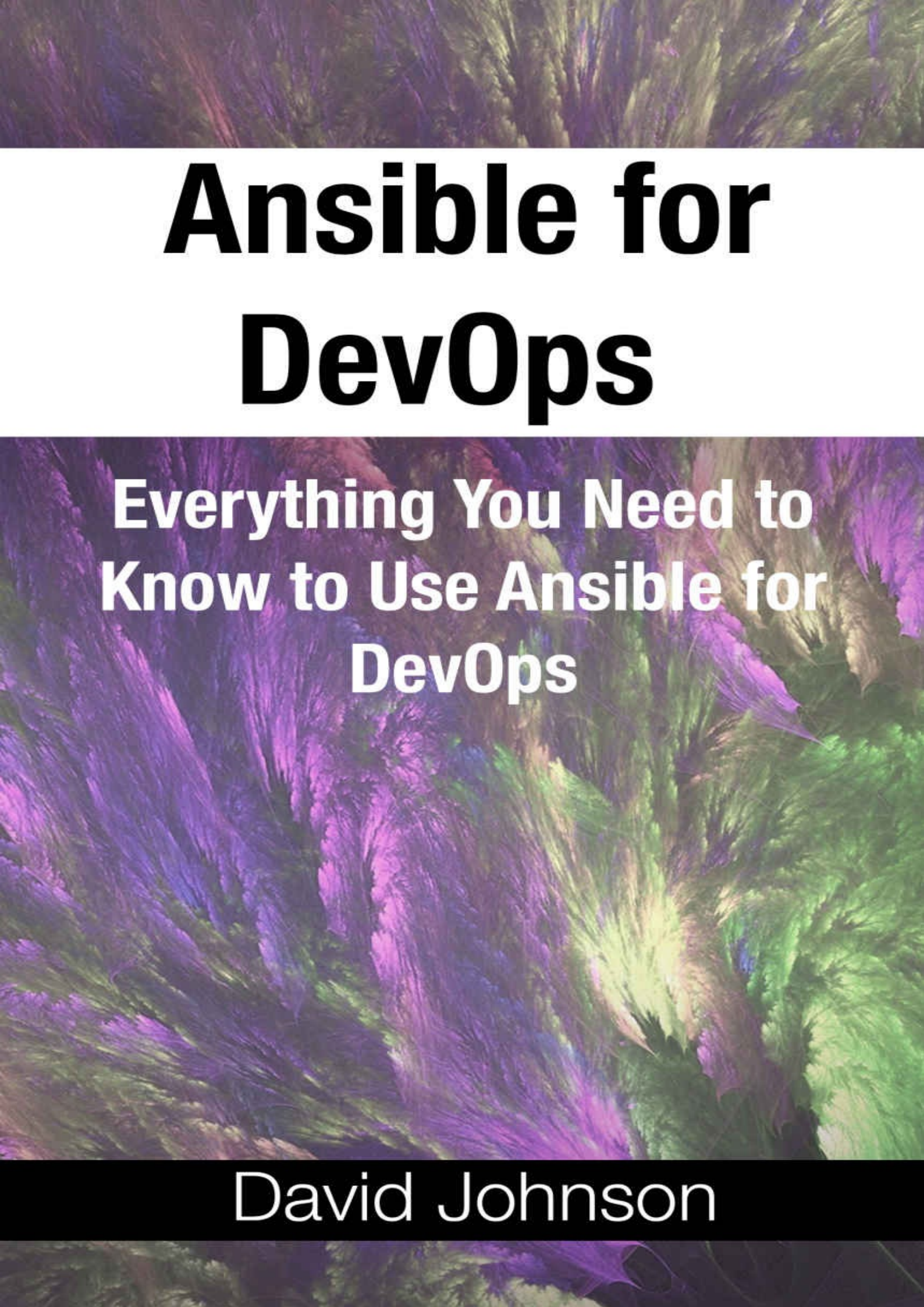
Ansible for DevOps
Everything You Need to Know to Use Ansible for DevOps
David Johnson

Ansible for DevOps:

Everything You Need to Know to Use Ansible for DevOps

By David Johnson

Table of Contents

Disclaimer

While all attempts have been made to verify the information provided in this book, the author does assume any responsibility for errors, omissions, or contrary interpretations of the subject matter contained within. The information provided in this book is for educational and entertainment purposes only. The reader is responsible for his or her own actions and the author does not accept any responsibilities for any liabilities or damages, real or perceived, resulting from the use of this information.

The trademarks that are used are without any consent, and the publication of the trademark is without permission or backing by the trademark owner. All trademarks and brands within this book are for clarifying purposes only and are the owned by the owners themselves, not affiliated with this document.

Introduction

Ansible is an Important IT tool for automation of most tasks. Due to its ease of use and how important the tool is, its popularity has increased. As an IT professional, it is good for you to know how to operate this tool. Ansible can be used for DevOps, a mechanism that is good for organization of tasks and development teams in Ansible. This book provides guidance on the important aspects of using Ansible for DevOps. Enjoy reading!

Chapter 1- Deployments with Ansible

Most teams find it is hard to deploy applications to the server. With the use of the right resources and modern deployment processes, the deployment process becomes very easy and efficient.

Strategies for Deployment

There are multiple ways we can deploy code to our servers. The following are the deployment strategies that can be used:

1. Single-server deployments.
2. Zero-downtime multi-server deployments.
3. Capistrano-style and blue-green deployments.

The above deployment strategies cover most of the deployment cases.

Simple single-server deployments

Most websites and small applications are usually run on a dedicated server or on a single virtual machine. It is easy for us to use Ansible for the purpose of managing the configuration on the server. Since you are managing only a single server, it will be good for you to encapsulate the setup so that you can avoid having a snowflake server.

Suppose you have a Ruby on Rails website in which users are allowed to perform some CRUD operations on some posts. The posts have been kept in a database, and each has a title and a body.

For this to be done, we should begin by creating a vagrant VM using the vagrantfile shown below:

```
# -*- mode: ruby -*-
# vi: set ft=ruby :
Vagrant.configure(2) do |config|
config.vm.box = “computerguy/ubuntu1404”
config.vm.provider “virtualbox” do |v|
v.name = “rails-example”
v.memory = 1024
v.cpus = 2
end
```

config.vm.hostname = “rails-example”

config.vm.network :private_n

etwork, ip: “192.168.160.7”

config.vm.provision “ansible” do |ansible|

ansible.playbook = “playbooks/main.yml”

ansible.sudo = true

end

end

Provisioning a Ruby on Rails Server

To be prepared for installation of our app, you should do the following:

1. Install Git (if your application version is controlled in the Git repository).
2. Install Node.js
3. Install Ruby, the latest version.
4. Install Passenger and Nginx.
5. Install the other dependencies.

We can now create a playbook for our provisioning tasks in a new file in “playbooks/provision.yml”.

–

- hosts: all

sudo: yes

vars_files:

- vars.yml

roles:

- computerguy.git

- computerguy.nodejs

- computerguy.ruby

- computerguy.passenger

tasks:

- name: Install app dependencies.

apt: “name={{ item }} state=present”

with_items:

- libsqlite3-dev

- libreadline-dev

- name: Ensure the app directory is available and is writeable.

file:

path: “{{ app_directory }}”

state: directory

owner: “{{ app_user }}”

group: “{{ app_user }}”

mode: 0755

Above is a very simple playbook. There are some variables that should be defined so as to ensure that the role installs the correct version of Ruby. The role needs to be configured correctly so that our app can be served correctly.

There are some other variables that should be defined in the file “playbooks/vars.yml”, and this can be done as shown below:

 # Variables for the app.

app_directory: /opt/sample-rails-app

app_user: www-data

 # Variables for the Passenger and Nginx.

passenger_server_name: 0.0.0.0

passenger_app_root: /opt/example-rails-app/public

passenger_app_env: production

passenger_ruby: /usr/local/bin/ruby

 # Variables for the Ruby installation.

ruby_install_from_source: true

ruby_download_url: http://cache.ruby-lang.org/pub/ruby/2.2/ruby-2.2.0.tar.gz

ruby_version: 2.2.0

The passenger variables will tell the Passenger to run the server that is available on every network interface and then launch the app with "*production*" settings by use of the ruby binary, which has been installed in the directory "/usr/local/bin/ruby". The ruby variables will tell the Ruby to install Ruby 2.2.0 from the source since the available packages through the standard apt repositories of Ubuntu are in older versions.

The playbook, which has been specified in the Vagrantfile "playbooks/main.ym", is not available. This is what we now need to create and then include in the playbook provioning.yml in the server. The deployment steps will be separated into another playbook and that will be included separately. The following should be added into the file "playbooks/main.yml":

–

- include: provision.yml

Deployment of a Rails app to Server

The deploy.yml file will have a number of tasks and it can now be created. It will be responsible for using the Git to check the latest release of the Rails app. It will also copy over the template “secrets.yml”, which is hold secure app data and is needed for running of the app. It will also ensure that all the gems that the app requires have been installed. Here is the code for the file:

–

- hosts: all

sudo: yes

vars_files:

- vars.yml

roles:

- computerguy.passenger

tasks:

- name: Ensure the sample app is at correct release.

git:

repo: https://github.com/computerguy/sample-rails-app.git

version: “{{ app_version }}”

dest: “{{ app_directory }}”

accept_hostkey: true

register: app_updated

notify: restart nginx

- name: Ensure the secrets file is available.

template:

src: templates/secrets.yml.j2

dest: “{{ app_directory }}/config/secrets.yml”

owner: “{{ app_user }}”

group: “{{ app_user }}”

mode: 0664

notify: restart nginx

- name: Install the required dependencies and bundler.

shell: “bundle install —path vendor/bundle chdir={{ app_directory }}”

when: app_updated.changed == true

notify: restart nginx

- name: Check if the database exists.

stat: “path={{ app_directory }}/db/{{ app_environment.RAILS_ENV }}.sqlite3”

register: app_db_exists

- name: Create database.

shell: “bundle exec rake db:create chdir={{ app_directory }}”

when: app_db_exists.stat.exists == false

notify: restart nginx

- name: Perform some deployment-related rake tasks.

shell: “{{ item }} chdir={{ app_directory }}”

with_items:

- bundle exec rake db:migrate

- bundle exec rake assets:precompile

environment: app_environment

when: app_updated.changed == true

notify: restart nginx

- name: Ensure sample app has the correct user for the files.

file:

path: “{{ app_directory }}”

state: directory

owner: “{{ app_user }}”

group: “{{ app_user }}”

recurse: yes

notify: restart nginx

In the above playbook, a new role has been added. Many tasks usually lead to changes in the filesystem and this can change the behavior of the app and the “restart nginx” handler will be notified. The Passenger will reload the configuration and then restart the app.

We now need to add some variables to the file “vars.yml” and a new template. At this point, we can create the secrets file in the “playbooks/templates/secrets.yml.j2”. This is shown below:

secret_key_base: {{ app_secrets.dev }}

test:

secret_key_base: {{ app_secrets.test }}

production:

secret_key_base: {{ app_secrets.prod }}

A dictionary variable will be used for the app_secrets, so let’s add it together with the other new variables to the file “playbooks/vars.yml”:

–

 # Variables for the app.

app_version: 1.2.2

app_directory: /opt/sample-rails-app

app_user: www-data

app_secrets:

dev: fe452ec1e41eedc5af7d83f5a123a6

test: 4408f36dd290766d2f368fdfcedf4d

prod: 7bf801da2a24c9a673ea86a1328caa

app_environment:

RAILS_ENV: production

 # Variables for the Passenger and the Nginx.

passenger_server_name: 0.0.0.0

passenger_app_root: /opt/sample-rails-app/public

passenger_app_env: production

passenger_ruby: /usr/local/bin/ruby

 # Variables for the Ruby installation.

ruby_install_from_source: true

ruby_download_url: http://cache.ruby-lang.org/pub/ruby/2.2/ruby-2.2.0.tar.gz

ruby_version: 2.2.0

Provision and Deploy the Rails App

Our project folder should now be as follows:

deployments/

playbooks/

templates/

secrets.yml.j2

deploy.yml

main.yml

provision.yml

vars.yml

Vagrantfile

Before running the playbook, it is a good idea for us to ensure that we have all the dependencies that are needed. In cases where we had to build everything from scratch, we would have to create a directory for roles, but since in our case we are using these from the Ansible Galaxy, we can include them directly.

The following should be added to the file “requirements.txt”:

computerguy.git

computerguy.ruby

computerguy.nodejs

computerguy.passenger

You can then open the command line and then execute the command *"$ ansible-galaxy install –r requirements.txt*" in the same directory. With that command, all the roles will be downloaded and then installed in the default directory for roles if they do not exist.

You can then navigate back to the main directory with the vagrantfile and then execute the command *"vagrant up"*. If everything runs correctly, the playbook should be seen completing execution successfully after some minutes as shown below:

TASK: [Ensure demo application has correct user for files.] *************

changed: [default]

NOTIFIED: [computerguy.passenger | restart nginx] ***********************

changed: [default]

PLAY RECAP

****

default : ok=46 changed=28 unreachable=0 failed=0

You can then open the browser and type the url http://192.168.160.7/. You will see the app displayed on the browser. The app will be working correctly, but some improvements can be made to it.

Deploying application updates

We need to test whether we can deploy without having to provision. We have to create an inventory file to tell the Ansible how to connect directly to the VM that is managed by Vagrant.

Begin by creating the file “playbooks/inventory-ansible” with the contents given below:

[rails]

192.168.160.7

[rails:vars]

ansible_ssh_user=vagrant

ansible_ssh_private_key_file=~/.vagrant.d/insecure_private_key

You can then run the command given below in the directory for playbooks in the directory for the playbooks:

$ ansible-playbook deploy.yml -i inventory-ansible

The playbook should run correctly. You can then update the version “app_version to 1.3.0” and then execute the command given below:

$ ansible-playbook deploy.yml -i inventory-ansible

The deployment should then take a number of minutes before completing and you will see an app that is an improvement on what you had previously.

Zero-downtime multi-server deployments

For those who need to deploy applications on multiple servers for horizontal redundancy and scalability, the deployment process will be somewhat tough, resulting in a complicated and downtime process.

You should begin by creating lightweight vagrant VMs by use of the vagrantfile given below:

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

Vagrant.configure("2") do |config|
# The Base VM OS configuration.
config.vm.box = "computerguy/ubuntu1404"
config.vm.synced_folder '.', '/vagrant', disabled: true
config.ssh.insert_key = false

config.vm.provider :virtualbox do |v|
v.memory = 256
v.cpus = 1
end

# Define the four VMs with a static private IP addresses.
boxes = [
```

```
{ :name => “nodejs1”, :ip => “192.168.160.2” },
{ :name => “nodejs2”, :ip => “192.168.160.3” },
{ :name => “nodejs3”, :ip => “192.168.160.4” },
{ :name => “nodejs4”, :ip => “192.168.160.5” }
]

# Provision each of our VMs.
boxes.each do |opts|
config.vm.define opts[:name] do |config|
config.vm.hostname = opts[:name]
config.vm.network :private_network, ip: opts[:ip]

# Provision all VMs using the Ansible after last VM has been booted.
if opts[:name] == “nodejs4”
config.vm.provision “ansible” do |ansible|
ansible.playbook = “playbooks/main.yml”
ansible.inventory_path = “inventory”
ansible.limit = “all”
end
end
end
end

end
```

In the above file, we have defined four VMs and each of these will use 256MB of RAM, a

unique IP address and hostname. The Node.js app will not need to use much power and memory. The following two files should be created in the folder with the vagrantfile:

deployments-rolling/

playbooks/

main.yml

inventory

Vagrantfile

We can now define the list of the Node.js API app VMs by the IP addresses in the inventory file as shown below:

[nodejs-api]

192.168.160.2

192.168.160.3

192.168.160.4

192.168.160.5

[nodejs-api:vars]

ansible_ssh_user=vagrant

ansible_ssh_private_key_file=~/.vagrant.d/insecure_private_key

We can then name two separate playbooks in the playbook “main.yml”; one will be for initial provisioning while the other one will be for deployment. This is shown below:

–

- include: provision.yml

- include: deploy.yml

= **–**

- hosts: nodejs-api

sudo: yes

vars:

nodejs_forever: true

firewall_allowed_tcp_ports:

- “22”

- “8080”

roles:

- computerguy.firewall

- computerguy.nodejs

The above playbook will run on all of the servers that have been defined in our inventory file, and two roles will be run on the servers. Since the two roles are being used from the Ansible Galaxy, it is good for us to include them in a requirements file so that any CI tools and others that make use of this file can install the necessary roles easily.

In your root folder, create a "*requirements.txt*" file and then add the code given below to it:

computerguy.firewall

computerguy.nodejs

With that, anyone who is new and needs to run the playbook will only need to execute the command "ansible-galaxy install -r requirements.txt" so as to install any roles that are expected. The following should then be the structure of the project directory:

deployments-rolling/

playbooks/

deploy.yml

main.yml

provision.yml

inventory

requirements.txt

Vagrantfile

Before running the vagrant up so as to observe our infrastructure in action, we need to build the playbook "deploy.yml", as this will ensure that our app is available and is running correctly on the servers. The following code should be added inside the file "deploy.yml":

–

- hosts: nodejs-api

gather_facts: no

sudo: yes

vars_files:

- vars.yml

The “sudo” command should be used for the purpose of making things simple. Note that there are a few variables that we have to define, and we need to track them separately for an easier revision history of the file. We will have to define the files in the file “vars.yml” and in the same directory as the playbook “*deploy.yml”*. This is shown below:

–

app_repository: https://github.com/computerguy/sample-nodejs-api.git

app_version: “1.0.0”

app_directory: /opt/sample-nodejs-api

Now that we have saved the file “*vars.yml”* we can continue building the “*deploy.yml”* file, beginning with the task so as to clone the repository for the app:

- name: Ensure that the Node.js API app is available.

git:

repo: “{{ app_repository }}”

version: “{{ app_version }}”

dest: “{{ app_directory }}”

accept_hostkey: true

register: app_updated

notify: restart forever apps

By using that variables for the repo and version for Git module, we ensure that there is flexibility, since changes to the app version can occur frequently and management of this in a separate “*vars.yml”* file is easy.

The handler “restart forever apps” should be notified once codebase has changed. The handler “restart forever apps handler” will be defined later in our playbook:

- **name: Stop all the running instances of our app.**

command: “forever stopall”

when: app_updated.changed

- name: Ensure that the Node.js API app dependencies are available.

npm: “path={{ app_directory }}”

when: app_updated.changed

- name: Run the Node.js API app tests.

command: “npm test chdir={{ app_directory }}”

when: app_updated.changed

Running the tests for the app each time we deploy is a way to make sure that the app is available and is in a functioning state. It is good for us to run the tests during the deployment time to ensure that there is zero down-time deployment. This is shown below:

```
- name: Get the list of all the running Node.js apps.
command: forever list
register: forever_list
changed_when: false

- name: Ensure that the Node.js API app has been started.
command: "forever start {{ app_directory }}/app.js"
when: "forever_list.stdout.find('app.js') == -1"
```

If the app is available and is running as expected, it is a good idea for us to make sure that it has been started. A command exists that can be used for checking the list of all running apps, and another one for starting the app if it has not been started. This is shown below:

```
- name: Add the cron entry to start the Node.js API app after reboot.
cron:
name: "Start Node.js API app"
special_time: reboot
job: "forever start {{ app_directory }}/app.js"
```

The final task is adding the cron job. This will ensure that the app is started once the

server has been rebooted. It is now time for us to define the handler that will execute the command “forever restartall”:

handlers:

- name: restart forever apps

command: “forever restartall”

At this point, we should have a complete configuration of Vagrant and Ansible playbook. The playbook will clone the project, run the tests to ensure it is okay, and then use forever to start the app and then ensure that it is started whenever the server has rebooted.

The command given below can be executed to test our new servers and ensure that the app is running as expected:

**$ for j in {2..5}; **

**do curl -w “\n” “http://192.168.160.$j:8080/hello/nicohsam”; **

done

If your server is online, then the text “hello nicohsam” should be printed on the screen 4 times.

Deployment to App Servers Behind Load Balancer

We have two separate API layers that are tasked with ensuring that the checking of the server is done despite being so complex, and this will not be determined whether some servers are on or off.

Create a new folder for a project named “deployments-balancer” and then add the vagrantfile given below:

 # -*- mode: ruby -*-

 # vi: set ft=ruby :

Vagrant.configure(“2”) do |config|

 # Base VM OS configuration.

config.vm.box = “computerguy/ubuntu1404”

config.vm.synced_folder ‘.’, ‘/vagrant’, disabled: true

config.ssh.insert_key = false

config.vm.provider :virtualbox do |v|

v.memory = 256

v.cpus = 1

end

 # Define the four VMs with a static private IP addresses.

boxes = [

```
{ :name => “bal1”, :ip => “192.168.160.2” },
{ :name => “app1”, :ip => “192.168.160.3” },
{ :name => “app2”, :ip => “192.168.160.4” }
]

# Provision each of our VMs.
boxes.each do |opts|
config.vm.define opts[:name] do |config|
config.vm.hostname = opts[:name]
config.vm.network :private_network, ip: opts[:ip]

# Provision the VMs by use of the Ansible after the last VM has been booted.
if opts[:name] == “app2”
config.vm.provision “ansible” do |ansible|
ansible.playbook = “playbooks/provision.yml”
ansible.inventory_path = “inventory”
ansible.limit = “all”
end
end
end
end

end
```

With the vagrantfile, three servers will be created and each will be running Ubuntu. Let us now create the inventory file having the connection variables and groupings. This is

shown in the code given below:

```
[balancer]
192.168.160.2

[app]
192.168.160.3
192.168.160.4

[deployments:children]
balancer
app

[deployments:vars]
ansible_ssh_user=vagrant
ansible_ssh_private_key_file=~/.vagrant.d/insecure_private_key
```

With the inventory, we can choose to operate just the balancer, the app servers, or all our servers together. We can then create a playbook that will help us in provisioning the servers. This is shown in the code given below:

```
–
- hosts: balancer
sudo: yes
```

vars:

firewall_allowed_tcp_ports:

- “22”

- “80”

haproxy_backend_servers:

- name: 192.168.160.3

address: 192.168.160.3:80

- name: 192.168.160.4

address: 192.168.160.4:80

roles:

- computerguy.firewall

- computerguy.haproxy

- hosts: app

sudo: yes

vars:

firewall_allowed_tcp_ports:

“22”

- “80”

roles:

- computerguy.firewall

- computerguy.apache

Another playbook should be created alongside the “provision.yml” and given the name “deploy.yml”. Add the following code to the file:

–

- hosts: app

sudo: yes

serial: 1

pre_tasks:

- name: Disable backend server in the HAProxy.

haproxy:

state: disabled

host: ‘{{ inventory_hostname }}’

socket: /var/lib/haproxy/stats

backend: habackend

delegate_to: “{{ item }}”

with_items: groups.balancer

tasks:

- debug: msg=“Deployment would be done here.”

post_tasks:

- name: Wait for the backend to come up again.

wait_for:

host: ‘{{ inventory_hostname }}’

port: 80

state: started

timeout: 60

- name: Enable our backend server in the HAProxy.

haproxy:

state: enabled

host: '{{ inventory_hostname }}'

socket: /var/lib/haproxy/stats

backend: habackend

delegate_to: "{{ item }}"

with_items: groups.balancer

When it comes to the deployment, the above playbook will not do much, but it shows us how to perform a zero-downtime rolling update on two or even more application servers.

The playbook can then be executed on the local infrastructure by use of the following command:

$ ansible-playbook -i inventory playbooks/deploy.yml

The command should take only some minutes to complete execution, and the servers should then be set for the load balancer. It is a good idea to test whether the deployment playbook is functioning as expected. This can be addition of a task that will always fail, immediately after the debug task. This is shown in the code given below:

[…]

tasks:

- debug: msg=“Deployment can be done here.”

- command: /bin/false

post_tasks:

[…]

Chapter 2- Ad Hoc Commands

Ad-hoc commands can be used for performing tasks quickly and easily in Ansible.

Suppose we need to create a build infrastructure with vagrant for testing purposes. In one of your local drives, create a new folder, and in it, create a new file, giving it the name "*vagrantfile*". Use your favorite text editor to open the file and then add the following code into it:

 # -*- mode: ruby -*-

 # vi: set ft=ruby :

VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|

config.ssh.insert_key = false

config.vm.provider :virtualbox do |vb|

vb.customize ["modifyvm", :id, "—memory", "256"]

end

 # Application server 1.

config.vm.define "app1" do |app|

app.vm.hostname = "orc-app1.dev"

app.vm.box = “computerguy/centos7”

app.vm.network :private_network, ip: “192.168.160.4”

end

 # Application server 2.

config.vm.define “app2” do |app|

app.vm.hostname = “orc-app2.dev”

app.vm.box = “computerguy/centos7”

app.vm.network :private_network, ip: “192.168.160.5”

end

 # Database server.

config.vm.define “db” do |db|

db.vm.hostname = “orc-db.dev”

db.vm.box = “computerguy/centos7”

db.vm.network :private_network, ip: “192.168.160.6”

end

end

The above vagrantfile has been used for defining the three servers that are needed for management Each of these has been given a unique hostname, IP address and machine name. You can then open the terminal and navigate to the folder with the vagrantfile you have just created. You can then launch the Vagrant to begin building the three VMs by issuing the command “vagrant up” on the terminal.

Using single Inventory File for Multiple Servers

There are a number of ways in which you can tell the Ansible about the servers that are to be managed, but adding these to the main inventory file for the Ansible is the best and most convenient way to do this. The Ansible inventory file should be located at the directory "/etc/ansible/hosts". If you had not created one, make sure that you create one now. Ensure that the user is permitted to have write permissions on the file. The following code should then be added to the file:

```
# Lines starting with a # represent comments, and they are included only #for illustration purposes. The comments are an overkill for the most of our inventory
# Application servers
[app]
192.168.160.4
192.168.160.5

# Database server
[db]
192.168.160.6

# Group 'multi' with the servers
[multi:children]
app
db
```

 # Variables which will be applied to the servers

[multi:vars]

ansible_ssh_user=vagrant

ansible_ssh_private_key_file=~/.vagrant.d/insecure_private_key

We want to be sure that the Vagrant has configured our VMs using the right hostnames. This can be done using the “-a” flag on the “hostname”. This is shown in the command given below:

$ ansible multi -a “hostname”

The above will be executed by Ansible on each of the three servers and then return the results. Again, you may have noticed that your command was not executed in the way that you expected. You can run the command a few extra times and then observe the order. This is shown below:

 # First and Second run results:

192.168.160.5 | success | rc=0 >> 192.168.160.6 | success | rc=0 >>

orc-app2.dev orc-db.dev

192.168.160.6 | success | rc=0 >> 192.168.160.5 | success | rc=0 >>

orc-db.dev orc-app2.dev

192.168.160.4 | success | rc=0 >> 192.168.160.4 | success | rc=0 >>

orc-app1.dev orc-app1.dev

The default is that the Ansible will run your commands in a parallel manner, by use of

multiple process forks, and this will make the command complete more quickly. For those who are managing more than a few servers, this can be a bit slow. This is when you should run the command in a serial manner, one after the other.

You can now run the command once again, but this time, use the flag *"-f 1"* so as to tell Ansible to run the commands using only one fork. This is shown below:

$ ansible multi -a "hostname" -f 1

192.168.160.4 | success | rc=0 >>

orc-app1.dev

192.168.160.5 | success | rc=0 >>

orc-app2.dev

192.168.160.6 | success | rc=0 >>

orc-db.dev

You can try to run the same command a number of times, but you will realize that you will only get the same result and in the same order. Rarely will you run your commands in such a way, but you can feel free to adjust the number of fork processes that you need depending on the amount that your network connection and system can handle.

Learning the Environment

Since we know that Ansible is capable of setting hostnames in the way that we want, we should work to make sure that everything is in the proper order. Let's check to be sure that our servers have the disk space that we need. The following command can be used for this purpose:

$ ansible multi -a "df -h"

192.168.160.6 | success | rc=0 >>

Filesystem Size Used Avail Use% Mounted on

/dev/mapper/centos-root 19G 1014M 18G 6% /

devtmpfs 111M 0 111M 0% /dev

tmpfs 120M 0 120M 0% /dev/sjm

tmpfs 120M 4.3M 115M 4% /run

tmpfs 120M 0 120M 0% /sys/fs/cgroup

/dev/sda1 497M 124M 374M 25% /boot

none 233G 217G 17G 94% /vagrant

192.168.60.5 | success | rc=0 >>

Filesystem Size Used Avail Use% Mounted on

/dev/mapper/centos-root 19G 1014M 18G 6% /

devtmpfs 111M 0 111M 0% /dev

tmpfs 120M 0 120M 0% /dev/shm

tmpfs 120M 4.3M 115M 4% /run

tmpfs 120M 0 120M 0% /sys/fs/cgroup

/dev/sda1 497M 124M 374M 25% /boot

none 233G 217G 17G 94% /vagrant

192.168.60.4 | success | rc=0 >>

Filesystem Size Used Avail Use% Mounted on

/dev/mapper/centos-root 19G 1014M 18G 6% /

devtmpfs 111M 0 111M 0% /dev

tmpfs 120M 0 120M 0% /dev/shm

tmpfs 120M 4.3M 115M 4% /run

tmpfs 120M 0 120M 0% /sys/fs/cgroup

/dev/sda1 497M 124M 374M 25% /boot

none 233G 217G 17G 94% /vagrant

We have a lot of room now and our application is very lightweight. Let’s check to be sure that our servers have enough space:

$ ansible multi -a “free -m”

192.168.160.4 | success | rc=0 >>

total used free shared buffers cached

Mem: 248 167 50 4 1 79

-/+ buffers/cache: 116 121

Swap: 1155 0 1155

192.168.160.6 | success | rc=0 >>

total used free shared buffers cached

Mem: 238 190 47 4 1 72

-/+ buffers/cache: 116 121

Swap: 1155 0 1155

192.168.160.5 | success | rc=0 >>

total used free shared buffers cached

Mem: 238 186 52 4 1 67

-/+ buffers/cache: 116 121

Swap: 1155 0 1155

We should be somewhat conservative since we are running three VMs in the same server. It is now time for us to check and be sure that the time is in sync. The following command can be used for checking this:

$ ansible multi -a “date”

192.168.160.5 | success | rc=0 >>

Fri May 20 21:23:09 UTC 2021

192.168.160.4 | success | rc=0 >>

Fri May 20 21:23:09 UTC 2021

192.168.160.6 | success | rc=0 >>

Fri May 20 21:23:09 UTC 2021

Although some apps tolerate a slight difference for the time between the different servers, it is always worth ensuring that the servers have only a small difference in their time. The Network Time Protocol is the best way for us to do this, and it easy for us to configure this.

Use Ansible Modules to make changes

We should install the NTP, which will help us ensure the time is kept in sync. Instead of having to execute the command “yum install -y ntp” on each of our servers, we will make use of the yum module for Ansible to do it. This is shown below:

$ ansible multi -s -m yum -a “name=ntp state=installed”

After execution of the command, you should see three “*success*” messages that are an indication that the tool has been installed and is running as expected.

We should now ensure that the NTP daemon has been set and will run after booting. We will do this using the “*service*” module for Ansible as shown below:

$ ansible multi -s -m service -a “name=ntpd state=started enabled=yes”

The servers should then show us a success message as shown below:

“changed”: true,
“enabled”: true,
“name”: “ntpd”,
“state”: “started”

Running the same command will give you the same output, but the Ansible will tell you that nothing has changed, so the value for "*changed"* will become "*false"*.

The NTP server has the actual time, so the time on our servers should be synced to ensure that it matches this. We can do this as shown below:

$ ansible multi -s -a "service ntpd stop"

$ ansible multi -s -a "ntpdate -q 0.rhel.pool.ntp.org"

$ ansible multi -s -a "service ntpd start"

The "*ntpd"* service has to be stopped for the "*ntpdate"* command to work. That is what we have done with the above commands.

Configuring Application Servers

Since our web app is going to use Django, we must ensure that we have the Django and its dependencies already installed. Since this is not available in the CentOS yum repository, we can use the easy_install for Python so as to install this. This can be done as shown below:

$ ansible app -s -m yum -a “name=MySQL-python state=present”

$ ansible app -s -m yum -a “name=python-setuptools state=present”

$ ansible app -s -m easy_install -a “name=django”

We could also have used *“pip”* for installation of the Django. We can install this using easy install but this can be a bit complex, so stick to the above method. You should then check that the Django is running as expected. The following command can be used for this purpose:

$ ansible app -a “python -c ‘import django; print django.get_version()’”

192.168.160.4 | success | rc=0 >>

1.8

192.168.160.5 | success | rc=0 >>

1.8

Configuring Database Servers

We have used the "*app group"* for configuration of the application servers and this has been defined in the main inventory for Ansible, while the database server can be configured using the "*db"* group, which has been defined.

We are now in a position to install MriaDB and perform the necessary configuration so that the firewall of the server can access the default port for MariaDB, which is 3306. The following commands can be used for this purpose:

$ ansible db -s -m yum -a "name=mariadb-server state=present"
$ ansible db -s -m service -a "name=mariadb state=started enabled=yes"
$ ansible db -s -a "iptables -F"
**$ ansible db -s -a "iptables -A INPUT -s 192.168.160.0/24 -p tcp **
-m tcp —dport 3306 -j ACCEPT"

If you try to establish a connection from the app servers to the database at this point, it will be impossible, since the MariaDB expects a further configuration. We need to be able to allow one of the users from app server to be able to access MySQL. This can be done as follows:

$ ansible db -s -m yum -a "name=MySQL-python state=present"
**$ ansible db -s -m mysql_user -a "name=django host=% password=wxyz **
priv=*.*:ALL state=present"

At this point, it should be possible for you to create or deploy an application of Django in your app servers, and then direct it to the database server using the username as *"django"* and password as "*wxyz*".

At this point, you have a small web app that is running Django and MySQL. While we don't have a load balancer at the front of the app that will work to spread our requests, our configuration has been done much more easily and quickly and we did not have to log into the servers. Also, you are in a position to run your Ansible commands over and over again and there is no change.

It is now time for you to check the status of the ntpd. Just issue the command given below:

$ ansible app -s -a "service ntpd status"

You can then restart the service on the server that was affected. This can be done by issuing the following command:

$ ansible app -s -a "service ntpd restart" —limit "192.168.160.4"

We have used the flag "*—limit*" so as to limit our command to a certain host in the specified group. This will match either a regular expression or an exact string. If you only needed to apply the command to the .4 server, you could have done the following:

 # Limit hosts having a simple pattern (asterisk is the wildcard).

$ ansible app -s -a “service ntpd restart” —limit “*.4”

 # Limit hosts having a regular expression (prefix and a tilde).

$ ansible app -s -a “service ntpd restart” —limit ~”.*.4”

Although we have been using the IP addresses rather than the hostnames in our examples, you should note that in the real world, you would be using the hostnames.

User and Group Management

The Ad-hoc commands in Ansible are commonly used for the purpose of management of users and groups. The “*user”* and “*group”* modules in Ansible make things very standard across the different Linux distributions.

Begin by adding the “*admin”* group to your app server; your server administrators will use this. The following command can be used for this purpose:

$ ansible app -s -m group -a “name=admin state=present”

We want to add the user “*nicohsam”* to the app server with the group we have just created, and then assign a home folder to him in “*/home/nicohsam”* . The following command can be used for this purpose:

$ ansible app -s -m user -a “name=nicohsam group=admin createhome=yes”

In some cases, you may need to generate an SSH key for the same user. This can be done by simply adding the additional parameter “*generate_ssh_key=yes*”. If you need to set the user ID for the user, you can pass the parameter “*uid=[uid]”* into the command, while the user’s shell can be set using the parameter “*shell=[shell]”*, and the password using the parameter “*password=[encrypted-password]”*.

The user account can be deleted using the following command:

$ ansible app -s -m user -a “name=nicohsam state=absent remove=yes”

The information regarding a particular file can be checked by use of the following command:

$ ansible multi -m stat -a “path=/etc/environment”

Information about the file permissions, owner or MD5 can be printed by use of the above command, which is from the Ansible’s “*stat*” module. You should know that the result of the above command is similar to the one from executing the “*stat*” command, but the difference comes in that the result is passed in JSON form, which is easy for us to parse.

Sometimes, you may need to copy files and directories to a remote server. This is usually done using the “*scp*” or “*rsync*” commands, but since the “*rsync*” mopdule was just recently introduced in Ansible, this operation can easily be done by use of the Ansible’s “*copy*” module. This can be used as shown below:

$ ansible multi -m copy -a “src=/etc/hosts dest=/tmp/hosts”

In our case, the “*src*” represents either a file or a directory. In cases where a trailing trash is included, the command will copy only the contents of the directory into our destination.

If you leave the trash, then the directory, together with its contents, will be copied into the destination.

Note that module "*copy*" is very efficient when we are copying single files and small directories. In cases where you need to copy multiple files, we recommend that you copy and then expand using the archive of the files through the "*unarchive*" module for Ansible, or using the "*synchronize*" module in Ansible.

Sometimes, you may need to retrieve the files from the server. This can be done by use of the "*fetch*" module in Ansible, which works in the same way as the "*copy*" module, but in the reverse manner. When used, the files are copied to the local destination, in a directory that will match the directory from which you have copied the files.

Suppose that you are in need of grabbing the host file from the servers. The following command can be used for that purpose:

$ ansible multi -s -m fetch -a "src=/etc/hosts dest=/tmp"

By default, the fetch will keep the "*/etc/hosts*" file in the folder with the host name in the destination, and then in the location that has been defined by "*src*". This means that db file for the server will end up being stored at "*/tmp/192.168.160.6/etc/hosts*".

If you need the Ansible to fetch the files directly into the "/temp" *directory,* just use the

parameter *“flat=yes”* and then set”*dest=/temp”* as the final dest. However, for this to work, the file names have to be unique, meaning that the technique does not work when we are copying files from multiple servers. The parameter *“flat=yes”* should only be applied when we are copying files from a single host.

Sometimes, you may need to create files and directories, and then manage ownership and permissions on the files and directories and as well as creation of symlinks.

To create a directory, use the following command:

$ ansible multi -m file -a “dest=/tmp/test mode=644 state=directory”

To create a symlink, you can use the following command:

$ ansible multi -m file -a “src=/src/symlink dest=/dest/symlink \owner=root group=root state=link”

If you need to delete a file or a directory, then just set its state to *“absent”*. This is demonstrated in the example given below:

$ ansible multi -m file -a “dest=/tmp/test state=absent”

In Ansible, some of the operations will take a really long time to execute. In such cases,

the Ansible can be instructed to run the commands asynchronously, and the servers can be polled so that they can show once execution of a particular command has been completed. This operation is not useful if you are operating on a single server, but if you are operating multiple servers, the command can be executed on all of them and then the status polled until they are updated.

If you need to run a command in background, then you can set the options given below:

- -B <seconds>- this is the maximum of time for which the job will be allowed to run.

- -P <seconds>- this is the amount to wait for polling of servers for a job status which has been updated.

The servers can be updated asynchronously, while monitoring the progress. If we need to update our servers, we just have to execute the command *"yum -y update"* on all of them. If we leave out the "*–P*" flag, then the default behavior is that Ansible will poll after every 10 seconds. This is shown below:

$ ansible multi -s -B 3600 -a "yum -y update"

background launch…

192.168.160.6 | success >> {

"ansible_job_id": "543351239087",

“results_file”: “/root/.ansible_async/543351239087”,

“started”: 1

}

… [other hosts] …

After that, just wait for some time and you will observe the output related to the one given below:

<job 543351239087> finished on 192.168.160.6 => {

“ansible_job_id”: “543351239087”,

“changed”: true,

“cmd”: [

“yum”,

“-y”,

“update”

],

“delta”: “0:13:13.973892”,

“end”: “2022-03-09 05:47:48.229714”,

“finished”: 1,

… [more info and stdout from job] …

Even though the command is running in the background, it is possible for you to check for the status by use of the “*async_status”* module of Ansible. However, for this to work, you must have the “*ansible_job_id”* so as to pass it in the parameter “*jid”*. This is

demonstrated in the command given below:

$ ansible multi -m async_status -a “jid=543351239087”

Fire-and-forget tasks

Sometimes, you may have to execute maintenance scripts that may take a long time to complete, or some other tasks that take a long time to execute, and you don't need to sit and wait for the process to complete. In such a case, we recommend that you set the "*-B*" flag to the highest value that you need and then set the "*-P*" flag to "0", so that the Ansible will fire off the command and forget about it. This is shown below:

$ ansible multi -B 3600 -P 0 -a "/path/to/fire-and-forget-script.sh"
background launch…
192.168.160.5 | success >> {
"ansible_job_id": "114951925175",
"results_file": "/root/.ansible_async/114951925175",
"started": 1
}
… [other hosts] …
$

In this case, it will be impossible for you to track the progress by use of the "*jid*", but this mechanism is very helpful when it comes to "*fire-and-forget*" tasks. Tasks in the Ansible playbook can also be run in the background, but you have to define the "*async*" and "*poll*" parameters on our play.

The Log Files

Whenever you are diagnosing errors in Ansible, it is always a good idea to check the log files. The log file operations will work through the "*ansible"* command and a few caveats. Suppose we need to see the last few lines of our message log file in our servers. We can use the command given below:

$ ansible multi -s -a "tail /var/log/messages"

To filter the messages log, it will be impossible for you to use the default "*command"* module of Ansible, but you can use the "*shell"* as shown below:

**$ ansible multi -s -m shell -a "tail /var/log/messages | **

grep ansible-command | wc -l"

192.168.160.5 | success | rc=0 >>

11

192.168.160.4 | success | rc=0 >>

10

192.168.160.6 | success | rc=0 >>

14

The above command will show the number of commands that have been run on each of

our servers. Note that this will not be the same in each server, so you may end up getting different numbers.

Management of Cron Jobs

Cron is used for running periodic tasks via the crontab. To alter the changes on the cron job settings, you will have to log into the server, and use the command “crontab –e” under the account in which the cron job resides, then type the interval and the job itself.

Ansible provides us with the “*cron”* module hat makes it easy for us to manage cron jobs. Suppose you need to run a shell script at exactly 3:00pm on each day and on each server. This can be set as a cron job as follows:

**$ ansible multi -s -m cron -a “name=‘my-cron-all-servers’ **
hour=3 job=’/path/to/my-script.sh’”

If you don’t specify some values, then the Ansible will assume *.

Also, you may need to delete a cron job from your system. You can do this by use of the “*cron”* command, in which you will specify the name of the cron job, and then set the “*state=absent”*. This is shown in the command given below:

$ ansible multi -s -m cron -a “name=‘daily-cron-all-servers’ state=absent”

You can also use the Ansible for the purpose of management of custom crontab files. This can be done using the same syntax we have been using by inserting the parameter “*cron_file=cron_file_name*” for the purpose of specifying the location of the crontab file.

Chapter 3- Inventories

Consider the basic inventory file given below:

 # Inventory file located at /etc/ansible/hosts

 # Groups have been defined by use of square brackets (example. #[groupname]).

Each server in our group has been defined on its own line.

[ourapp]

www.ourapp.com

If you need to run the Ansible playbook on all the "*ourapp*" servers in the inventory, it can be set up as shown:

–

- hosts: ourapp

tasks:

[…]

In case you need to run an ad-hoc command on all the "*ourapp*" servers in our inventory, you can run the command as shown below:

 # Using ansible for checking the memory usage on the ourapp servers.

$ ansible ourapp -a “free -m”

The above example is useful for a single server, but in real life, multiple servers will be used and these are normally separated. Consider the inventory file given below, which belongs to a small web app that can be used in a real-life scenario:

 # Check.in servers for an individual server

[servercheck-web]

www1.servercheck.in

www2.servercheck.in

[servercheck-web:vars]

ansible_ssh_user=servercheck_svc

[servercheck-db]

db1.servercheck.in

[servercheck-log]

log.servercheck.in

[servercheck-backup]

backup.servercheck.in

[servercheck-nodejs]

atl1.servercheck.in

atl2.servercheck.in

nic1.servercheck.in

nic2.servercheck.in

nic3.servercheck.in

ned1.servercheck.in

ned2.servercheck.in

[servercheck-nodejs:vars]

ansible_ssh_user=servercheck_svc

foo=bar

 # The Server Check.in distribution based on groups.

[centos:children]

servercheck-web

servercheck-db

servercheck-nodejs

servercheck-backup

[ubuntu:children]

servercheck-log

Although the inventory may seem to be complex, when you look at it closely, you will realize that it represents a very simple architecture. If you need to update all the checkin servers for your servers, then use the following command:

$ ansible centos -m yum -a “name=bash state=latest”

You can go ahead and create a small playbook that can help you create a very small playbook to patch the vulnerability and then execute some tests so as to be sure that there was no vulnerability.

Consider the master playbook shown below, which can be used for the purpose of configuring the servers completely:

–

 # Setting up standardized, basic components across all the servers.

- hosts: all

sudo: true

roles:

- security

- logging

- firewall

 # Configuring the web application servers.

- hosts: servercheck-web

roles:

- nginx

- php

- servercheck-web

 # Configuring the database servers.

- hosts: servercheck-db

roles:

- pgsql

- db-tuning

 # Configuring the logging server.

- hosts: servercheck-log

roles:

- java

– elasticsearch

- logstash
- kibana

 # Configuring the backup server.
- hosts: servercheck-backup
roles:
- backup

 # Configuring the Node.js application servers.
- hosts: servercheck-nodejs
roles:
- servercheck-node

The example given below shows how variable definition can be done inside an inventory file:

[www]

 # defininition of host-specific variables can be done inline with the host.

www1.sample.com ansible_ssh_user=nicohsam

www2.sample.com

[db]

db1.sample.com

db2.sample.com

 # You may add ‘[group:vars]’ heading for creating variables which will #apply to the entire inventory group.

[db:vars]

ansible_ssh_port=5222

database_performance_mode=true

It is recommended that you don’t throw a lot of variables into your inventory file since the variables are less visible and they are mixed with the definition of the architecture.

host_vars

Consider a situation in which you have a directory layout that appears as shown below:

hostedapachesolr/

host_vars/

nyc1.hostedapachesolr.com

inventory/

hosts

main.yml

The file “inventory/hosts” will have a very simple definition of our servers by group. This is shown below:

[solr]

nic1.hostedapachesolr.com

nic2.hostedapachesolr.com

jap1.hostedapachesolr.com

[log]

log.hostedapachesolr.com

The Ansible will have to search for the file from wherever necessary and if it finds some

variables defined in it, then the variables will override all the facts that have been gathered from role variables and other playbooks.

group_vars

Consider the example given below:

hostedapachesolr/

group_vars/

solr

host_vars/

nyc1.hostedapachesolr.com

inventory/

hosts

main.yml

Note that it is just like the previous example, except we have added the *“group_vars”* directory.

Conclusion

We have come to the conclusion of this book. Ansible can be used for DevOps(Developer Operations). When working with Ansible in a group, you can apply DevOps for you to perform most of the tasks required in a smooth manner. It is always good to employ such mechanisms when working with Ansible ,as this will allow you to perform your tasks very smoothly. By now, you should be aware of the various strategies that can be used for deployment of apps. You should also understand how to do this when using either one or multiple servers.

As explained in this book, Ad-hoc commands can be used for the purpose of making most Ansible tasks quick and easy to do. To manage users and groups with ease, you can make use of the available Ad.hoc commands discussed in this book. The database servers, which are tough to configure, can be configured using ad-hoc commands very easily.

In Ansible, some commands will usually take longer to complete. Instead of babysitting these while waiting for them to complete, it is better to run these commands in the background and continue performing some other tasks. Management of cron jobs can also be done easily through the use of ad-hoc commands as has been discussed in this book. I hope you enjoyed reading!