**Ansible**

**Ansible Up and Running**

Ansible is simple, and that is the best part. Eliminating management daemons and relying instead on OpenSSH meant the system could start managing a computer fleet immediately, without having to set up anything on the managed machines. Further, the system was apt to be more reliable and secure You can wire up these services by hand: spinning up the servers you need, SSHing to each one, installing packages, editing config files, and so forth, but it’s a pain. It’s time-consuming, error-prone, and just plain dull to do this kind of work manually, especially around the third or fourth time. And for more complex tasks, like standing up an OpenStack cloud inside your application, doing it by hand is madness. There’s a better way.

**What’s in the name?**

An ansible is a fictional communication device that can transfer information faster than the speed of light.

Ansible is a great tool for deployment as well as configuration management. It can orchestrate deployment (have to order on actions) and also provides infrastructure.

**How Ansible Works**

In Ansible, a script is called a playbook. A playbook describes which hosts (what Ansible calls remote servers) to configure, and an ordered list of tasks to perform on those hosts.

Run ansible-playbook with $ ansible-playbook webservers.yml Ansible will make parallel SSH connections to the hosts.

When you use:

- name: Install nginx

apt: name=nginx

Ansible will do the following:

1. Generate a Python script that installs the Nginx package
2. Copy the script to web1, web2, and web3
3. Execute the script on web1, web2, and web3
4. Wait for the script to complete execution on all hosts

For each task, Ansible will generate a Python script and executes it in parallel on all the hosts.

Ansible is push-based, that is you run the playbook, and the update is done. In a push-based system, you push the configuration updates to a central configuration management service, and the agents running on all machines periodically check for updates and run the changes.

Ansible supports a pull-based model with ansible-pull

Ansible obeys Alan Kay’s maxim: “Simple things should be simple; complex things should be possible.”

Ansible’s modules (that it ships with and community-contributed ones too) are declarative. They are also idempotent. If the deploy user doesn’t exist, Ansible will create it. If it does exist, Ansible won’t do anything.

The primary unit of reuse in the Ansible community is the module. Ansible playbooks aren’t really intended to be reused across different contexts. Roles are a way of collecting playbooks together, so they are more reusable.

**Using Vagrant to Set Up a Test Server**

If you prefer not to spend the money on a public cloud, I recommend you install Vagrant on your machine. Vagrant is an excellent open-source tool for managing vir‐ tual machines. You can use Vagrant to boot a Linux virtual machine inside your lap‐ top, and you can use that as a test server.

Vargant is responsible for building and maintaining portable virtual environments. It is like a Dockerfile for virtual machines.

Vagrant creates a vagrant user and a ssh key to enable the vagrant ssh command

$ ssh vagrant@127.0.0.1 -p 2222 -i /path/to/.vagrant/machines/default/virtualbox/private\_key

**Telling Ansible about the test server**

Ansible can manage only the servers it explicitly knows about. You provide Ansible with information about servers by specifying them in an inventory file. Create a file called hosts in the Playbookss directory. This file will serve as the inventory file.

Each server needs a name that Ansible will use to identify it. You can use the hostname of the server, or you can give it an alias and pass additional arguments to tell Ansible how to connect to it.

To tell Ansible about the Vagrant machine, we have to specify the user, ssh key etc

# Example 1-1. playbooks/hosts. We can call our vagrant machine testserver

testserver ansible\_host=127.0.0.1 ansible\_port=2222 \

ansible\_user=vagrant \

ansible\_private\_key\_file=.vagrant/machines/default/virtualbox/private\_key

We can also use the ansible.cfg file to avoid having to be so verbose in the inventory file. For an ec2 instance, it can be something like:

testserver ansible\_host=ec2-203-0-113-120.compute-1.amazonaws.com \

ansible\_user=ubuntu ansible\_private\_key\_file=/path/to/keyfile.pem

Example ansible.cfg file:

# Example 1-2. ansible.cfg

[defaults]

inventory = hosts

remote\_user = vagrant

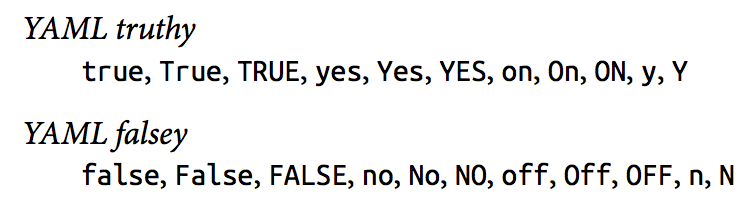
private\_key\_file = .vagrant/machines/default/virtualbox/private\_key

host\_key\_checking = False

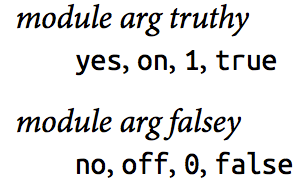
The command module is the default module. So all are valid:

1. $ ansible testserver -m command -a uptime
2. $ ansible testserver -a uptime,
3. $ ansible testserver -a “tail /var/log/dmesg”

The ansible syntax is YAML:

[](https://github.com/darshanime/notes/blob/master/assets/screenshot_2018-08-09_22-44-00.png)

Ansible also accepts more truthy/falsey arguments for modules:

[](https://github.com/darshanime/notes/blob/master/assets/screenshot_2018-08-09_22-44-27.png)

An Ansible convention is to keep files in a subdirectory named les, and Jinja2 templates in a subdirectory named templates.

We can create groups of hosts and mention them in our inventory file (aka the host’s file). Inventory files are in the .ini file format.

So, now our host’s file looks like so:

[webservers]

testserver ansible\_host=127.0.0.1 ansible\_port=2222

Execute the playbook by: $ ansible-playbook web-notls.yml If your playbook file is marked as executable and starts with a line that looks like this #!/usr/bin/env ansible-playbook (the shebang), then you can execute it by invoking it directly, like this: $ ./web-notls.yml

**YAML syntax**

**Start of File**

YAML files are supposed to start with three dashes to indicate the beginning of the document: —

**Comments**

Comments start with a number sign and apply to the end of the line, the same as in shell scripts, Python, and Ruby: # This is a YAML comment.

**Strings**

In general, YAML strings don’t have to be quoted, although you can quote them if you prefer. Even if there are spaces, you don’t need to quote them. For example, this is a string in YAML: this is a lovely sentence

Ansible will need you to quote strings if you use variable substitution, indicated by the use of {{ braces }}

**Lists**

YAML lists are like arrays in JSON and Ruby or lists in Python. Technically, these are called **sequences** in YAML, but I call them lists here to be consistent with the official Ansible documentation.

They are delimited with hyphens, like this:

* My Fair Lady
* Oklahoma
* The Pirates of Penzance

Note, no quoting is needed.

YAML also supports an inline format for lists, which looks like this: [My Fair Lady, Oklahoma, The Pirates of Penzance]

**Dictionary**

YAML dictionaries are like objects in JSON, dictionaries in Python, or hashes in Ruby. Technically, these are called mappings in YAML, but I call them dictionaries here to be consistent with the official Ansible documentation.

They look like this:

address: 742 Evergreen Terrace

city: Springfield

state: North Takoma

The JSON equivalent is shown here:

{

"address": "742 Evergreen Terrace", "city": "Springfield",

"state": "North Takoma"

}

YAML also supports an inline format for dictionaries, which looks like this: {address: 742 Evergreen Terrace, city: Springfield, state: North Takoma}

**Line folding**

You can do this with YAML by using line folding with the greater than (>) character. The YAML parser will replace line breaks with spaces.

For example:

address: >

Department of Computer Science,

A.V. Williams Building,

University of Maryland

city: College Park

state: Maryland

The JSON equivalent is as follows:

{

"address": "Department of Computer Science, A.V. Williams Building,\nUniversity of Maryland",

"city": "College Park",

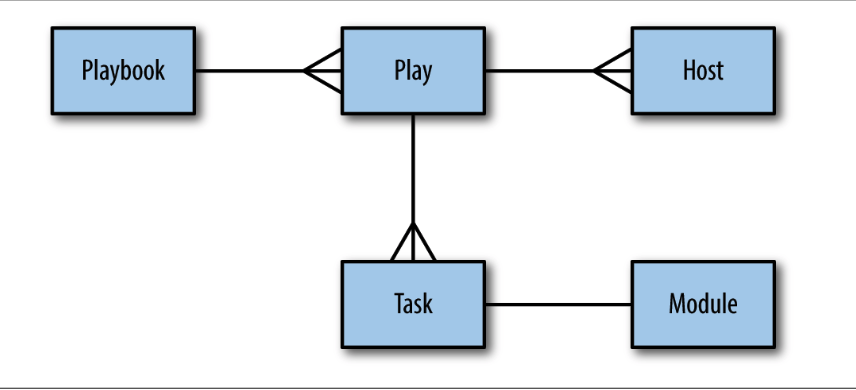
"state": "Maryland"

}

**A valid JSON file is also a valid YAML file. This is because YAML allows strings to be quoted, considers true and false to be valid Booleans, and has inline lists and dictionary syntaxes that are the same as JSON arrays and objects.**

If you think about it, a playbook is a list of dictionaries. Modules are scripts that come packaged with Ansible and perform some kind of action on a host. Ansible ships with the ansible-doc command-line tool, which shows documentation about modules. For example, to show the documentation for the service module, run this: $ ansible-doc service The modules that ship with Ansible all are written in Python, but modules can be written in any language. Recall from the first chapter that Ansible executes a task on a host by generating a custom script based on the module name and arguments, and then copies this script to the host and runs it.

**Ansible components**

[](https://github.com/darshanime/notes/blob/master/assets/screenshot_2018-08-09_23-07-43.png)

A playbook has a lof of Plays, which are just a series of tasks (using one module each) and running on a series of hosts.

You can define vars at a playbook level:

- name: Configure webserver with nginx and tls

hosts: webservers

become: True

vars:

key\_file: /etc/nginx/ssl/nginx.key

cert\_file: /etc/nginx/ssl/nginx.crt

conf\_file: /etc/nginx/sites-available/default

server\_name: localhost

tasks:

- name: Install nginx

apt: name=nginx update\_cache=yes cache\_valid\_time=3600

- name: create directories for ssl certificates

file: path=/etc/nginx/ssl state=directory

In our example, each value is a string (e.g., /etc/nginx/ssl/nginx.key), but any valid YAML can be used as the value of a variable. You can use lists and dictionaries in addition to strings and Booleans.

**Handlers**

They are just tasks which run only when they are notified by other tasks. Tasks notify only when they detect a state change caused by them. Handlers can be used if you want to for eg restart a service on a config change etc.

- name: copy TLS key

copy: src=files/nginx.key dest={{ key\_file }} owner=root mode=0600

notify: restart nginx

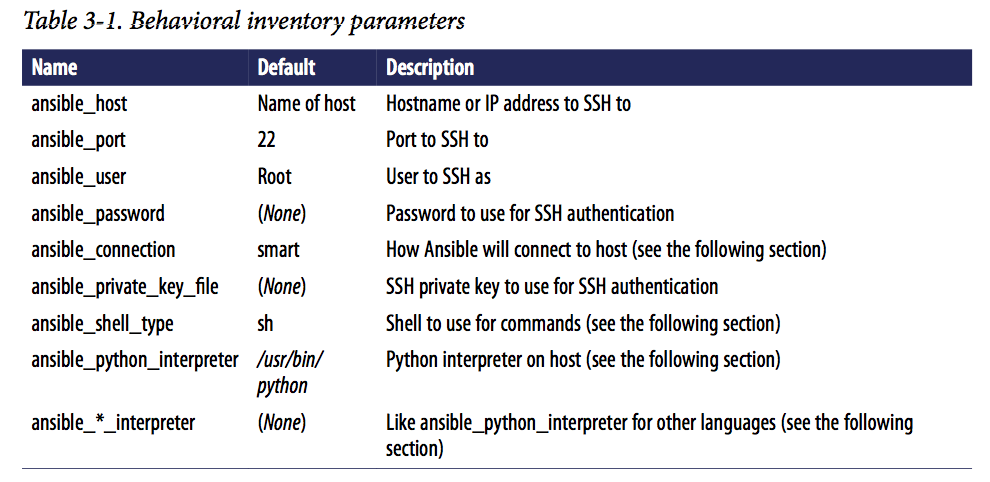
handlers:

- name: restart nginx

service: name=nginx state=restarted

**The Inventory File**

The default way to describe your hosts in Ansible is to list them in text files, called inventory files. Ansible has several parameters to control the behavioral inventory parameters.

[](https://github.com/darshanime/notes/blob/master/assets/screenshot_2018-08-10_22-48-58.png)

The ansible\_connection can support multiple transports to connect to the host. If the SSH client supports Control-Persist, Ansible will use the local SSH client. If the SSH client doesn’t support ControlPersist, the smart transport will fall back to using a Python-based SSH client library called Paramiko. ControlPersist, also known as SSH multi‐ plexing.

If the inventory file is marked executable, Ansible will assume it is a dynamic inven‐ tory script and will execute the file instead of reading it.

The Interface for a Dynamic Inventory Script An Ansible dynamic inventory script must support two command-line flags:

1. –host=<hostname> for showing host details
   * this is to show details of a particular host
2. –list for listing groups
   * this is to show listings of all the groups, and details about the individual hosts.

# assuming our inventory file is dynamic.py

$ ./dynamic.py --host=vagrant2

{ "ansible\_host": "127.0.0.1", "ansible\_port": 2200, "ansible\_user": "vagrant"}

$ ./dynamic.py --list

{"lb": ["delaware.example.com"],

"web": ["georgia.example.com", "newhampshire.example.com",

"newjersey.example.com", "ontario.example.com", "vagrant1"]}

Ansible ships with several dynamic inventory scripts that you can use. You can grab these by going to the Ansible GitHub repo and browsing the contrib/inventory directory. Many of these inventory scripts have an accompanying configuration file.

If you want to have both a regular inventory file and a dynamic inventory script, just put them all in the same directory and configure Ansible to use that directory as the inventory.

Ansible will let you add hosts and groups to the inventory during the execution of a playbook using add\_host.

Even if you’re using dynamic inventory scripts, the add\_host module is useful for scenarios where you start up new virtual machine instances and configure those instances in the same playbook. If a new host comes online while a playbook is executing, the dynamic inventory script will not pick up this new host.

**Variables**

Ansible has variables and a certain type of variable that Ansible calls a fact. The simplest way to define variables is to put a vars section in your playbook with the names and values of variables.

Ansible also allows you to put variables into one or more files, using a section called vars\_files for debugging, it’s often handy to be able to view the output of a variable. We saw how to use the debug module to print out an arbitrary message. We can also use it to output the value of the variable. It works like this: - debug: var=myvarname Often, you’ll find that you need to set the value of a variable based on the result(output) of a task. To do so, we create a registered variable using the register clause when invok‐ ing a module

The value of a variable set using the register clause is always a dictionary. Some of the keys always present are:

* changed -> indicates if state changed
* cmd -> invoked command as a list of strings
* stderr
* stdout

Ansible uses Jinja2 to implement variable dereferencing on the dicts.

When Ansible runs a playbook, before the first task runs, this happens:

GATHERING FACTS \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [servername]

When Ansible gathers facts, it connects to the host and queries it for all kinds of details about the host: CPU architecture, operating system, IP addresses, memory info, disk info, and more. This information is stored in variables that are called facts, and they behave just like any other variable.

Here’s a simple playbook that prints out the operating system of each server:

- name: print out operating system

hosts: all

gather\_facts: True

tasks:

- debug: var=ansible\_distribution

Ansible implements fact collecting through the use of a special module called the setup module.

Any Module Can Return Facts. The use of ansible\_facts in the return value is an Ansible idiom. If a module returns a dictionary that contains ansible\_facts as a key, Ansible will create variable names in the environment with those values and associate them with the active host.

For eg:

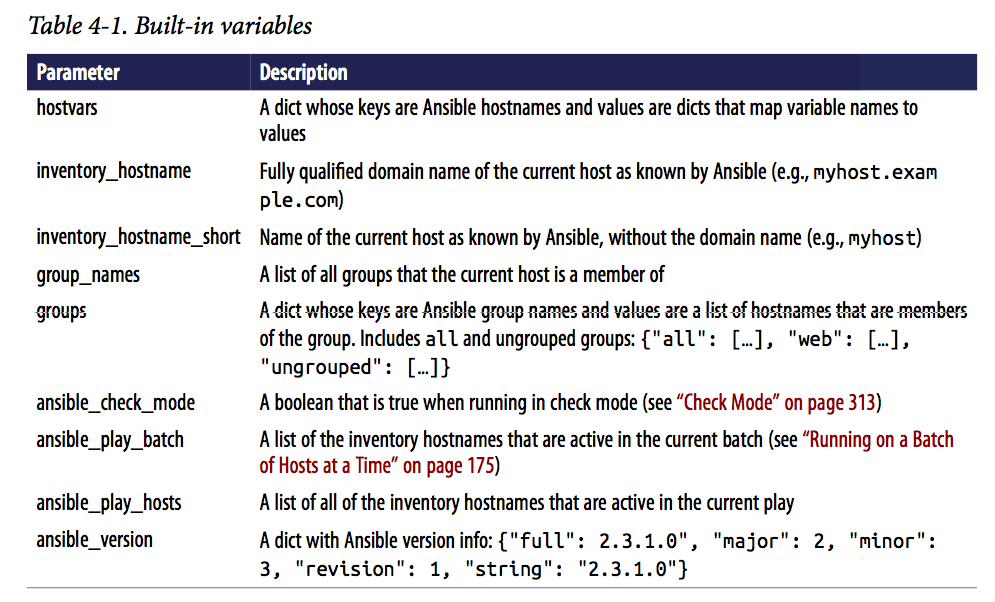
- name: get ec2 facts

ec2\_facts:

- debug: var=ansible\_ec2\_instance\_id

Here, 🔝, the variable ansible\_ec2\_instance\_id was returned by ec2\_facts module.

Some of Ansible’s variables that are always available:

[](https://github.com/darshanime/notes/blob/master/assets/screenshot_2018-08-10_23-40-38.png)

In Ansible, variables are scoped by host. It only makes sense to talk about the value of a variable relative to a given host.

Eg: {{ hostvars['db.example.com'].ansible\_eth1.ipv4.address }}

This evaluates to the ansible\_eth1.ipv4.address fact associated with the host named db.example.com.

Here, we did not use hostvars.db.example.com since the string “db.example.com” has periods.

The groups var can be useful to access variables for a group of hosts.

# sample configuration file

backend web-backend

{% for host in groups.web %}

server {{ hostvars[host].inventory\_hostname }} \

{{ hostvars[host].ansible\_default\_ipv4.address }}:80

{% endfor %}

# generated file

backend web-backend

server georgia.example.com 203.0.113.15:80

server newhampshire.example.com 203.0.113.25:80

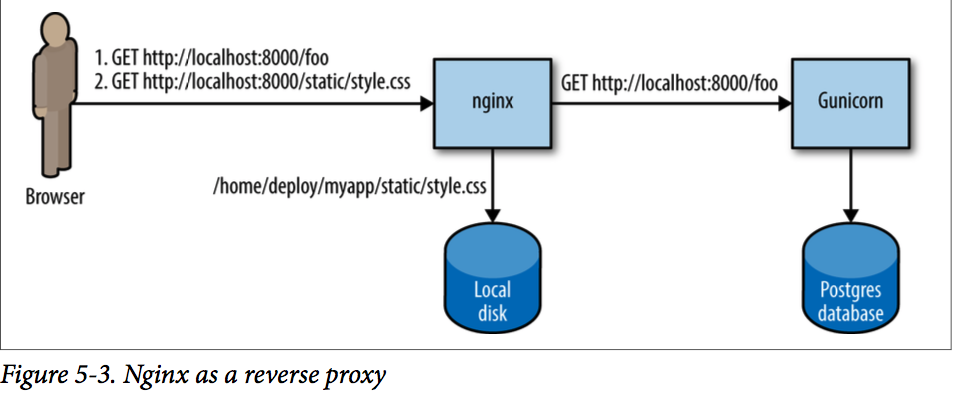
server newjersey.example.com 203.0.113.38:80

Variables set by passing -e var=value to ansible-playbook have the highest precedence. It is --extra-vars for ansible-playbook.

Django implements the standard Web Server Gateway Interface (WSGI),2 so any Python HTTP server that supports WSGI is suitable for running a Django application such as Mezzanine. We’ll use Gunicorn, one of the most popular HTTP WSGI servers.

Gunicorn will execute our Django application, just like the development server does. However, Gunicorn won’t serve any of the static assets associated with the application.

Although Gunicorn can handle TLS encryption, it’s common to configure Nginx to handle the encryption.

[](https://github.com/darshanime/notes/blob/master/assets/screenshot_2018-08-10_23-52-45.png)

We need to run Guni‐ corn as a daemon, and we’d like to be able to easily stop it and restart it. Numerous service managers can do this job. We’re going to use Supervisor because that’s what the Mezzanine deployment scripts use.

**Listing tasks in a Playbook**

Useful for getting the list of tasks that will be run: $ ansible-playbook --list-tasks mezzanine.yml

Ansible ships with a django\_manage module that invokes manage.py commands. We could invoke it like this:

* name: initialize the database django\_manage: command: createdb –noinput –nodata app\_path: “{{ proj\_path }}” virtualenv: “{{ venv\_path }}”

script module instead. This will copy over a custom script and execute it.

In order to run these scripts in the context of the virtualenv, I also needed to set the path variable so that the first Python executable in the path would be the one inside the virtualenv.

- name: set the site id

script: scripts/setsite.py

environment:

PATH: "{{ venv\_path }}/bin"

PROJECT\_DIR: "{{ proj\_path }}"

PROJECT\_APP: "{{ proj\_app }}"

We have the cron module as well:

- name: install poll twitter cron job

cron: name="poll twitter" minute="\*/5" user={{ user }} job="{{ manage }} \

poll\_twitter"

**Roles - Scaling up your playbooks**

One of the things I like about Ansible is how it scales both up and down. I’m not referring to the number of hosts you’re managing, but rather the complexity of the jobs you’re trying to automate.

Ansible scales down well because simple tasks are easy to implement. It scales up well because it provides mechanisms for decomposing complex jobs into smaller pieces.

In Ansible, the role is the primary mechanism for breaking a playbook into multiple files. This simplifies writing complex playbooks, and it makes them easier to reuse. Think of a role as something you assign to one or more hosts. For example, you’d assign a database role to the hosts that will act as database servers.

**Basic Structure of a Role**

Say, we have a role called database. It lives in the roles/database directory.

* roles/database/tasks/main.yml
  + Tasks
* roles/database/ les/
  + Holds files to be uploaded to hosts
* roles/database/templates/
  + Holds Jinja2 template files
* roles/database/handlers/main.yml
  + Handlers
* roles/database/vars/main.yml
  + Variables that shouldn’t be overridden
* roles/database/defaults/main.yml
  + Default variables that can be overridden
* roles/database/meta/main.yml
  + Dependency information about a role

Ansible looks for roles in the role’s directory alongside your playbooks. It also looks for systemwide roles in /etc/ansible/roles. You can customize the systemwide location of roles by setting the roles\_path setting in the defaults section of your ansible.cfg or setting the ANSIBLE\_ROLES\_PATH env var.

When we are done writing roles, we can assign them to our hosts like so:

- name: deploy mezzanine on vagrant

hosts: web

vars\_files:

- secrets.yml

roles:

- role: database

database\_name: "{{ mezzanine\_proj\_name }}" # these vars can be defined in vars/main.yml, or defaults/main.yml

database\_user: "{{ mezzanine\_proj\_name }}"

- role: mezzanine

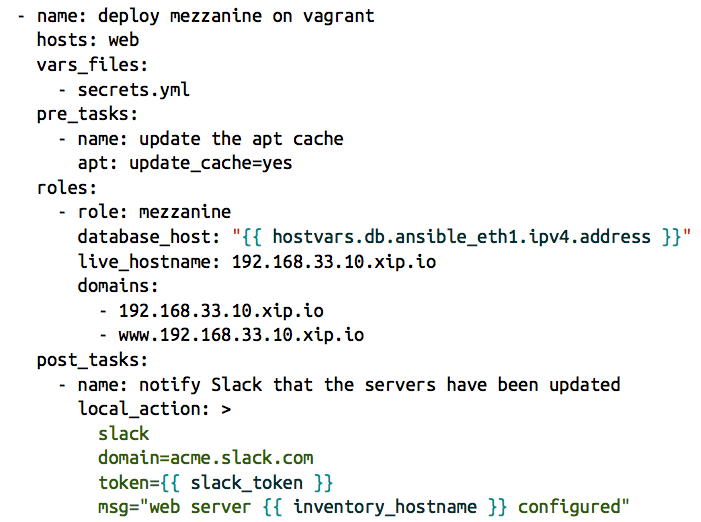
live\_hostname: 192.168.33.10.xip.io

domains:

- 192.168.33.10.xip.io

- www.192.168.33.10.xip.io

Ansible allows you to define a list of tasks that execute before the roles with a pre\_tasks section, and a list of tasks that execute after the roles with a post\_tasks section

[](https://github.com/darshanime/notes/blob/master/assets/screenshot_2018-08-11_09-03-52.png)

**Writing the database role**

**roles/database/tasks/main.yml**

**has all the tasks like in a regular playbook**

**roles/database/defaults/main.yml**

**here, we can give the default value of the variables that we use in our tasks/main.yml playbook**

**roles/database/handlers/main.yml**

**defines a handler (like say, restart postgres). Any task can use notify to call this handler based on it’s result - execute if state changed**

**roles/database/files/pg\_hba.conf**

**roles/database/files/postgresql.conf**

So, Ansible roles are just long playbooks that have been broken down into organized smaller files.

Ansible doesn’t have any notion of namespace across roles. This means that variables that are defined in other roles, or elsewhere in a playbook, will be accessible everywhere. So, it’s a good practice to prefix variables in the role with the name of the role.

Ansible ships with another command-line tool, ansible- galaxy. Its primary purpose is to download roles that have been shared by the Ansible community. It can also be used to generate scaffolding, an initial set of files and directories involved in a role: $ ansible-galaxy init -p playbooks/roles web

The -p flag tells ansible-galaxy where your roles directory is. If not specified, the role files will be created in your current directory.

When you have a role that is dependent on another role already having been executed, you can leverage ansible’s support for dependent roles. Eg, for django role, you could have mentioned:

dependencies:

- { role: web }

- { role: memcached }

**Introduction to Ansible**

“Ansible” is a fictional machine capable of superluminal communication (faster than light communication)

Use cases:

* provisioning
* configuration management
* application deployments
* rolling upgrades - CD
* security and compliance
* orchestration

Ansible has a powerful and simple declarative language. (You just specify what you want, not it should be done)

Key components:

* Modules (Tools)
  + bits of code copied to the target system
  + executed to satisfy the task declaration
  + customizable
  + examples:
    - cloud modules, database modules, files, monitoring, network, notification modules
  + commonly used modules:
    - apt/yum, copy, file, git etc
* Tasks
* Inventory
  + contains the information about hosts in ini format
* Plays
* Playbook

**Ansible Commands**

We can run commands using one of the several modules and giving it the required arguments and specifying the hosts file

* each command needs to have an inventory specified with -i <hosts file>
* ansible all -i ./hosts -m command -a “uptime”
* this 🔝 uses the command module, gives it argument “uptime” and runs it in all hosts mentioned in the hosts file
* the hosts file has:

[lh] localhost ansible\_connection=local

We can install HTTPD package: ansible all -i ./hosts -m apt -a “name=httpd state=present”

We can start/stop HTTPD service: ansible all -i ./hosts -m service -a “name=httpd enabled=yes start=started”

We can test ansible connections to all the hosts using ping ansible all -i ./hosts -m command -a “ping”

Or 🔝 we can use the ping module! ansible all -i ./hosts -m ping

**Ansible playbooks**

- name: This is a play # this is the name of the play

hosts: web-servers # we select hosts here

remote\_user: ec2-user # the arguments for the playbook

become: yes # the arguments for the playbook, do you want to be superuser?

gather\_facts: no # the arguments for the playbook

vars: # here we define the variables to be used in the tasks later

state: present

tasks: # here we define the tasks using modules, giving them args (possibly from the vars)

- name: Install Apache

yum: name=httpd state={{ state }}

Here, in the tasks, we used the yum module and passed it args like we did in the commands 🔝

We can run the playbook like so:

ansible-playbook play.yml -i hosts

Perform a “dry run” ansible-playbook play.yml -i hosts –check

**Loops**

Loops are possible in the playbooks - the playbooks are a DSL!

tasks:

- name: Install Apache and PHP

yum: name={{item}} state={{state}}

with\_items:

- httpd

- php

Many types of loops:

* with\_nested
* with\_dict
* with\_fileglob
* with\_together
* with\_sequence
* until
* with\_random\_choice
* with\_first\_found
* with\_indexed\_items
* with\_lines

**Handlers**

We have handlers that run a task if it has “changed” status

tasks:

- yum: name={{item}} state=installed

with\_items:

- httpd

- memcached

notify: Restart Apache

handlers:

- name: Restart Apache

service: name=httpd state=restarted

**Tags**

tasks:

- name: Install Apache and PHP

yum: name={{item}} state={{state}}

with\_items:

- httpd

- php

tags:

- configuration

Tags are used to specify where to run the playbook

ansible-playbook example.yml –tags “frontend-prod” ansible-playbook example.yml –skip-tags “frontend-prod”

We have special tags like “tagged”, “untagged”, “all”

**Results**

We can register task outputs as well (for debugging etc)

- shell: httpd -v | grep version | awk '{print $3}' | cut -f2 -d'/'

register: result

- debug: var=result

**Conditional tasks**

Run these when some condition is satisfied

tasks:

- name: Install Apache and PHP

yum: name={{item}} state={{state}}

with\_items:

- httpd

- php

tags:

- configuration

when: ansible\_os\_family == "RedHat"

**Errors**

By default, ansible stops on errors We can add ignore\_error parameter to skip potential errors

tasks:

- name: Install Apache and PHP

yum: name={{item}} state={{state}}

with\_items:

- httpd

- php

ignore\_errors: yes

# we can define the condition on which to declare the failure

- name: this command prints FAILED when it fails

command: /usr/bin/example-command -x -y -z

resiter: command\_result

faield\_when: "'FAILED' in command\_result.stderr"

# managing errors using blocks

tasks:

- block:

- debug: msg='i execute normally'

- command: /bin/false

- debug: msg='i never execute, cause ERROR!'

rescue:

- debug: msg='I caught an error'

- command: /bin/false

- debug: msg='I also never execute :-('

always:

- debug: msg="this always executes"

**Example playbook**

- name: All server setup

hosts: all

become: yes # we'll need to be root

vars:

selinux: permissive

tasks:

- name: Change SELinux to permissive mode

selinux:

policy: targeted

state: "{{ selinux }}"

- name: Copy motd file

copy:

content: "Welcome to my server!" dest=/etc/motd

- name: Web server setup

hosts: web-server

become: yes # we'll need to be root

tasks:

- name: Install HTTPD

yum: name=httpd start=present

notify: Restart Apache

- name: Start and enable httpd

service: name=httpd restarted=restarted

when: just\_installed\_httpd

- name: Copy hello world

copy:

content: "Hello World!"

dest: /var/www/html

- name: Set sshd.conf to not allow root login

lineinfile:

path: /etc/ssh/sshd\_config

regexp: "^PermitRootLogin "

insertafter: "^PermitRootLogin" line="no"

notify: RestartSSH

handlers:

- name: Restart Apache

service: name=httpd state=restarted enabled=yes

- name: RestartSSH

service: name=sshd state=restarted enabled=yes

Now, we can run this and pass the vars: ansible-playbook -i ../hosts lab2.yml -e “selinux=permissive”

**Ansible variables**

The precedence of variables:

1. Extra vars
2. Task vars (only for the task)
3. Block vars
4. Role and include vars
5. Play vars\_files
6. Play vars\_prompt
7. Play vars
8. Set\_facts

etc

**Special variables**

Ansible has some special variables as well:

* hostvars
* group\_names
  + is a list (array) of all the groups the current host is in
* groups
  + is a list of all the groups and hosts in the inventory

We use **debug** to view the content

- name: debug

hosts: all

tasks:

- name: Show hostvars[inventory\_hostname]

debug: var=hostvars[inventory\_hostname]

**Templates**

Templates allow us to create dynamic configuration files using variables

* template: src=/mytemplates/foo.j2 dest=/etc/file.conf owner=bin group=wheel mode=0644

Jinja2 is just like Django templating system

{{ variable }}

{% for server in groups.webservers %}

{{ server }}

{% endfor %}

We have variables

{% set my\_var='this-is-a-test' %}

{{ my\_var | replace('-', '\_') }}

In YAML, template variable must be quoted

vars:

var1: {{ foo }} <<< ERROR!

var2: “{{ bar }}”

var3: Echoing {{ foo }} here is fine

**Ansible roles**

Roles are a redistributable and reusable collection of:

* tasks
* files
* scripts
* templates
* variables

Roles are used to set up and configure services.

* install packages.
* copying files
* starting daemons

Example: Apache, MySQL, Nagios etc

Directory structure: roles

* myapp
  + defaults
  + files
  + handlers
  + meta
  + tasks
  + templates
  + vars