does not divide equally into the number of passes, the final pass will contain the remainder.

Rolling Updates

The batch size can be specified as a list with integer, or percent:

```
- name: test play
  hosts: webservers
  serial:
    - 1
    - 5
    - 10
```

Above the first batch would contain a single host, next 5, and if any left, each would have 10 until complete.

Rolling Updates

The batch size can be specified as a list with integer, or percent:

```
- name: test play
  hosts: webservers
  serial:
    - "10%"
    - "20%"
    - "100%"
```

Rolling Updates

Maximum Threshold Percentage:
Ansible executes tasks on all hosts in the defined group unless `serial` is defined.

In some situations, such as with the rolling updates, it may be desirable to abort the play when a certain threshold of failures have been reached. To achieve this, you can set a maximum failure percentage on a play as follows:

```
- hosts: webservers
  max_fail_percentage: 30
  serial: 10
```

Ansible Vault

Ansible Vault

Ansible Vault encrypts variables and files so you can protect sensitive content such as passwords or keys rather than leaving it visible as plaintext in playbooks or roles.

To use Ansible Vault you need one or more passwords to encrypt and decrypt content.

Use the passwords with the `ansible-vault` command-line tool to create and view encrypted variables, create encrypted files, encrypt existing files, or edit, re-key, or decrypt files. You can then place encrypted content under source control and share it more safely.

Ansible Vault

Ansible Vault can prompt for a password every time, or you can configure it to use a password file.

```
#ansible.cfg
      [defaults]
        vault_password_file = ~/.vault_pass
```

Ansible Vault

Each time you encrypt a variable or file with Ansible Vault, you must provide a password. When you use an encrypted variable or file in a command or playbook, you must provide the same password that was used to encrypt it.

POP QUIZ: DISCUSSION

Things to consider:

- Do you want to encrypt all your content with the same password, or use different passwords for different needs?
- Where do you want to store your password(s)?

Ansible Vault

Small teams can use a single password for everything encrypted. Store the vault password securely in a file or secret manager.

If you have a large team or many sensitive values to manage it is recommended to use multiple passwords.

You can use different passwords for different users or different levels of access. Depending on your needs, you might want a different password for each encrypted file, for each directory, or for each environment.

Ansible Vault

You might have a playbook that includes two vars files, one for the dev environment and one for the production environment, encrypted with two different passwords.

When you run the playbook, select the correct vault password for the environment you are targeting, using a vault ID.

Vault Id

A vault ID is an identifier for one or more vault secrets.

Vault IDs provide labels to distinguish between individual vault passwords.

To use vault IDs, you must provide an ID label of your choosing and a source to obtain its password (either `prompt` or a file path):

```
--vault-id label@source
```

This switch is available for all commands that interact with vaults:

- ansible-vault
- ansible-playbook
- etc.

Ansible Vault

Create a new encrypted data file

```
ansible-vault create foo.yml
```

Prompt for vault password

```
ansible-playbook --ask-vault-pass myplay.yml
```

Use password file

```
ansible-playbook --vault-password-file pass myfile.yml
```

Ansible Vault

Common vault commands

```
# Edit encrypted files
ansible-vault edit playbook.yml

# Rekeying
ansible-vault rekey play.yml task.yml report.yml

# Encrypt existing files
ansible-vault encrypt foo.yml bar.yml baz.yml

# Decrypting files
ansible-vault decrypt task.yml run.yml play.yml

# View encrypted files
ansible-vault view break.yml fix.yml fun.yml
```

Ansible Tags

Tags

If you have a large playbook, it may become useful to be able to run only a specific part of it rather than running everything in the playbook. Ansible supports a `tags` attribute for this reason.

Tags can be applied at multiple levels including:

- Tasks
- Roles
- Plays
- Blocks

Tags

At the simplest level, you can apply one or more tags to an individual task. You can add tags to tasks in playbooks, in task files, or within a role.

It is also possible to add the same tag to multiple tasks.

Tags

Here's an example showing tasks to install and configure software. Using tags, it is possible to specify which task runs.

```
tasks:
- name: Install
  yum:
    name:
    - httpd
    - memcached
    state: present
  tags:
  - packages
  - webservers
- name: Configure
  template:
    src: templates/src.j2
    dest: /etc/foo.conf
  tags:
  - configuration
```

Tags

This example shows the 'ntp' tag

```
- name: Install ntp
  yum:
    name: ntp
    state: present
  tags: ntp

- name: Install nslookup
  yum:
    name: nslookup
    state: present
```

If you ran these tasks in a playbook with `--tags ntp`, Ansible would run the one tagged `ntp` and skip the other.

Tags

Inheritance:
No one wants to add the same tag to multiple tasks. To avoid repeating code you can tag the play, block, or role. Ansible applies the tags down the dependency chain to all child tasks.

Blocks are useful for applying a tag to many, but not all, of the tasks in your play.

Plays are better suited if every task in the play should have the same tags.

Adding tags to roles allows you to run specific roles.

Tags

Define tags at the block level

```
- name: ntp tasks
  tags: ntp
  block:
  - name: Install ntp
    yum:
      name: ntp
      state: present

  - name: Enable and run ntp
    service:
      name: ntpd
      state: started
      enabled: yes
  tags: ntp

- name: Install utils
  yum:
    name: ntf-utils
    state: present
```

Ansible Galaxy

Ansible Collections

You can extend Ansible by adding custom modules or plugins. You can create them from scratch or copy existing ones for local use.

A simple way to share plugins and modules with your team or organization is by including them in a collection and publishing the collection on Ansible Galaxy.

Ansible Collections

Modules:

Modules are reusable, standalone scripts that can be used by the Ansible API, the `ansible` command, or the `ansible-playbook` command. Modules provide a defined interface. Each module accepts arguments and returns information to Ansible by printing a JSON string to stdout before exiting. Modules execute on the target system (usually that means on a remote system) in separate processes.

Plugins:

Plugins extend Ansible's core functionality and execute on the control node within the `/usr/bin/ansible` process. Plugins offer options and extensions for the core features of Ansible - transforming data, logging output, connecting to inventory, and more

Ansible Galaxy

Use ansible-galaxy to install collection or role

```
ansible-galaxy (collection|role) install
```

You can install from the community, or any .tar.gz file.

```
ansible-galaxy collection install azure.azcollection
```

Use new collection (full namespace, collection, `collections` element)

```
- name: Azure collection
  hosts: localhost
  collections:
    - azure.azcollection
  tasks:
    - azure_rm_storageaccount:
        resource_group: myRG
        name: myStorageAccount
        account_type: Standard_LRS
```

Ansible Galaxy

Use ansible-galaxy to install role

```
ansible-galaxy role install
```

You can install from the community, or any .tar.gz file.

```
ansible-galaxy role install weareinteractive.users
```

Use new role:

```
- name: Create user
  hosts: all
  roles:
    - weareinteractive.users
  vars:
    users:
    - username: newuser
      append: yes
      password: $6$paD7LIRYpWiv7
```

Lab: Ansible Roles