

Ansible intro

Jan Skalny, 2025-02-19

Outline

- What is ansible
- Basic concepts
- Writing our first playbook
- Roles, Collections, Galaxies and more

Why automate?

Why automate

- Efficiency, consistency and accuracy
- Reusability between projects/organizations
- Better release engineering / DevOps practices
- Infrastructure is a cattle, not a pet
- Microservices / alternative to docker
- Detect misconfigurations and configuration drift
- Faster disaster recovery

What is ansible

- open-source automation and configuration management tool
- agentless (... kinda)
- infrastructure as a code
- define once and run many times
- avoid misconfigurations and discover deviations
- ideal for hardening, auditing and verification
- (mostly) idempotent

YAML intermezzo

- white-space has a meaning
- objects, lists, strings, ...
- block scalars | (without new lines >)
- comments, begin/end doc, etc.
- transformable to/from JSON
- `yamllint` and `yq` are your friends

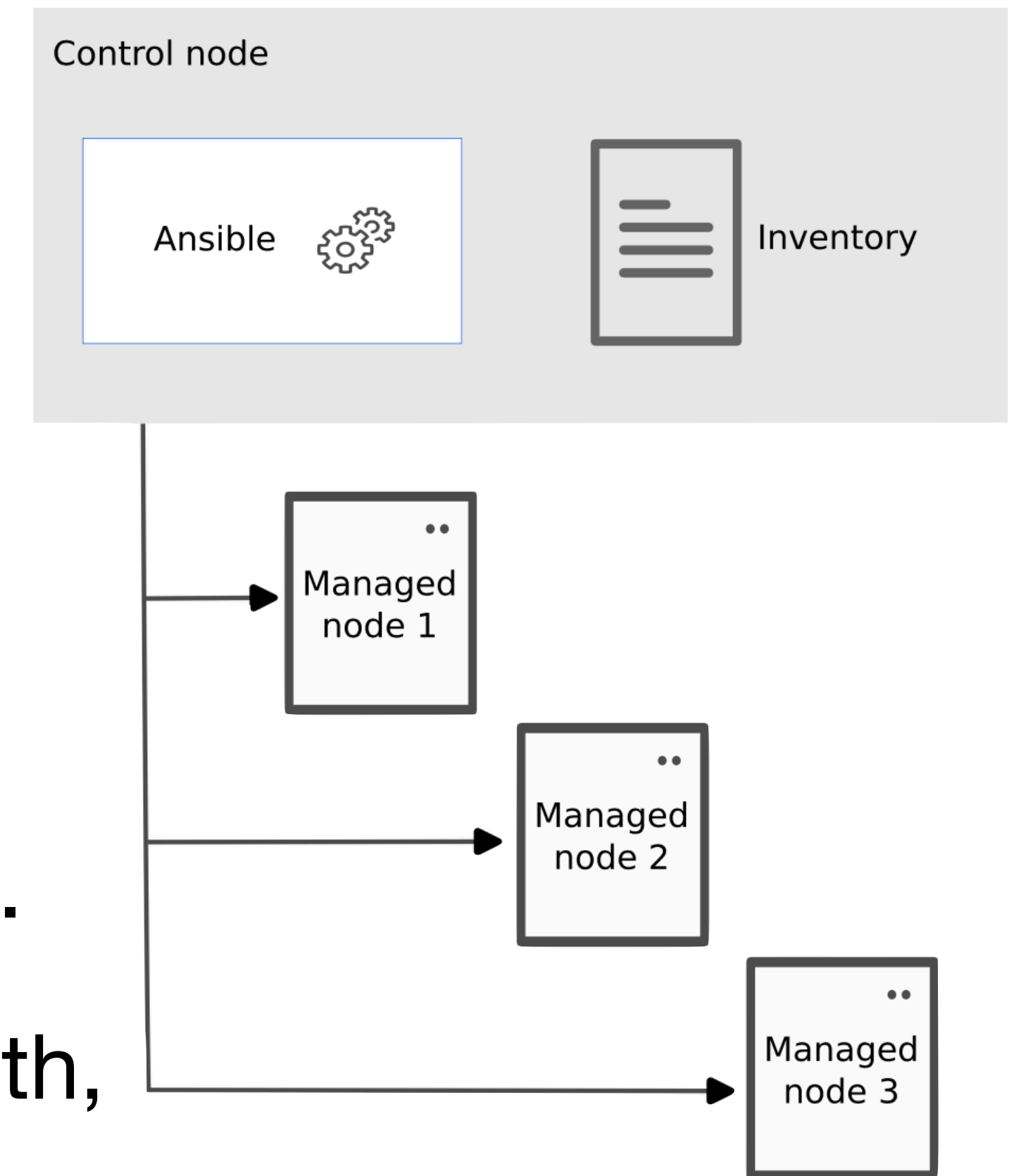
```
user_object:
  name: "user 1"
  password: "krokodil"
  groups:
    - users
    - admins
  description: |
    this is a sample user
    description spanning
    multiple lines...
  aliases: ["user1", "foo"]

example_val: 42
```

Ansible concepts

Ansible terminology

- **Control node**, where you run ansible tools,
... uses **inventory** to group **managed hosts**
... and define **variables** for each host.
- Executing **playbook** on managed host,
... connects to device using ssh,
... copies over and executes python executor,
... generated and defined by **ansible modules**,
... will put specific *object* on a host into desired state.
- Much of control node functionality can be tinkered with,
see `ansible.cfg` file and `ansible-config`



Modules, tasks and playbooks

- **Task**

- is single unit of work - ie. operation or desired state
- executes exactly one ansible module
- can succeed (either by changing something on remote system or doing nothing) or fail

“create directory named foo in /tmp”

```
file:  
  state: directory  
  path: /tmp/foo
```

```
ok=5  changed=2  unreachable=0  
failed=0  skipped=1  rescued=0  ignored=0
```

- **Playbooks**

- group, order and organize tasks into reusable and repeatable plays
- defines desired state for each matching host
- should be idempotent (re-running plays)
- Tasks are (usually) executed in sequence and synchronously on all matching hosts in parallel.
- Failure to execute specific task removes failing host from execution, but rest can continue.

```
check_mode: no  
changed_when: False
```

```
strategy, serial, throttle, run_once, ...
```

```
max_fail_percentage
```

More on playbooks

- Each **task**
 - is a YAML object with attributes
 - has exactly one **module**
 - can be named (via `name` attribute)
 - made conditional (via `when` attribute),
 - ignored when failed or run in check mode,
 - be iterated using `with_items` and `loop` attributes,
 - have custom variables defined (using `vars` attribute),
 - and much more.
- **Modules**
 - are also YAML objects,
 - implemented using specific plugin from a collection,
 - attributes are also called “parameters” to distinguish from task attributes.
 - see specific module docs for more.
- Troubleshooting typos using
 - generic yaml linters, eg. `yamllint`
 - `ansible-playbook --list-tasks` etc.
 - `ansible-lint`

```
- hosts: reverse_proxy
  environment: "{{ proxy_env }}"
  tasks:
    - name: install apache2
      package:
        name: apache2
        tags: install

    - name: enable apache2 modules
      apache2_module:
        name: "{{ item }}"
        state: present
      with_items:
        - ssl
        - proxy_http
        - headers
        - authnz_ldap
      tags: install
      notify: restart apache2

    - name: generate apache proxy conf
      copy:
        content: "..."
        dest: /etc/apache2/001-proxy.conf
      tags: configure
      notify: reload apache2

    - name: enable site
      shell: a2ensite 001-proxy.conf
      tags: install
```

Connectivity

- For unix/linux/windows systems, ansible uses ssh to upload and execute python script on target host.
- For networking devices, modules are executed locally and changes are made using ssh-cli/netconf/rest-api or in other platform specific way.
- Testing connectivity:
 1. test connectivity (and update `~/.ssh/known_hosts`)
`ssh -v test.demo-cert.sk`
 2. check connectivity using ping module
`ansible -m ping -i test.demo-cert.sk, all`
 3. if user name is different, use `~/.ssh/config` to define Host — User mapping,
or `ansible_ssh_user` variable
 4. preferably, use ssh-keys for authentication (`ssh-copy-id` or modify `~/.ssh/authorized_keys`),
 5. otherwise you need `ssh-askpass` with `ansible_ssh_pass` variable

SSH Intermezzo

- ssh-keygen, ssh-add and agent
- authorized_keys and ssh-copy-id
- known_hosts
- ssh_config
- ProxyJump
- and other...

Building inventory

- Group hosts by:
 - *What* - db, webservers, firewall, ...
 - *Where* - datacenter, region, network segment, ...
 - *When* - production, staging, test, ...
- Avoid hyphens and don't start with numbers
- YAML format is more flexible than INI
- Hosts can belong to multiple groups
- Groups can be nested
- By default under `/etc/ansible/hosts`, but can be modified using `ansible.cfg` "inventory" variable

Sample inventory.yml

```
www:
  hosts:
    bit.demo-cert.sk:
    dev.demo-cert.sk:
    test[1:4].demo-cert.sk:

cluster:
  children:
    cluster_compute:
      hosts:
        c[1:3].demo-cert.sk:
    cluster_storage:
      hosts:
        s[1:2].demo-cert.sk:

kvm:
  children:
    cluster:
  hosts:
    dev.demo-cert.sk:
```

Variables

- Same naming requirements as host groups - avoid hyphens, keywords, don't start with numerals, ...
- Can be defined in multiple sources
 - Inventory - within inventory file, group_vars, host_vars, etc.
 - Playbooks (and roles)
 - Runtime - from facts, command line, registered from previous tasks, ...
- Different data types available
 - Strings, Integers, Floats, Lists, Dictionaries, ...
 - Booleans with “truthy” and “falsely” values - (yes, 1, y, true, True, “True”, “yes”, “on”, 1.0, ...)
 - Referencing other variables (Jinja templates), evaluated at the time of use - “{{ my_var }}/config.ini”

Separating inventory variables

- Variables can be stored separately within `host_vars` and `group_vars` directories and subdirectories
- Each host (based on `inventory_hostname`) and group can:
 - either have single host/group specific file
eg. `host_vars/ (name) .yml`
 - or better - subdirectory with multiple files.
eg. `group_vars/ (name) / fw .yml`
- Special group “`all`” can be used to define common variables for all hosts

Sample inventory with variables

```
group_vars/all/dns.yml
```

```
dns_server: "1.1.1.1"
```

```
group_vars/internal/dns.yml
```

```
dns_server: "192.168.1.1"
```

```
dns_search: "demo-cert.sk"
```

```
host_vars/dev.demo-cert.sk/dns.yml
```

```
dns_server: "147.175.159.11"
```

```
dns_search: ~
```

```
host_vars/dev.demo-cert.sk/apache.yml
```

```
apache_sites:
```

- domain: www.demo-cert.sk
owner: www-demo
- domain: test.demo-cert.sk
owner: www-demo

Variable precedence (simplified)

- “... | default (...)” filters
- role defaults
- `group_var/all/...`
- **specific** `group_var/...`
- **specific** `host_var/...`
- overrides from playbook
- overrides from CLI `-e ...` argument
- ... and much more

Using variables

- Within task attributes, using jinja “{{ variable_name }}” convention
- As conditionals (ie. when), without any escaping or jinja
- With with_items and loop iterators
- Inside templates and jinja functions...
- Use `ansible-inventory --host ...` to check defined variables

Basic modules

- Create directories and links, delete files, change permissions — `file`
- Copy files from control node to managing host — `copy`
- Manage users and groups — `user`, `group`
- Install and update software — `package`, `apt`, ...

file module

- Tamper with remote files, links, directories etc.
- Ability to change permissions and owners/groups
 - Recursion works, but think of directories and permissions (+rX)
 - Beware of octal vs decimal permissions.
Either use letters or don't forget leading zero!

copy module

- Upload or download single local file to/from remote host ... or between two remote hosts
- Local path is relative to ansible root dir, but can be absolute path as well
- Permission/and owners similar to “`file`” module
- Instead of specifying source file, content can be in-lined with `content`:

user and group modules

- Manage accounts, groups and membership on remote os
- Can change shells, lock accounts, disable passwords for users, etc...
- To deploy `authorized_keys` file...
either use `authorized` parameter of `user` module,
or more advanced `authorized_key` module.
- Not really idempotent - deleting “unknown” accounts is more complex
- Windows needs more specific `win_user` module

https://docs.ansible.com/ansible/latest/collections/ansible/builtin/user_module.html

https://docs.ansible.com/ansible/latest/collections/ansible/builtin/group_module.html

https://docs.ansible.com/ansible/latest/collections/ansible/posix/authorized_key_module.html

Deleting unknown accounts can be a hassle

```
- hosts: all !unmanaged
tasks:
  - name: list existing users
    check_mode: no
    shell: "getent passwd | awk -F: '{ $2 > 1000 || $2 < 10000 {print $1} }'"
    register: existing_users

  - name: remove any not in existing_users
    user:
      name: "{{ item }}"
      remove: yes
      force: yes
      state: absent
    with_items: "{{ existing_users.stdout_lines }}"
    when: item not in admins and not in user_whitelist
```

SKIP ME!

... for now :)

Privilege escalation

- By default, ansible modules are executed in context of ssh user
- Different “become” methods are available using plugins
- Some require special permissions of remote host (wheel group, sudoers, ...), or additional ansible variables to execute (`ansible_become_password`), or can ask for passwords using “`--ask-become-pass`” argument.
- Configured either globally (see `ansible-config` and `ansible.cfg`) or per task/playbook via variables.

```
# ansible.cfg
[privilege_escalation]
become = yes
become_method = sudo
become_user = root
```

```
# playbooks/sample.yml
- name: install sw
  package:
    name: apache2
    state: present
  become: yes
  become_method: sudo
```

```
# group_vars/webservers/become.yml
become: yes
become_method: su
become_user: www-data
```

Check mode

- Playbook can be executed in “check” mode with `-C` argument
- Useful with “show differences” mode with `-D` argument and for auditing and verification purposes.
- Requires idempotent tasks and modules - not always the case
- Non-modifying tasks can be executed in check mode as well
 - `“check_mode: no”`
 - useful for dynamic variable registration (see later)
- Changed result of task can be modified
 - `“changed_when: False”`
 - using conditionals and jinja is possible
- Good practice to run playbooks with `‘-CD’` before executing some bigger play

Using limits

- Especially useful with bigger inventories or split staging/production enviros
- Applicable to:
 - `playbook` host selection
 - `ansible-playbook` command - to further limit playbook host selection
 - `ansible` command - to limit “`all`” statement
- All specified hosts and groups are added to set (OR)
- Ampersand “`&`” can be used to perform unions (AND), ie. select only hosts matching both groups.
- Exclamation mark “`!`” can be used to negate set (NOT). Super-useful with unions.
- Asterisk matches “`*`” and square bracket notation “`[1:9]`” also available.
- Watch out for special characters and escaping, especially CLI and `!`, `*`.

fix formatting!

```
web mail
web:mail
web,mail
```

```
web &staging
```

```
web !staging
```

```
\!unmanaged
```

```
srv-dmz-[1:3]-*.mng-dc.*
```

```
web\*.demo-cert.sk
```

Basic modules (cont)

- Generate config files using jinja — `template`
- Modifying files in place — `lineinfile`, `blockinfile`
- rsync-ing larger data sets — `synchronize`
- Executing raw shell commands — `shell`
- `service`, `sysctl`, `reboot`, ...
- And so much more...

template module

- Templating using jinja2 - better alternative to “copy: { content: ... }”
- Conditionals
`{% if ... %} ... {% endif %}`
- Iterators
`{% for x in ... %}
... {{ x.y }} ...
{% endfor %}`
- Default variables for undefined vars can be good practice
`{% if my_var | default("foo") == "bar" %} ... {% endif %}`
- White-space trimming possible using “-“, but maj. PITA. eg. “{% -“,
- Source file relative to ansible root dir or role dir (see later)
- Custom functions, filters, matching, variable registration, includes and much more!

Modifying existing files

- Alternative to generating whole file using template (jinja)
- Necessary for files managed by different roles/tasks and users
- Avoid if possible, since it's harder to maintain consistency and idempotency
- `lineinfile` and `blockinfile` module
- Match text using `regexp:` `insertbefore:` `insertafter:`
- Can change permissions as well

synchronize module

- Requires `rsync` installed on both managing and remote node(s)
- Possibility to run between nodes as well
- Compression, special fs flags and more/

shell module

- “Hammer” when something needs bashing :)
- Execute shell script on remote host and record results (`stdout`, `stderr`, `rc`)
- Always marked as “changed”
Use `changed_when` and `failed_when` to parse outputs using conditionals.
- Good for additional “facts” gathering
Don’t forget `changed_when: false` and `check_mode: no` to run in `-CD`.
- Avoid if not necessary, but often useful if module does not exists
 - `reboot: vs shell: shutdown -r now`
 - `docker_compose: vs shell: docker compose up -d`

Task iterators using `with_items`

- Tasks can be easily repeated using items iterator.
- Define `with_items` attribute with list for strings/objects/...
- Tasks gets executed for each item from the list, with new “`item`” variable defined.
- Beware of `include` and nested `with_items`.
Can cause variable confusion / overwrite.
Use `loop ... with` instead.

Task iterators using loop(s)

- Better version of `with_*` iterators.
- Define list of objects or strings using “loop” attribute
- Iterator variable can be renamed using attribute
`loop_control: { loop_var: my_var }`
- Results can be registered into “results” list variable

Registering variables

- When executed, modules produce results (success/fail, stdout, objects, etc.)
- These can be stored into host-specific variables using `register: attribute`, and later used in conditionals or templating.
- Variables are ephemeral and host specific
- If necessary for playbook, non-modifying tasks must be run in check-mode, using “`check_mode: no`”
- Alternatively some tasks can be skipped or ignored in check mode

Deleting unknown accounts can be a hassle

```
- hosts: all !unmanaged
tasks:
  - name: list existing users
    check_mode: no
    shell: "getent passwd | awk -F: '$3 > 1000 && $3 < 10000 {print $1}'"
    register: existing_users

  - name: remove any non-defined but existing users
    user:
      name: "{{ item }}"
      remove: yes
      force: yes
      state: absent
    with_items: "{{ existing_users.stdout_lines }}"
    when: item not in admins and not in user_whitelist
```

Facts and magic variables

- Special set of variables automatically defined at start of playbook execution, or when specifically requested using “gather_facts” module.
- Can be cached locally or disabled for faster execution
- Useful for conditional playbooks and roles (eg. platform specific configuration)
- Example:
`ansible -m gather_facts -i test.demo-cert.sk,`

Registering variables — `set_fact`

- Using `set_fact` module to define variable on the fly
- Transform dynamic or complex variables using jinja2 filters
- Troubleshooting can get more difficult, and reusability will suffer

```
- name: Extend cortex_analyzers with default configuration
  set_fact:
    cortex_analyzers: "{{ cortex_analyzers | default([]) + [ {
      'name': item.name,
      'rate': item.rate | default(None),
      'rateUnit': item.rateUnit | default(None),
      'jobCache': item.jobCache | default(cortex_default_jobCache),
      'jobTimeout': item.jobTimeout | default(cortex_default_jobTimeout),
      'configuration': cortex_default_configs | combine(item.configuration |
```

Registering variables — `include_vars`

- Dynamically using `include_vars` module
- Good for platform specific variable definition

```
- name: set distro-specific variables
  include_vars: '{{ item }}'
with_first_found:
  - '{{ ansible_os_family }}.yaml'
  - '{{ ansible_system }}.yaml'
  - default.yaml
```

Debugging

- Dumping host variables using:
`ansible-inventory --host xxx`
- Using debug module within playbooks,
referencing `vars` or `hostvars[...]` variables.
- CLI tools accept “`-v`” verbosity argument (multiple times to increase level).
- Modules can have `verbosity:` attribute set.
- Running in check mode and diff mode first `-CD`
- lint-ers are your friends, especially with YAML files

How to get ansible - easy way

- Use your OS-es native package manager, eg.
`brew install ansible`
`apt-get install ansible`
 - less control over ansible version you get :(
- Or, install python w/ pip and run
`pip3 install ansible`
 - dependencies are a hassle :(
- Or, use existing docker container. eg.
`docker run --rm --it willhallonline/ansible /bin/sh`
 - ... and suffer when mounting ssh auth socket from you MacOS host :(

Our lab environment

- `git clone https://github.com/JanSkalny/fiit-bos-ansible.git`
- `docker compose up -d`
- `docker run --work /root/ --it ansible bash`
- See README.md for more

Advanced concepts

More on playbooks

- Playbooks can contain multiple blocks of hosts+tasks
- Failing within one block DOES NOT remove host from next block of tasks.
- If something needs to be executed before main roles/tasks, use `pre_tasks`:
(important when mixing roles, see later)
- If something needs to be executed after main roles/tasks, use `post_tasks`:
(usefull for cleaning up after failed executions)
- For single-shot conditional tasks, use `handlers` and `notify` (see later).

sample playbook with multiple blocks, post_tasks
and handlers

Handlers

- Occasionally some tasks need to be executed:
 - conditionally (only when something is changed),
 - once (multiple tasks might need same after-action),
 - and at well defined time.

Example: restarting apache service after modifying files or installing new modules

- Tasks that need to run specific handler (or handlers), can use “`notify:`”
- For each notified item, `handler` must be defined:
 - either within playbook,
 - or in executed role.

Conditionals

- As task attributes:
 - `when`: to limit task execution
 - `changed_when`: to ignore or improve “changed” detection (especially useful with generic modules, such as `shell`)
 - `failed_when`: to mark tasks as failed depending on result / condition
- Combine `when` with `loop:(s)` for filtering of items.
- Conditional variable inclusion is also possible via `vars_files`
- Beware of conditional imports/includes (see later)

Roles

- Separate functionality and related templates, files, variables and handlers into shareable entity
- Included directly in playbook using `roles:` attribute, or by using `include_role` module (with `loop` and other goodies)
- Roles are loaded from collections, `roles/` directory or `role_paths` - in that order.
- When writing roles:
 - do define sensible defaults for most variables (eg. api versions, ports, generic db names, ...)
 - don't define defaults for variables that have to be supplied by user (eg. api_keys, urls, ...)
 - don't produce too generic roles (with many variables, supporting every possible platform, ...),
 - don't produce roles that take more time to configure, than it takes to reimplement them,
 - do write roles compatible with most ansible configurations (no `exocit depends`, `hash_beh.`),
 - do write documentation, readme, commented defaults, tests, etc..

Roles (cont.)

- Typical role structure:
 - `tasks/main.yml` main playbook - may include additional files
 - `defaults/main.yml` default variables
 - `handlers/main.yml` dictionary of all handlers used within role
 - `vars/main.yml` other vars used by role (good for platform specific stuff)
 - `templates/` and `files/` source directories for `template` and `copy` modules
 - `meta/main.yml` for role dependencies, author and version information, etc.
 - `README.md` :)
- Newer versions allow args validation using `meta/argument_specs.yml`

Tags

- Every task within play or role can be tagged with one or more tags
- Tags can be applied also to `block:` of tasks or whole playbook.
- `ansible-playbook` allows skipping (`--skip-tags`) or running only tasks (`-t`) with specific tags.
See `ansible-playbook --list-tags` and `--list-tasks`
- Beware of nested tasks (includes vs imports)!

Includes vs imports

- Import - evaluated at parse time (when starting play)
- Include - evaluated when executed (when task gets reached)
- Include is more suitable for looping, but has drawbacks:
 - `--list-tags` does not work,
 - `--start-at` does not work,
 - handlers from included roles/plays are not considered,
 - tags are applied only to “include”, not “included”, ...
- Read more later...
https://docs.ansible.com/ansible/2.9/user_guide/playbooks_reuse.html

Rolling deployment

- Limit execution to N hosts percents to keep services operational using `serial` attribute, as part of playbook
- Value can be also list, eg. first deploy to one node, then 2, then rest.
- Failed execution can be stopped using `max_fail_percentage`
- Alternatively use `throttle` and `run_once` attribute at task level, to limit parallel execution

```
- hosts: nameservers
  serial: 1
  tasks: ...
```

```
- hosts: nameservers
  serial: [ 1, 2, 10 ]
  max_fail_percentage: 10
  tasks: ...
```

```
- hosts: nameservers
  tasks:
    - foo:
        throttle: 1
```

Speeding up (re)execution

- Use `ansible-playbook --start-at="task name"` when developing playbook or when something breaks “somewhere in the middle”. But beware of dynamically defined variables.
- Limit execution to retry file using `-l @playbooks/xxx.retry` (must enable `retry_files_enabled` in `ansible.cfg`)
- Use tags for configuration only steps (see later)
- Use imports instead of includes
- SSH pipelining, forks, control paths, timeouts and retries
- Benchmark your network, ssh and for global “`become: yes`” consider using root.
- Mitogen?

Strategies and async

- Parallelize task execution using argument `strategy: free` (default is linear)
 - Harder to do rollouts and watch out for hidden dependencies in your env.
- Execution is typically limited by number of “forks”
- Tasks can be run in “async” mode, without keeping connections opened
 - Use `async` to limit execution time and `poll` to specify polling interval

```
- hosts: all
  tasks:
    - shell: /bin/sleep 15
      async: 45
      poll: 5
```

Solving chicken and egg problem

- Installing ansible dependencies using ansible?
- Initial playbook
 - gathering facts require working python3, so `gather_facts: no`
 - we can use `raw` module to run shell commands,
 - install python3 and dependencies,
 - (pre)modify sudoers,
 - and install `authorized_keys`...

```
- hosts: all
gather_facts: no
tasks:
- raw: apt-get install -y --force-yes -o "Acquire::http::Timeout=10" --no-upgrade python3-apt
- raw: pkg install -y python
- raw: yum install -y python
```

More modules

- ansible-core (built-in) modules
- Generic and platform specific modules
- Collection specific modules
- https://docs.ansible.com/ansible/latest/collections/index_module.html

Collections

- Group of vendor-specific roles, plugins, modules and playbooks
- Distributed via ansible galaxy, using

```
ansible-galaxy collection list
ansible-galaxy collection install paloaltonetworks.panos
ansible-galaxy collection install paloaltonetworks.panos --upgrade
```
- Use collections either by referencing full collection and module name, eg.
`dellemc.unity.volume:`
- Or using `collection:` attribute as part of playbook to make short names usable, eg.
 - `hosts: all`
`collections:`
 - `dellemc.unity``tasks:`
 - `volume: ...`

Galaxy roles

- Ready-made roles distributed using `ansible-galaxy role` command
- Beware of unmaintained, incompatible, too complex to use roles, as well as supply chain attacks
- HOWTO
 - find a role on galaxy.ansible.com and understand what it does
 - make sure you have `role_dir` defined in `ansible.cfg`
 - `ansible-galaxy role install foo`
 - review
 - use as regular role in playbooks
 - **use** `ansible-galaxy role install --upgrade` **regularly**
 - write your own role :D

ansible-vault

- At-rest encryption of whole files, for storage of sensitive data (passwords, API keys and other secrets) in public / cloud environments
- Use separate inventory files for encrypted secrets using:
`ansible-vault create (or) edit host_vars/.../secrets.yml`
- Password is either:
 - read from stdin (enforcable by using `--ask-vault-pass` switch)
 - read from `--vault-password-file` (`ansible.cfg vault_password_file`)
 - read from `ANSIBLE_VAULT_PASSWORD_FILE` env variable
 - read from stdout of executable, if vault file points to an executable

jinja hacks

- merging variables
- data transformation
- “do” macro
- custom functions
- see. https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_filters.html

How to get ansible - better way

- venv & pip
- docker
-



venv, pip, docker

To be continued...

- custom plugins and filters
- Ansible tower / AWX / semaphore
- dynamic inventories
- task delegation (delegate_to)
- ansible-console
- ...

Conclusions

Conclusions

ANSIBLE!!

