ANSIBLE

OVERVIEW

WHAT IS IT GOOD FOR?

- Provisioning and Configuration Management
- Application Deployment
- Orchestration
- Plays nicely with Docker and most Cloud Providers

So, it covers most of your needs for a typical web app

WHAT IS IT?

- An IT automation tool
- Written in `Python`
- Configured with templated `YAML` and fancy `INI-like` syntax
- Declarative approach
- Masterless, utilizes `Push` instead of `Pull` idiom
- Mostly targeted at *NIX, has limited Windows support
- Uses SSH (or Remote PowerShell for Windows) to communicate with managed hosts

PUSHING INSTEAD OF PULLING

Ansible relies on Push idiom: you push changes to managed nodes, instead of nodes pulling changes from a master.

- Greatly simplifies Orchestration
- Goes nicely with modern Continuous Integration pipelines.
- Doesn't scale quite as well (in the free version)

REQUIREMENTS

Host:

- Python 2.6+

Managed nodes (one of):

- Python 2.5
- Python 2.4 with `python-simplejson`
- Nothing for `raw` or `script` modules

RELATED RESOURCES

- Ansible Tower - Enterprise-scale Ansible
- Ansible Vault - Managing "secrets" with Ansible
- Ansible Galaxy - Collection of open-source roles

INSTALLATION

4.1

Ubuntu

```
sudo apt-get install software-properties-common
sudo apt-add-repository ppa:ansible/ansible
sudo apt-get update
sudo apt-get install ansible
```

Debian

```
echo "deb http://ppa.launchpad.net/ansible/ansible/ubuntu trus
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys
sudo apt-get update
sudo apt-get install ansible
```

RedHat, Centos, Fedora - Add EPEL repository (enable `testing` for newer versions)

```
yum install ansible
```

KEY CONCEPTS

- `Inventory` - a list of all managed hosts, possibly divided into groups
- `Module` - a unit responsible for configuring one specific system aspect (managing a file or service)
- `Task` - a configured application of Module
- `Role` - a collection of Tasks responsible for setting up a "logical unit" of configuration. The specificity is up to you: a "mysql" role can coexist with broader "database" role
- `Play` - a collection of Tasks and Roles applied to a set of hosts from inventory

INVENTORY FILE

The core part of every Ansible setup

Inventory files use an `INI-like` syntax to define managed hosts, optionally divided into groups

```
mail.example.com

[webservers]
web1.example.com
web2.example.com

[dbservers]
db1.example.com
db2.example.com
```

Feel free to use hostnames, ip addresses and specify non-default ports

```
example.com:22200
195.66.141.75
```

Numeric and Alphabetical ranges are supported

```
[webservers]
192.168.1.[1:255]
www[01:50].example.com

[databases]
db-[a:f].example.com
```

Groups may include other groups with special `:children` syntax

```
mail.example.com

[webservers]
web1.example.com
web2.example.com

[dbservers]
db1.example.com
db2.example.com

[appservers:children]
webservers
dbservers
```

Note, that two groups are always present:

- `all` - includes all hosts
- `ungrouped` - includes hosts without group

INVENTORY PARAMETERS

You may specify additional parameters to alter Ansible's default connection parameters or to create host aliases

```
loadbalancer    ansible_port=22200    ansible_host=192.168.0.2

localhost       ansible_connection=local

192.168.1.1     ansible_user=vagrant
```

Full list of inventory parameters

Parameters may be applied on per-group basis with `:vars` section

```
[webservers]
web1.example.com
web2.example.com

[webservers:vars]
ansible_user=maintenance
```

All non-reserved parameters will be passed for you to use later in playbooks as variables

```
[webservers]
web1.example.com    proxy_host=fallback.example.com
web2.example.com

[webservers:vars]
proxy_host=proxy.example.com
proxy_port=8081
```

Preferred way to define parameters is by splitting them into separate files:

```
├── inventory
├── group_vars
│   ├── all.yml                <- Vars for all hosts
│   ├── webserver.yml          <- Vars for all hosts on webserve
│   └── dbserver.yml           <- and so on
└── host_vars
    └── mail.example.com.yml <- vars only for this host
```

Note that this approach only supports `YAML` or `JSON` syntax for parameter definition

USING INVENTORY AND GROUPS

To specify which inventory file to use, supply the `-i` option

```
ansible -i inventory
```

You can also supply a host-matching pattern as a first unnamed parameter

```
ansible webservers -i inventory
ansible *.example.com -i inventory
```

Patterns are very flexible:

- Include hosts from both groups

```
webservers:dbservers
```

- Exclude hosts from another group

```
webservers:!old
```

- Intersect groups (use hosts present in both groups)

```
webservers:&old
```

Combine them

```
webservers:dbservers:&old:!provisioning
```

You can mix-and-match different types of patterns

```
*.example.com:&webservers
```

And use regular expressions

```
~(web|db).*\.example\.com
```

DYNAMIC INVENTORIES

If the inventory file is marked as executable, it will be treated as a python script which output will be used to construct actual inventory file.

For example, setting up Amazon EC2 inventory script

```
wget https://raw.githubusercontent.com/ansible/ansible/devel/c
chmod +x ec2.py

export AWS_ACCESS_KEY_ID='AK123'
export AWS_SECRET_ACCESS_KEY='abc123'

ansible -i ec2.py
```

More on inventory scripts

If a directory is supplied as inventory, all files in it will be treated as inventory files:

- Executable files as dynamic inventories
- Other files as static inventories
- `.orig`, `.bak`, `.ini`, `.cfg`, `.retry`, `.pyc`, `.pyo` will be ignored

```
└── inventories
    ├── host_vars
    │   └── ...
    ├── group_vars
    │   └── ...
    ├── ec2.py
    └── inventory
```

```
ansible -i ./inventories
```

Will result in Ansible using both static and dynamic inventories.

Host and Group variables will still work as expected.

MODULES

You can find a list of available modules online

OFFLINE DOCUMENTATION

List all modules with

```
ansible-doc -l
```

Look up documentation on specific module with

```
ansible-doc <module>
```

AD-HOC COMMANDS

Having only inventory file, you may already run modules across your hosts using

```
ansible [...] -m <module> -a "<params>"
```

For example:

- Pinging all hosts

```
ansible -i inventory -m ping
```

- Get uptime for dbservers group

```
ansible webservers -i inventory -m command -a "uptime"
```

- Restart httpd service on all hosts in webservers group

```
ansible webservers -i inventory -m service -a "name=httpd state
```

YAML

Yet Another Markup Language

8.1

YAML is used to represent structured data and consists of:

- Basic types: strings, numbers, booleans, null
- Arrays
- Hashes (also known as dictionaries)

STRINGS

Strings are usually written as-is, even with spaces

```
string value
```

But have to be quoted if they contain control-characters, like ":", "{" or "["

```
"string[value]"
```

ARRAYS

Short syntax - strings are always quoted

```
['value1', 'value2', 'value3']
```

Expanded syntax

```
- value1
- value2
- value3
```

HASHES

Keys are always strings, values can be of arbitrary type

Short syntax

```
{key1: value1, key2: value2}
```

Expanded syntax

```
key1: value1
key2: value2
```

COMBINING EXPANDED SYNTAX

Nested expanded syntax is denoted with a tab.

Array in hash

```
hashKey: hashValue
hashArray:
  - arrayItem1
  - arrayItem2
anotherHashKey: hashValue
```

Hash as element of an array

```
- arrayItem1
-
  hashKey: hashValue
  hashKey2: hashValue2
- arrayItem3
```

SUMMING UP

Every YAML file starts with a `---`.

So, a full example would be

```yaml
---
typicalKey: string value
anotherKey: "complex:string{value}"
expandedHash:
  hashKey1: hashValue1
  nestedHash:
    nestedArray:
      - stringValue
      -
        anotherHashKey: anotherHashValue
        anotherHashKey2: anotherHashValue2
      - ['array inside array']
  hashKey2: yet another string value
shortHash: {hashKey1: hashValue1, nestedArray: ['0', '1', '2']
```

PLAYBOOKS

Setting things in motion

A single play is defined by:

- Pattern of hosts
- Tasks and Roles to apply

For example

```
---
- hosts: webservers
  tasks:
    - name: ensure httpd is at the latest version
      yum:
        name: httpd
        state: latest
    - name: ensure httpd is running (and enable it at boot)
      service:
        name: httpd
        state: started
        enabled: yes
```

`Playbook` is a set of `plays` potentially targeting different host groups.

You are encouraged to have more than one playbook -- as many as needed. You may have a configuration playbook, a deployment playbook, separate them by roles, etc.

A playbook may contain one or more plays

```
- hosts: webservers
  tasks:
    - name: ensure httpd is at the latest version
      yum:
        name: httpd
        state: latest
    - name: ensure httpd is running (and enable it at boot)
      service:
        name: httpd
        state: started
        enabled: yes

- hosts: dbservers
  tasks:
    - name: ensure mysql is at the latest version
```

To test playbook syntax, run

```
ansible-playbook playbook.yml --syntax-check
```

To apply a playbook, run

```
ansible-playbook playbook.yml -i inventory
```

`ansible-playbook` accepts most of parameters `ansible` does

VARIABLES

USING VARIABLES

Both playbook and template files are processed by Python's `Jinja2` templating system.

To use a variable, you surround it with "`{{  }}`":

```
"{{ var_name }}"
```

For example, we have a `package_name` variable defined in `group_vars`

```
# ./group_vars/dbservers.yml
package_name: mysql
```

Then, you can include it into your plays and tasks

```
# ./playbook.yml
- hosts: dbservers
  tasks:
    - name: ensure mysql is at the latest version
      yum:
        name: "{{package_name}}"
        state: latest
    - name: ensure mysql is running (and enable it at boot)
      service:
        name: "{{package_name}}"
        state: started
        enabled: yes
```

Accessing nested arrays or dictionaries properties is supported like in Python

```
# ./group_vars/dbservers.yml
package:
  name: mysql
  versions: ['5.5', '5.6']
```

```
"{{package.name}} - {{package.versions[0]}}"
```

Variables are separated in following categories:

- Predefined variables (host_vars, group_vars, role defaults, play and task variables, etc.)
- Facts (hosts information) gathered by `setup` module
- Dynamic variables that were defined during task execution

Variable precedence (last has the most priority):

- Role defaults
- group_vars
- host_vars
- Facts
- Play variables
- Role variables
- Task variables
- Registered (dynamic) variables
- Extra vars

You may additionally pass variables from command line

```
ansible-playbook playbook.yml -i inventory --extra-vars="deplo
```

You may define variables on task and play levels to save some typing or as preparation to branching it into a role

```
- hosts: dbservers
  vars:
    package_name: mysql
  tasks:
    - name: ensure mysql is at the latest version
      yum:
        name: "{{package_name}}"
        state: latest
    ...
```

FACTS

Ansible facts are provided by `setup` module, so you can easily check the full list with

```
ansible localhost --connection=local -m setup
```

The best part is, facts of all hosts acting in current play are always available as `hostvars`

FILTERS

Ansible provides a lot of useful filters to process your variables in-place.

You can apply filter by "piping" a variable into it, just like in bash.

```
"{{variable | filter(param) | anotherFilter}}"
```

For example

- Setting a default value if variable is not defined

```
"{{package_name | default(mysql)}}"
```

- Hashing a string

```
"{{password | hash(sha1)}}"
```

- Extract ip addresses of hosts in dbservers group

```
"{{groups['dbservers'] | map('extract', hostvars, ['ansible_de
```

LOOPS

A task can be run multiple times with items from an array using `with_items`

```
name: ensure packages are installed
yum:
  name: "{{item}}"
  state: latest
with_items:
  - vim
  - wget
  - screen
```

Or with items from dictionary using `with_dict`

```
name: ensure packages are in required state
vars:
  packages:
    vim: latest
    wget: present
    screen: absent
yum:
  name: "{{item.key}}"
  state: "{{item.value}}"
with_dict: "{{ packages }}"
```

CONDITIONS

Use `when` paired with Python boolean expression to run task conditionally

```
tasks:
  - name: "Use apt for debian"
    apt:
      name: nginx
      state: present
    when: ansible_os_family == "Debian"
  - name: "Use yum for CentOS"
    yum:
      name: nginx
      state: present
    when: ansible_os_family == "CentOS"
```

`when` can also be used to filter out `with_items` or `with_dict`

```
name: "Use apt for debian"
command: "echo {{ item }}"
with_items: [0, 1, 2, 3, 4, 5]
when: item > 3
```

HANDLERS

Handlers are special tasks, that are triggered by other tasks using `notify`.

For example, restarting apache after its configuration has changed

```
---
- hosts: webservers
  tasks:
  - name: write the apache config file
    template:
      src: /srv/httpd.j2
      dest: /etc/httpd.conf
    notify:
    - restart apache
  handlers:
    - name: restart apache
      service:
        name: httpd
        state: restarted
```

- Handlers never run unless notified
- Handlers run only once if notified multiple times
- Handlers are run in the very end of the play

BECOME

`become` is Ansible's `sudo/su`

`become` can be used to temporarily switch to another user at global, play, role or task levels.

For example:

- You need root to manage apache
- You need to manage php application with ssh user

```
---
- hosts: webservers
  become: true
  tasks:
  - name: write the apache config file
    template:
      src: /srv/httpd.j2
      dest: /etc/httpd.conf
    notify:
    - restart apache
  - name: clear php app cache
    become: false
    command: "/var/www/app/bin/console cache:clear"
  handlers:
```

REGISTER

Capturing task output as a variable

For example, there is a clever tool that outputs path to its configuration in JSON format and you need to manage this configuration. Essentially, you need to "lift" data that tool outputs into your play.

The tool outputs something along the lines

```
{"configuration_path": "/etc/tool/revision-hash/202cb962ac590
```

To capture its output into variable, use register

```
tasks:
  - command: /bin/tool --config
    register: tool_output
  - name: write the tool config file
    template:
      src: /srv/tool.j2
      dest: "{{tool_output.stdout | from_json | json_query('c
```

Docs on what you can find in registered result

ERROR HANDLING

By default, Ansible can tell if a task has failed by looking up the process exit code

ERROR CONDITIONS

However, you may sometimes want to define your own error conditions

For example, another tool prints "FAILED" when it fails, but returns a successful exit code

```
- name: this command prints FAILED when it fails
  command: /usr/bin/example-command -x -y -z
  register: command_result
  failed_when: "'FAILED' in command_result.stderr"
```

Or, a tool reports failure exit code on successful run:

```
- name: this command prints SUCCESS when it succeeds
  command: /usr/bin/example-command -x -y -z
  register: command_result
  ignore_errors: true
  failed_when: "'SUCCESS' not in command_result.stdout"
```

ABORTING THE PLAY

You can abort the play from task using `fail` syntax

```
- name: Unexpected Error
  fail: msg="Something went wrong"
  when: result|failed
```

Or, you can abort the play on any error by default using

```
- hosts: somehosts
  any_errors_fatal: true
  tasks:
    ...
```

FALLBACK STEPS

Now, you may actually handle the error. For example, enabling a new apache virtual host with rollback steps if configuration is somehow invalid.

```
- hosts: webservers
  tasks:
    - name: Push future default virtual host configuration
      copy: src=files/awesome-app dest=/etc/apache2/sites-ava

    - name: Activates our virtualhost
      command: a2ensite awesome-app

  - name: Check that our config is valid
    command: apache2ctl configtest
    register: result
    ignore_errors: true

  - name: Rolling back - Restoring old default virtualhost
    command: a2ensite default
```

SERIAL

Serial allows you to define batches in which play should be executed

It accepts a number, a percentage value or an array of those.

For example, updating one, then 10% of servers left each batch

```
- hosts: webservers
  serial:
   - 1
   - 10%
```

You may also set maximum failed hosts threshold

```
- hosts: webservers
  max_fail_percentage: 5
  serial: 10
```

RUN ONCE

A task can be marked as run once.

```yaml
---
# ...
  tasks:
    # ...
    - command: /var/www/app/bin/console migrations:migrate
      run_once: true
    # ...
```

However, together with `serial` task will be executed for each batch. Use when together with hostname to overcome this.

```
---
# ...
  tasks:
    # ...
    - command: /var/www/app/bin/console migrations:migrate
      when: inventory_hostname == webservers[0]
# ...
```

DELEGATION

A host may delegate action to another host

```
- hosts: webservers
  serial: 5
  tasks:
  - name: take out of load balancer pool
    command: /usr/bin/take_out_of_pool {{ inventory_hostname }}
    delegate_to: groups['loadbalancers'][0]
  - name: actual steps would go here
    yum: name=acme-web-stack state=latest
  - name: add back to load balancer pool
    command: /usr/bin/add_back_to_pool {{ inventory_hostname }}
    delegate_to: groups['loadbalancers'][0]
```

You may also use `local_action` for command machine itself

```
- name: take out of load balancer pool
  delegate_local: command /usr/bin/take_out_of_pool {{ inver
```

DELEGATED FACTS

Delegation may also be used to collect facts about other hosts

```
- hosts: app_servers
  tasks:
    - name: gather facts from db servers
      setup:
      delegate_to: "{{item}}"
      delegate_facts: true
      with_items: "{{groups['dbservers']}}"
```

This way you can access facts of dbservers even when they were not part of the play

STRATEGIES

Ansible supports different strategies for executing plays

- linear (default): all hosts in a batch wait for every step to be finished by other hosts before moving to next step
- serial: all hosts in a batch run tasks as fast as they can and wait for last host before moving to next batch
- free: all hosts in a batch run tasks as fast as they can

You can specify strategy on per-play basis

```
- hosts: all
  strategy: free
  tasks:
  ...
```

There's also one special debugger strategy to help you in developing.

ROLES

Reusable logical units

ROLE STRUCTURE

```
.
└── my_new_role
    ├── README.md
    ├── defaults       <- Role configuration interface
    │   └── main.yml
    ├── files          <- static files
    ├── handlers       <- post run commands, notified by tasks
    │   └── main.yml
    ├── meta           <- informations about role. Author, depe
    │   └── main.yml
    ├── tasks          <- Main part of the role
    │   └── main.yml
    ├── templates      <- Jinja2 file templates
    ├── tests          <- Tests for role
    │   ├── inventory
```

You can safely delete any of the unused parts

Use `ansible-galaxy` to bootstrap a new role

```
ansible-galaxy init --init-path=roles/ --offline my_new_role
```

defaults:

- Configuration, exposed to the user
- Some kind of parameters, you can use for the role

vars:

- Configuration inside the role
- Typically environment specific and loaded by facts

IMPORTING ROLES

Ansible Galaxy has a vast selection of roles

When using imported roles, always create a requirements.yml file to list them

Roles may be imported by name or from git

```
# requirements.yml
- geerlingguy.jenkins # short name for roles from ansible gala

- src: https://github.com/bennojoy/nginx
  version: master
  name: nginx
```

Always have a separate directory for imported roles and configure it

```
# ansible.cfg
[defaults]
roles_path = ./galaxy_roles:./roles
```

Finally, fetch the roles using `ansible-galaxy`

```
ansible-galaxy install -r=requirements.yml -p=galaxy_roles
```

INCLUDES

Code reusage is achieved though gradual inclusion of smaller files.

You can include:

- Playbooks into playbooks
- Roles into play
- Tasklists into role and play
- Vars into everything above

INCLUDING PLAYBOOKS

Playbook includes are listed at the root of playbook and may be mixed with definition of play

```
- include: initplays.yml

- hosts: localhost
  tasks:
    ...

- include: otherplays.yml
```

INCLUDING ROLES

Roles can be included with `role` syntax, optionally overriding role default parameters

```
- hosts: webservers
  roles:
    - common

    - role: test_app_a
      app_dir: '/opt/a'
      app_port: 5000

    - role: test_app_b
      dir: '/opt/b'
      app_port: 5001
```

TASKLISTS

A Tasklist is a .yml containing one or more tasks

```yaml
# service_task.yml
---
- name: install service
  apt:
    name: "{{service_name}}"
    state: present
- name: enable service
  service:
    name: "{{service_name}}"
    state: started
    enabled: yes
```

Tasklists can also be included optionally overriding their parameters

```
- hosts: webservers
  tasks:
    - include: service_task.yml
      vars:
        service_name: my_service
```

Tasklist includes can also be looped

```
- hosts: webservers
  tasks:
    - include: service_task.yml service_name={{item}}
      with_items:
        - nginx
        - mysql
```

VARS

Vars can be included anywhere with `include_vars` module

For example

```
# vars/services.yml
---
- nginx
- mysql
```

```
- hosts: webservers
  include_vars:
    file: vars/services.yml
    name: services
  tasks:
    - include: service_task.yml service_name={{item}}
      with_items: "{{services}}"
```

ARCHITECTURE

Examples of structuring your Ansible site

```
├── production          # inventory file for production servers
├── staging             # inventory file for staging environment
│
├── group_vars
│   ├── group1
│   └── group2
├── host_vars
│   ├── hostname1
│   └── hostname2
│
├── site.yml            # master playbook
├── webservers.yml      # playbook for webserver tier
├── dbservers.yml       # playbook for dbserver tier
│
└── roles
```

```
├── inventories
│   ├── production
│   │   ├── hosts
│   │   ├── group_vars
│   │   │   ├── group1
│   │   │   └── group2
│   │   └── host_vars
│   │       ├── hostname1
│   │       └── hostname2
│   │
│   └── staging
│       ├── hosts
│       ├── group_vars
│       │   ├── group1
│       │   └── group2
```

Note that vars includes and tasklist into play includes are heavily discouraged.

It's okay to use them during development, but be sure to move them into a role.

PRACTICE

Roll out an ansible playbook with:

- Nginx as front-tier cache
- Apache with php-fpm
- Mysql
- A default WordPress installation
- UFW Firewall

WHAT TO GO FOR

Ansible Docs

Skim through modules list, there are a lot of unexpected pleasantries

Learn about typical deployment paths for:

- JavaEE
- Node.js
- Python
- Ruby
- PHP
- Golang

Learn about logging and monitoring solutions:

- ELK Stack
- Nagios
- Grafana + Zabbix

Learn about what DevOps became for industry leaders:

Site Reliability Engineering