Automation Developer

TEKsystems Global Services

WORKFORCE DEVELOPMENT

Lab page

https://jruels.github.io/automation-dev/

Blocks

Example with Ansible Blocks

```
---
- name: Ansible Blocks
  hosts: server1
  gather_facts: false

  tasks:
    - block:
        - name: List usr directory
          command: "ls -l /usr/"

        - name: List root directory
          command: "ls -l /root"
      become: yes

    - name: List home directory
      command: "ls -l ~/"
```

Recovery

An additional benefit of using ansible blocks is to perform recovery operations.
If any of the tasks within a block fail, the playbook will exit.

With blocks, we can assign a `rescue` block that can contain a bunch of tasks. If any of the tasks within the block fail, the tasks from the recovery block will automatically be executed to perform clean-up activity.

Rescue Variables

Ansible provide a couple of variables for tasks in the `rescue` portion of a block:

- `ansible_failed_task`

  The task that returned 'failed' and triggered the rescue. For example, to get the name use `ansible_failed_task.name`

- `ansible_failed_result`

  The captured return result of the failed task that triggered the rescue. The same as registering the variable.

Rescue Block

Example with Rescue block

```
---
- name: Ansible Blocks
  hosts: server1
  gather_facts: false

  tasks:
    - block:
        - name: List home directory
          command: "ls -l ~/"

        - name: Failing intentionally
          command: "ls -l /tmp/not-home"
      rescue:
       - name: Rescue block (perform recovery)
         debug:
           msg: "Something broke! Cleaning up.."
```

Always Block

An `always` block will be called independent of the task execution status. It can be used to give a summary or perform additional tasks whether the block tasks fail or not.

Always Block

Example with Always block

```
---
- name: Ansible Blocks
  hosts: server1
  gather_facts: false

  tasks:
    - block:
        - name: List home directory
          command: "ls -l ~/"

        - name: Failing intentionally
          command: "ls -l /tmp/not-home"
      always:
       - name: This always executes
         debug:
           msg: "Can't stop me..."
```

Block Practical Example

Now that we've discussed how a block can be used, let's look at practical examples.

- Install, configure, and start a service
- Apply logic to all tasks in the block
- Enable error handling

Block Practical Example

Practical example (Install, configure, and start Apache

```
---
tasks:
  - name: Install, configure, start Apache
    block:
     - name: Install httpd and memcached
       ansible.builtin.yum:
         name:
         - httpd
         - memcached
         state: present
     - name: Apply config template
       ansible.builtin.template:
         src: templates/src.j2
         dest: /etc/template.conf

     - name: Start/enable service
       ansible.builtin.service:
         name: httpd
         state: started
         enabled: true
    when: ansible_facts['distribution'] == 'CentOS'
    become: true
    become_user: root
```

Block Practical Example

The `when` condition evaluated for all tasks in block.

Practical example (Install, configure, and start Apache

```
---
tasks:
  - name: Install, configure, start Apache
    block:
     - name: Install httpd and memcached
       ansible.builtin.yum:
         name:
         - httpd
         - memcached
         state: present
     - name: Apply config template
       ansible.builtin.template:
         src: templates/src.j2
         dest: /etc/template.conf

     - name: Start/enable service
       ansible.builtin.service:
         name: httpd
         state: started
         enabled: true
    when: ansible_facts['distribution'] == 'CentOS'
    become: true
    become_user: root
```

Block Rescue Practical Example

Practical example (Install nginx and capture failed task)

block_rescue.yml

```yaml
- block:
      - name: Install Nginx
        ansible.builtin.yum:
          name: nginx
          state: present

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

    rescue: # These tasks run only if there is a failure on the block
      - name: Show which task failed
        ansible.builtin.debug:
          msg: "{{ ansible_failed_task }}"
```

Block Rescue Task Status

If an error occurs in the block and the rescue task succeeds, Ansible reverts the failed status of the original task for the run and continues to run the play as if the original task had succeeded.

The rescued task is considered successful. However, Ansible still reports a failure in the playbook statistics.

Force Block Failure

You can force a block to fail even if the rescue task is successful:

fail_block.yaml

```yaml
- block:
    - name: Task 1
      ansible.builtin.debug:
        msg: "A task"

    - name: Task 2
      ansible.builtin.debug:
        cmd: "A second task"

  rescue:
  - name: A rescue task
    ansible.builtin.debug:
      msg: "Rescued"

  - name: Fail the block when rescued
    ansible.builtin.fail:
      msg: "A task failed, so failing the whole block"
```

Block Handlers

You can use blocks with `flush_handlers` in a rescue task to ensure that all handlers run even if an error occurs:

```
---
tasks:
  - name: Attempt graceful rollback
    block:
     - name: Print a message
       ansible.builtin.debug:
         msg: 'I execute normally'
       changed_when: yes
       notify: run me even after an error

     - name: Force a failure
       ansible.builtin.command: /bin/false
    rescue:
      - name: Make sure all handlers run
        meta: flush_handlers

handlers:
  - name: Run me even after an error
    ansible.builtin.debug:
      msg: 'This handler runs even on error'
```

Lab: Ansible error handling

Jinja2 Templates

Jinja2 templates are simple template files that store variables that can change from time to time. When Playbooks are executed, these variables get replaced by actual values defined in Ansible Playbooks. This way, templating offers an efficient and flexible solution to create or alter configuration file with ease.

Jinja2 Templates

A Jinja2 template file is a text file that contains variables that get evaluated and replaced by actual values upon runtime or code execution. In a Jinja2 template file, you will find the following tags:

`{{ }}` : These double curly braces are the widely used tags in a template file and they are used for embedding variables and ultimately printing their value during code execution.

`{% %}` : These are mostly used for control statements such as loops and if-else statements.

`{# #}` : These denote comments that describe a task.

Jinja2 Templates

In most cases, Jinja2 template files are used for creating files or replacing configuration files on servers.

Apart from that, you can perform conditional statements such as `loops` and `if-else` statements and transform the data using filters and so much more.

Template files have the `.j2` extension, implying that Jinja2 templating is in use.

Jinja2 Templates

A simple Jinja2 template example.

Hey guys! Apache webserver {{ version_number }} is running on {{ server }} Enjoy!

```
The variables are "{{ version_number }}"
and "{{ server }}"
```

Jinja2 Templates

When the playbook is executed, the variables in the template file are replaced with declared vars.

```
---
- hosts:
  vars:
    version_number: "2.3.52"
    server: "Ubuntu"
tasks:
  - name: Jinja 2 template example
    template:
      src: my_template.j2
      dest: /home/ansible/myfile.txt
```

Lab: Ansible templates

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

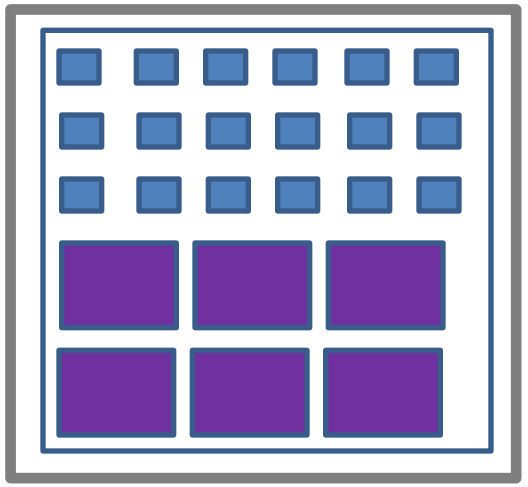
```
ansible-playbook --vault-password-file pass myfile.yml
```

Ansible Vault

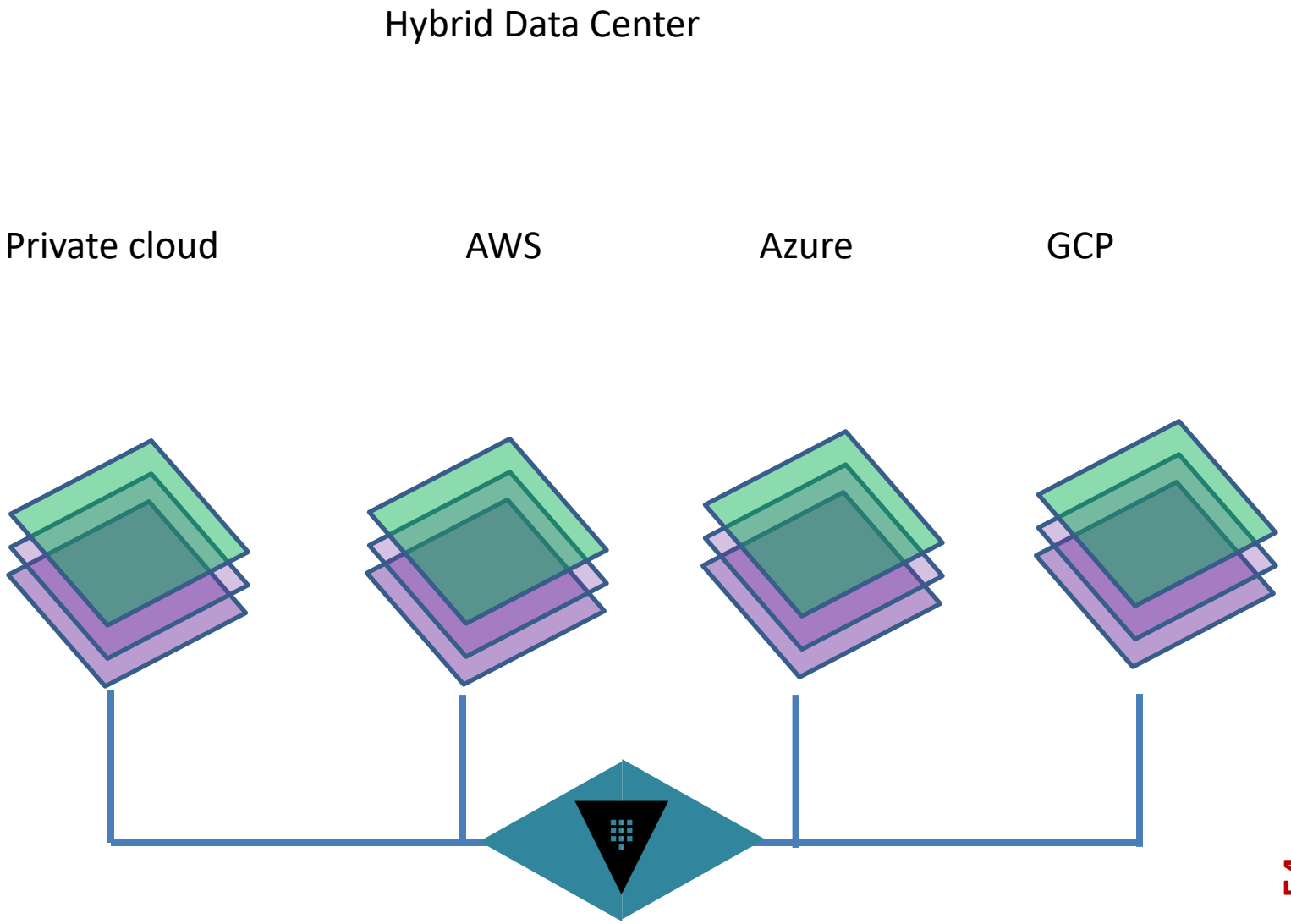
Common vault commands

```
# 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
```

Secure Infrastructure using Vault

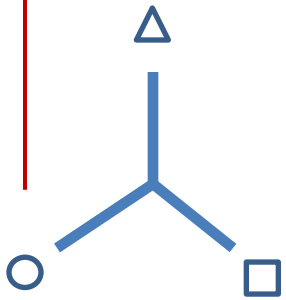


With each new cloud, network topologies become more complex.



Vault Objectives

- Provide single source of secrets for humans and machines .
- Scale to meet security needs of largest organizations.
- Allow for complete secret lifecycle management.



Eliminate Secret Sprawl

Securely Store any Secret

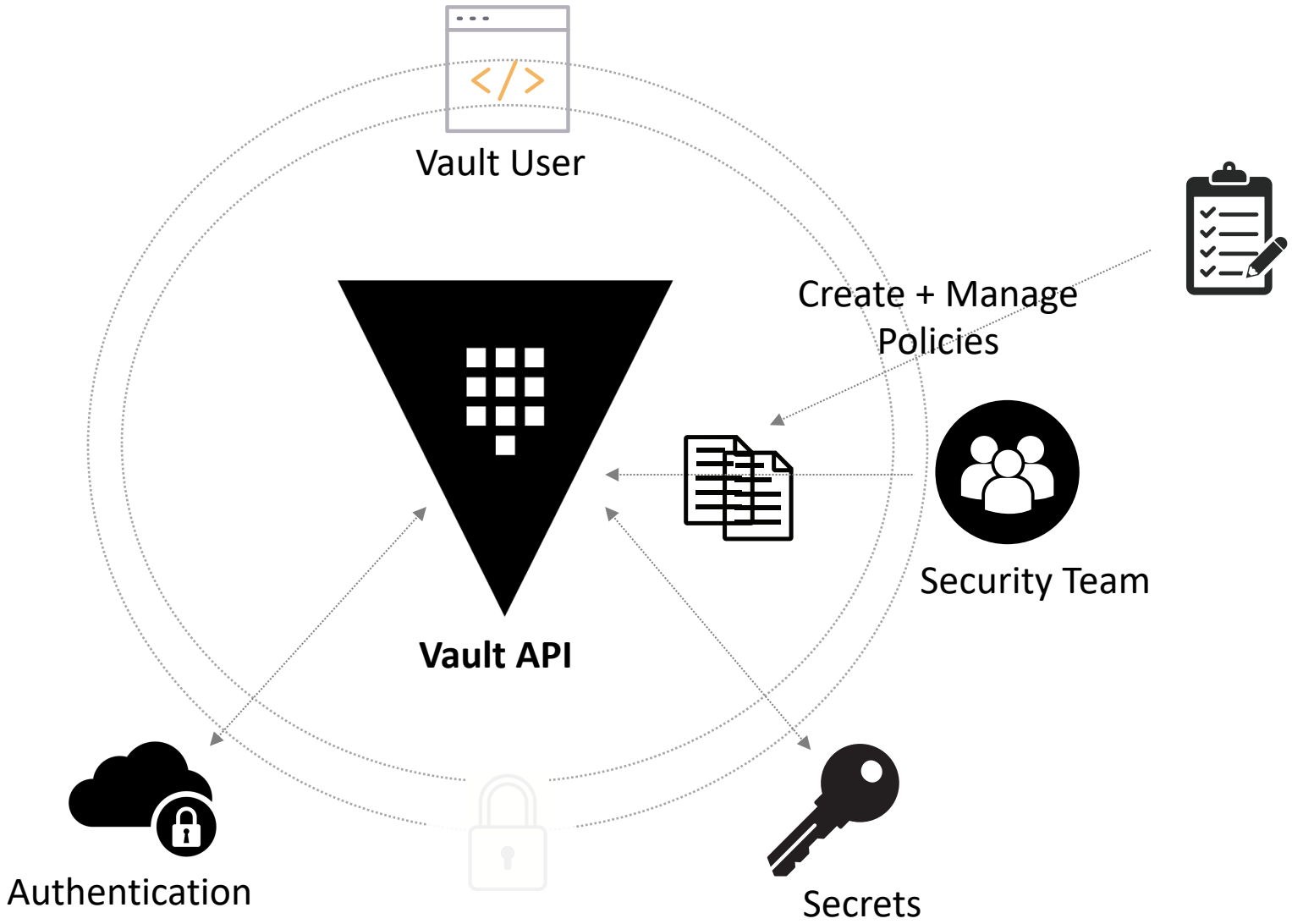
Secret Governance

Use Cases

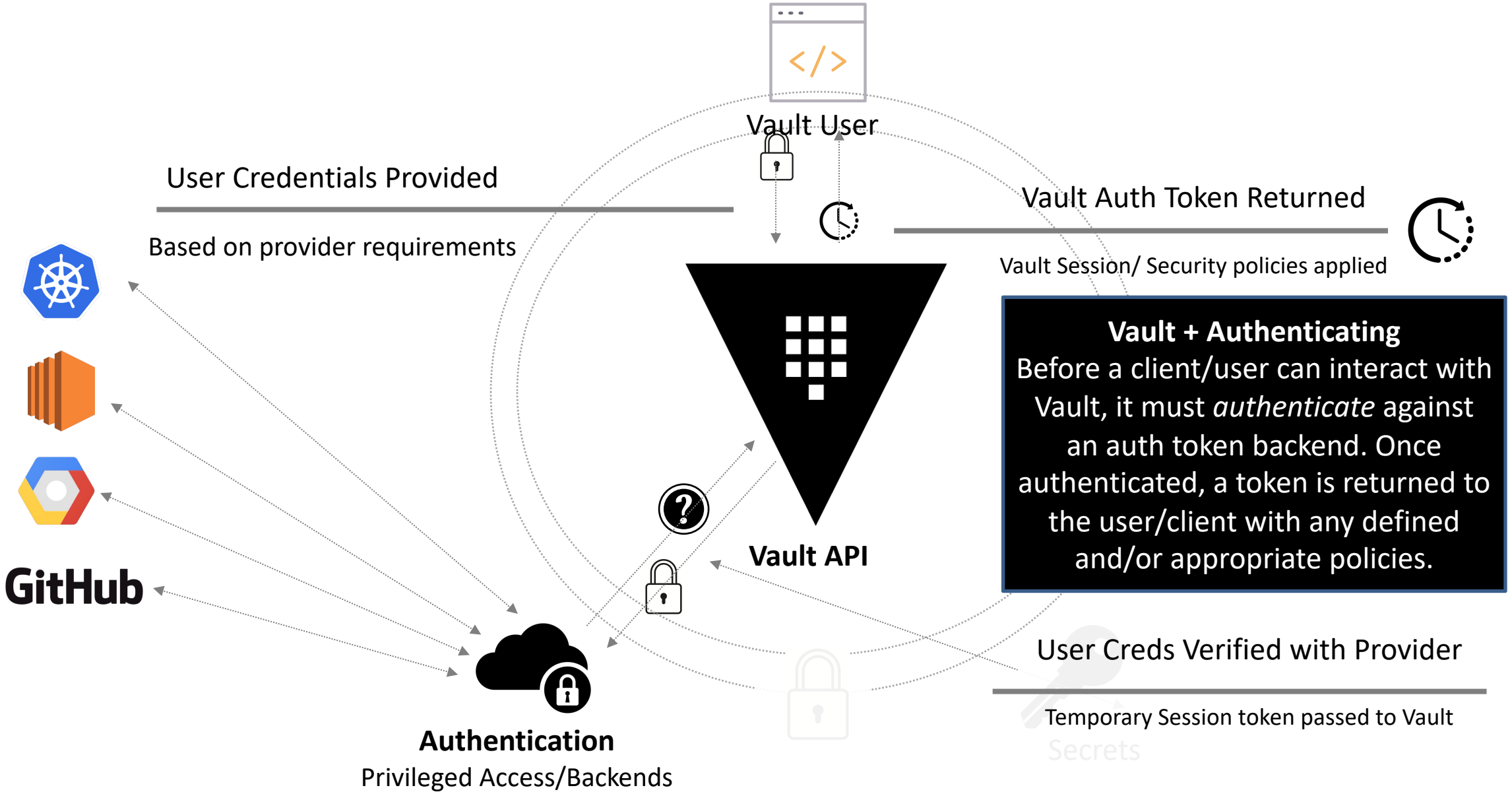
Secrets Management	Encryption as a Service	Identity Access Management
Secrets, identity, and access policy management workflow to secure any infrastructure and application resources.	One workflow to create and control the keys used to encrypt your data	Empower developers and operators to securely make application and infrastructure changes.

Vault - Security Policies

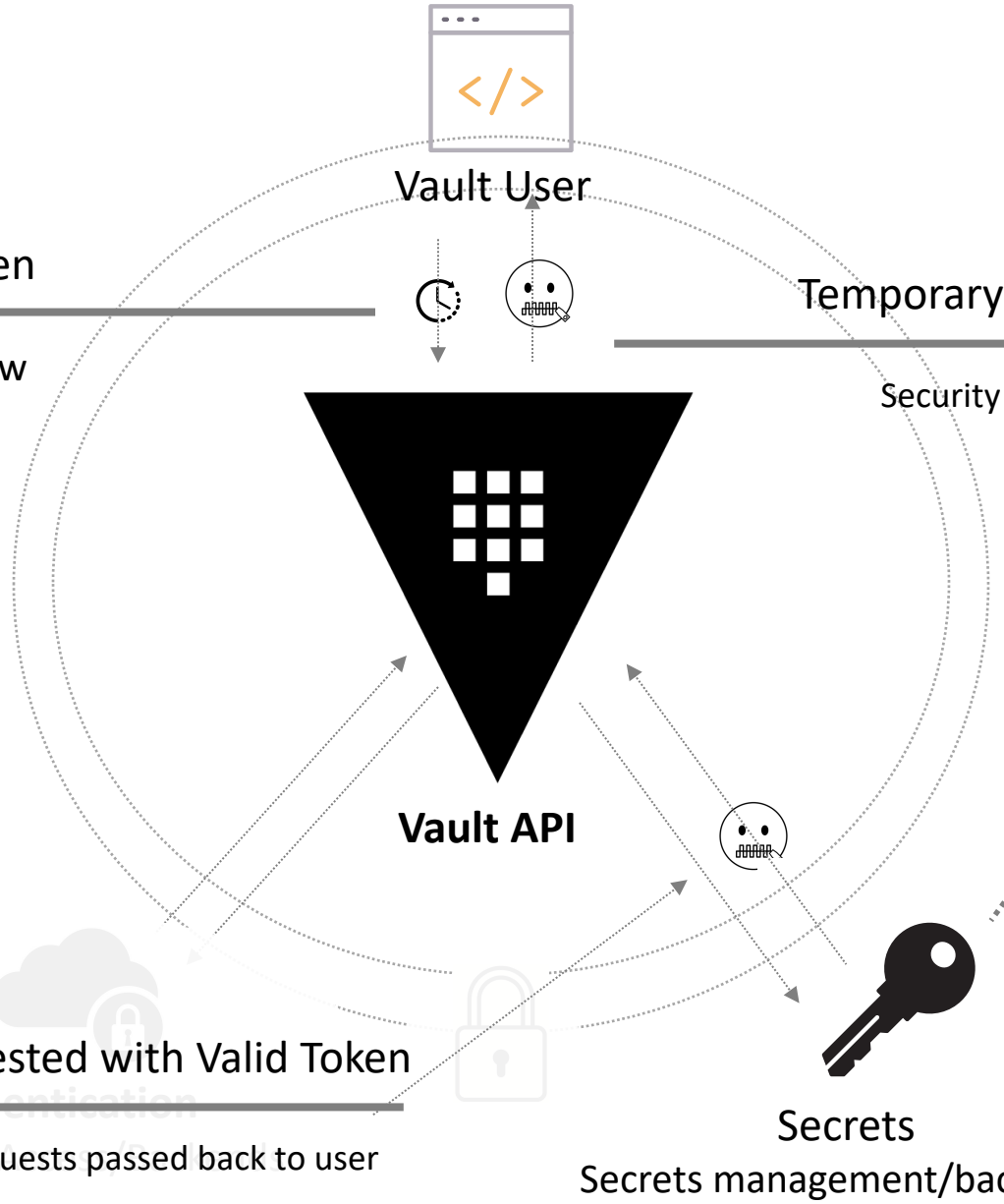
Policies with Vault
Vault uses policies to manage and safeguard access and secret distribution to applications and infrastructure. Policies provide a declarative way to **grant** and **deny** access to operations and paths.



Authentication Workflow



Secrets Workflow



Vault + Secrets

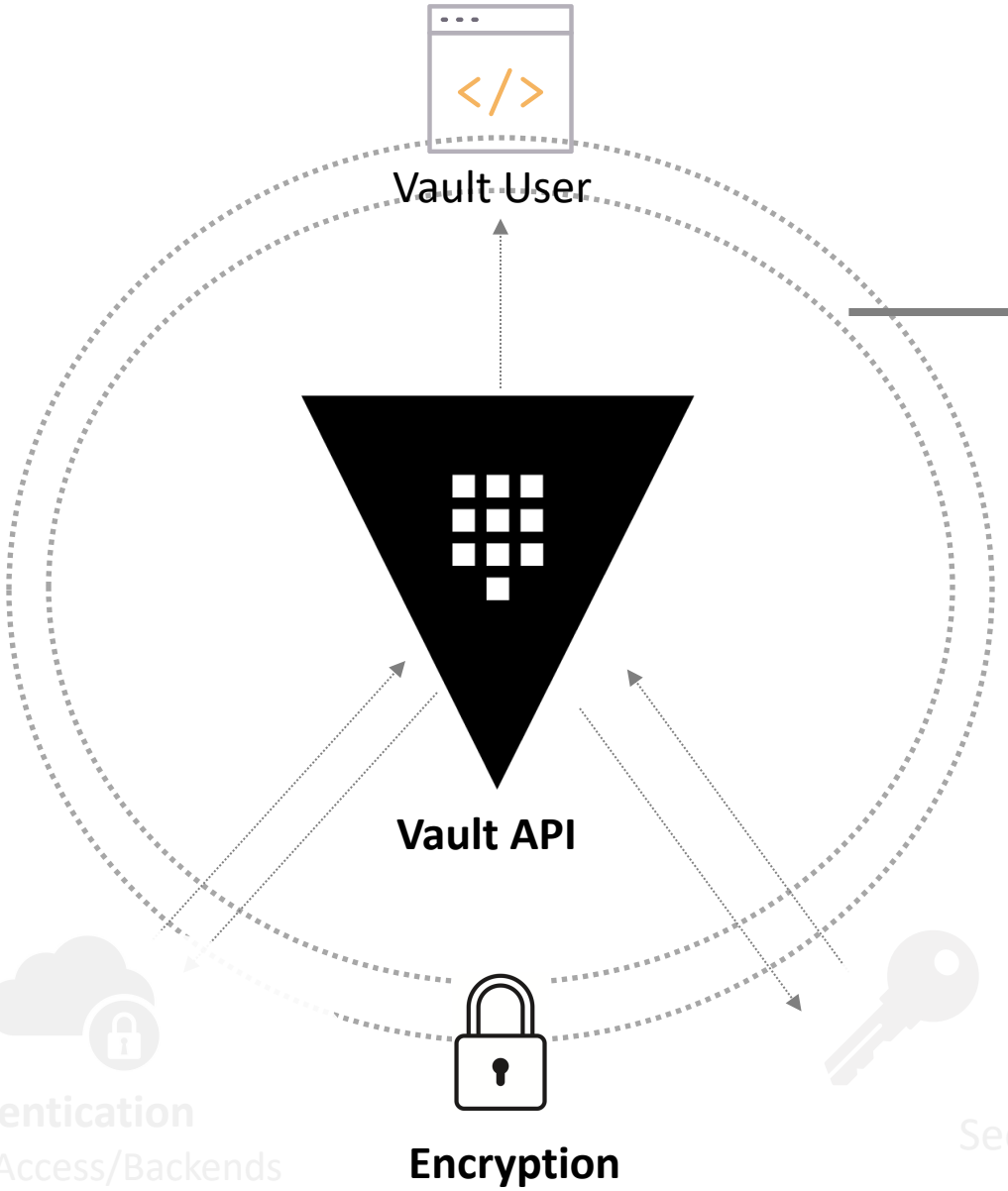
Authentication Token Required

Secrets can be stored and generated. Some secrets can be generated dynamically, while others are verbatim. Secrets are returned to the user/client with any defined and/or appropriate policies.

- Key/Value
- Databases
- Consul
- AWS
- RabbitMQ
- Custom plugins

Encryption(as a Service)

Encrypt everything
Vault believes that everything should always be encrypted. Vault uses ciphertext wrapping to encrypt all data at rest and in-flight. This minimizes exposure of secrets and sensitive information.



Full Data Encryption

Data is encrypted in-flight and at rest.

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

Developing a module

A module is a reusable, standalone script that Ansible runs on your behalf, either locally or remotely. Modules interact with your local machine, an API, or a remote system to perform specific tasks like changing a database password or spinning up a cloud instance.

```
- name: Install ntp
  package:
    name:
      - ntp
      - nslookup
    state: present
  tags: utils
```

Developing a module

A module provides a defined interface, accepts arguments, and returns information to Ansible by printing a JSON string to stdout before exiting.

module.yml

```
def main():

    fields = {
        "github_auth_key": {"required": True, "type": "str"},
        "username": {"required": True, "type": "str"},
        "name": {"required": True, "type": "str"},
        "description": {"required": False, "type": "str"},
        "private": {"default": False, "type": "bool"},
        "has_issues": {"default": True, "type": "bool"},
        "has_wiki": {"default": True, "type": "bool"},
        "has_downloads": {"default": True, "type": "bool"},
        "state": {
            "default": "present",
            "choices": ['present', 'absent'],
            "type": 'str'
        },
    }
```

Developing a module

To create a module:

1. Create a library directory in your workspace. Your test play should live in the same directory.
2. Create your new module file:
    - `library/my_test.py`.
3. Start writing your module.

Developing a module

All modules require documentation. Python docstrings should include examples of how to use the module.

module.yml

```
#!/usr/bin/python3

from __future__ import (absolute_import, division,
print_function)
__metaclass__ = type

DOCUMENTATION = r'''
---
module: github_repo

short_description: This module manages GitHub repositories
'''

EXAMPLES = r'''
- name: Create a github Repo
  github_repo:
    github_auth_key: "..."
    name: "Hello-World"
    description: "This is your first repository"
    private: yes
    has_issues: no
    has_wiki: no
    has_downloads: no
  register: result

- name: Delete that repo
  github_repo:
    github_auth_key: "..."
    name: "Hello-World"
    state: absent
  register: result
'''
```

Developing a module

This code defines a function that creates a new GitHub repository using the GitHub API. It extracts the GitHub authentication key from the provided data, sends a POST request to the /user/repos endpoint with the repository details.

The function returns different outcomes based on the response: success with changes (201), no changes due to an existing repository (422), or a default response indicating an error with the status code and response metadata for debugging.

module.yml

```
from ansible.module_utils.basic import *
import requests

api_url = "https://api.github.com"


def github_repo_present(data):

    api_key = data['github_auth_key']

    del data['state']
    del data['github_auth_key']

    headers = {
        "Authorization": "token {}" . format(api_key)
    }
    url = "{}{}" . format(api_url, '/user/repos')
    result = requests.post(url, json.dumps(data), headers=headers)

    if result.status_code == 201:
        return False, True, result.json()
    if result.status_code == 422:
        return False, False, result.json()

    # default: something went wrong
    meta = {"status": result.status_code, 'response': result.json()}
    return True, False, meta
```

Developing a module

This function deletes a specified GitHub repository by interacting with the GitHub API. It constructs the request URL using the repository's name and the user's username, authenticates using the provided GitHub token, and sends a DELETE request.

The function returns success with changes (204), no changes if the repository does not exist (404), or a default response indicating an error with the status code and response metadata for debugging.

module.yml

```
def github_repo_absent(data=None):
    headers = {
        "Authorization": "token {}" . format(data['github_auth_key'])
    }
    url = "{}/repos/{}/{}" . format(api_url, data['username'],
data['name'])
    result = requests.delete(url, headers=headers)

    if result.status_code == 204:
        return False, True, {"status": "SUCCESS"}
    if result.status_code == 404:
        result = {"status": result.status_code, "data":
result.json()}
        return False, False, result
    else:
        result = {"status": result.status_code, "data":
result.json()}
        return True, False, result
```

Developing a module

The main function serves as the entry point for an Ansible module to manage GitHub repositories. It defines a schema for input fields using fields, including parameters like github_auth_key, username, name, and other repository properties.

Based on the state parameter, it maps to either github_repo_present or github_repo_absent to create or delete a repository. The function processes the action, and based on the outcome, it either exits successfully with the result or fails with an error message, providing the necessary feedback to Ansible.

The if __name__ == '__main__': condition ensures that the code inside the block is executed only when the script is run directly. If the module is imported elsewhere, this block will not run.

It allows the module to define functions or classes that can be imported without executing the script's main logic.

module.yml

```yaml
def main():

    fields = {
        "github_auth_key": {"required": True, "type": "str"},
        "username": {"required": True, "type": "str"},
        "name": {"required": True, "type": "str"},
        "description": {"required": False, "type": "str"},
        "private": {"default": False, "type": "bool"},
        "has_issues": {"default": True, "type": "bool"},
        "has_wiki": {"default": True, "type": "bool"},
        "has_downloads": {"default": True, "type": "bool"},
        "state": {
            "default": "present",
            "choices": ['present', 'absent'],
            "type": 'str'
        },
    }

    choice_map = {
        "present": github_repo_present,
        "absent": github_repo_absent,
    }

    module = AnsibleModule(argument_spec=fields)
    is_error, has_changed, result = choice_map.get(
        module.params['state'])(module.params)

    if not is_error:
        module.exit_json(changed=has_changed, meta=result)
    else:
        module.fail_json(msg="Error deleting repo", meta=result)


if __name__ == '__main__':
    main()
```

Lab: Write a module