Lab 6 - Ansible

Connectivity Instructions:

IP	Cluster IP
Username	Cluster User
Password	Cluster Pass

Lab Overview

In this lab participants will install, configure, and deploy the Ansible orchestration software stack on CentOS Server v7 VM. Once Ansible is stable, learners will develop and execute playbooks to deploy a LAMP stack and compare & contrast the differences with NuCalm. We'll also explore developing running a simple Ansible module to standup infrastructure (i.e. VM creation).

Introduction

Configuration management systems are designed to make controlling large numbers of servers easy for administrators and operations teams. They allow you to control many different systems in an automated way from one central location. While there are many popular configuration management systems available for Linux systems, such as Chef and Puppet, these are often more complex than many people want or need. Ansible is a great alternative to these options because it has a much smaller overhead to get started.

Ansible works by configuring client machines from an computer with Ansible components installed and configured. It communicates over normal SSH channels in order to retrieve information from remote machines, issue commands, and copy files. Because of this, an Ansible system does not require any additional software to be installed on the client computers. This is one way that Ansible simplifies the administration of servers. Any server that has an SSH port exposed can be brought under Ansible's configuration umbrella, regardless of what stage it is at in its life cycle.

Ansible takes on a modular approach, making it easy to extend to use the functionalities of the main system to deal with specific scenarios. Modules can be written in any language and communicate in standard JSON. Configuration files are mainly written in the YAML data serialization format due to its expressive nature and its similarity to popular markup languages. Ansible can interact with clients through either command line tools or through its configuration scripts called Playbooks.

In this lab, you'll install Ansible on a CentOS 7 server and learn some basics of how to use the software.

Step 1 - Environment Setup

To follow this tutorial, you will need:

- Deploy 2x CentOS v7 Servers (One will be used for Web server & One for DB server). Name the VM's: *WebServer* and *DBServer* respectively.
- Deploy 1x CentOS v7 VM to host Ansible. Name the VM *Ansible*. Follow the steps in configure-centos-server-v7 to create a non-root user.
- Make sure you can connect to the servers using a password-less connection/session.

Step 2 — Installing Ansible

To begin exploring Ansible as a means of managing our various servers, we need to install the Ansible software on at least one machine. In this lab we'll install ansible using *yum*, but to be fare to the learner, the Ansible stack can also be installed using *git* or *pip*.

To get Ansible for CentOS 7, first ensure that the CentOS 7 EPEL repository is installed:

```
$ sudo yum install epel-release
```

Once the repository is installed, install Ansible with yum:

```
$ sudo yum install ansible
```

Step 3 — Configuring Ansible Hosts

Ansible keeps track of all of the servers that it knows about through a *"hosts"* file. We need to set up this file first before we can begin to communicate with our other computers.

Open the file with root privileges like this:

```
$ sudo vi /etc/ansible/hosts
```

You will see a file that has a lot of example configurations commented out. Keep these examples in the file to help you learn Ansible's configuration if you want to implement more complex scenarios in the future.

The hosts file is fairly flexible and can be configured in a few different ways. The syntax we are going to use though looks something like this:

```
[group_name]
alias ansible_ssh_host=your_server_ip
```

The *group_name* is an organizational tag that lets you refer to any servers listed under it with one word. The alias is just a name to refer to that server.

Imagine you have three servers you want to control with Ansible. Ansible communicates with client computers through SSH, so each server you want to manage should be accessible from the Ansible server by typing:

```
$ ssh user@your_server_ip
```

You should **NOT** be prompted for a password. While Ansible certainly has the ability to handle password-based SSH authentication, SSH keys help keep things simple (see password-less configuration).

Let's set this up so that we can refer to these individually as host1 and host2, or as a group of servers. To configure this, you would add this block to your hosts file:

/etc/ansible/hosts

```
[servers]
host1 ansible_ssh_host=IP ADDRESS
host2 ansible_ssh_host=IP ADDRESS
```

Hosts can be in multiple groups and groups can configure parameters for all of their members. Let's try this out now.

Ansible will, by default, try to connect to remote hosts using your current username. If that user doesn't exist on the remote system, a connection attempt will result in this error:

```
Annsible connection error
host1 | UNREACHABLE! => {
    "changed": false,
    "msg": "Failed to connect to the host ia ssh.",
    "unreachable": true
}
```

Let's specifically tell Ansible that it should connect to servers in the "servers" group with the **ansible** user. Create a directory in the Ansible configuration structure called group_vars.

```
$ sudo mkdir /etc/ansible/group_vars
```

Within this folder, we can create YAML-formatted files for each group we want to configure:

```
$ sudo vi /etc/ansible/group_vars/servers
```

Note

Other text editors other than "vi" can be used as needed (i.e. nano, emacs, etc…). Caution: They may need to be installed.

Add this code to the file:

```
---
ansible_ssh_user: ansible
```

YAML files start with "—", so make sure you don't forget that part.

Save and close this file when you are finished. Now Ansible will always use the sammy user for the servers group, regardless of the current user.

If you want to specify configuration details for every server, regardless of group association, you can put those details in a file at /etc/ansible/group_vars/all. Individual hosts can be configured by creating files under a directory at /etc/ansible/host_vars.

Step 4 — Using Simple Ansible Commands

Now that we have our hosts set up and enough configuration details to allow us to successfully connect to our hosts, we can try out our very first command.

Ping all of the servers you configured by typing:

```
$ ansible -m ping all
```

Ansible will return output like this:

```
Output
host1 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}

host2 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}

host3 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

This is a basic test to make sure that Ansible has a connection to all of its hosts.

The -m ping portion of the command is an instruction to Ansible to use the “ping” module. These are basically commands that you can run on your remote hosts. The ping module operates in many ways like the normal ping utility in Linux, but instead it checks for Ansible connectivity.

The all portion means “all hosts.” You could just as easily specify a group:

```
$ ansible -m ping servers
```

You can also specify an individual host:

```
$ ansible -m ping host1
```

You can specify multiple hosts by separating them with colons:

```
$ ansible -m ping host1:host2
```

The shell module lets us send a terminal command to the remote host and retrieve the results. For instance, to find out the memory usage on our host1 machine, we could use:

```
$ ansible -m shell -a 'free -m' host1
```

As you can see, you pass arguments into a script by using the -a switch. Here’s what the output might look like:

```
Output
host1 | SUCCESS | rc=0 >>
            total        used        free      shared  buff/cache   available
Mem:         3765         295        1712          16        1757        3181
Swap:        1023           0        1023
```

By now, you should have your Ansible server configured to communicate with the servers that you would like to control. You can verify that Ansible can communicate with each host you know how to use the ansible command to execute simple tasks remotely.

Although this is useful, we have not covered the most powerful feature of Ansible in this lab: **Playbooks.** You have configured a great foundation for working with your servers through Ansible, so your next step is to learn how to use Playbooks to do the heavy lifting for you.

Step 5 - Preparing The System for Development - Installing Python

Installation of Python on CentOS consists of a few (simple) stages, starting with updating the system, followed by getting any desired version of Python, and proceeding with the set up process.

Remember: You can see all available releases of Python by checking out the Releases page. Using the instructions here, you should be able to install any or all of them.

Note

This guide should be valid for CentOS version 7 as well as 6.x and 5.x.

Updating The Default CentOS Applications

Before we begin with the installation, let’s make sure to update the default system applications to have the latest versions available.

Run the following command to update the system applications:

```
$ sudo yum -y update
```

Preparing The System for Development Installations

CentOS distributions are lean - perhaps, a little too lean - meaning they do not come with many of the popular applications and tools that you are likely to need.

This is an intentional design choice. For our installations, however, we are going to need some libraries and tools (i.e. development [related] tools) not shipped by default. Therefore, we need to get them downloaded and installed before we continue.

There are two ways of getting the development tools on your system using the package manager yum:

Option #1 (not recommended): Consists of downloading these tools (e.g. make, gcc etc.) one-by-one. It is followed by trying to develop something and highly-likely running into errors midway through - because you will have forgotten another package so you will switch back to downloading.

The recommended and sane way of doing this is following **Option #2:** Simply downloading a bunch of tools using a single command with yum software groups.

YUM Software Groups

YUM Software Groups consist of bunch of commonly used tools (applications) bundled together, ready for download all at the same time via execution of a single command and stating a group name. Using YUM, you can even download multiple groups together.

The group in question for us is the Development Tools.

How to Install Development Tools using YUM on CentOS

In order to get the necessary development tools, run the following:

```
$ sudo yum groupinstall -y development
```

or;

```
$ sudo yum groupinstall -y 'development tools'
```

Note

The former (shorter) version might not work on older distributions of CentOS.

To download some additional packages which are handy:

```
$ sudo yum install -y zlib-dev openssl-devel sqlite-devel bzip2-devel
```

Remember: Albeit optional, these “handy” tools are very much required for most of the tasks that you will come across in future. Unless they are installed in advance, Python, during compilation, will not be able to link to them.

Step 6 - Run Ansible Playbook to Deploy LAMP stack

These playbooks require Ansible 1.2 or greater

These playbooks are meant to be a reference and starter’s guide to building Ansible Playbooks. These playbooks were tested on CentOS 7.x so we recommend that you use CentOS Server v7 to test these modules.

Download the playbook.tar (see link below) and copy it to directory /etc/ansible/ on the server hosting Ansible.

`playbooks.tar`

Extract the archive as follows:

```
$ tar -xzvf calm_workshop_lab6_lamp_example.tar.gz
```

CentOS v7 reflects playbook changes in:

1. Network device naming scheme has changed
2. iptables is replaced with firewalld
3. MySQL is replaced with MariaDB

This LAMP stack can be on a single node or multiple nodes. The inventory file ‘hosts’ defines the nodes in which the stacks should be configured.

```
[webservers]
 ntnxwebhost ansible_ssh_host=IP ADDRESS

[dbservers]
 ntnxdbhost ansible_ssh_host=IP ADDRESS
```

Here the [webservers] would be configured on the ntnxweb host and the [dbservers] on a server called ntnxdbhost. The stack can be deployed using the following command:

```
$ ansible-playbook -i hosts site.yml
```

Once done, you can check the results by browsing to http://ntnxwebhost/index.php. You should see a simple test page and a list of databases retrieved from the database server.

Note

Replace http://ntnxwebhost/index.php with the ip-address of your webserver vm. e.g. if your websever ip-address is 10.21.68.92 you would use http://10.21.68.92/index.php

If successfull, your browser should connect to the new webserver and display the following message:

```
Homepage_
Hello, World! I am a web server configured using Ansible and I am :
CentOS.localdomain
List of Databases:
information_schema foodb mysql performance_schema test
```

Click on the hyperlink Homepage displayed in the browser. The browser should display the following message:

```
Hello Calm Workshop! My App deployed via Ansible...
```

Ansible Architecture

Ansible is a radically simple IT automation engine that automates cloud provisioning, configuration management, application deployment, intra-service orchestration, and many other IT needs.

Being designed for multi-tier deployments since day one, Ansible models your IT infrastructure by describing how all of your systems inter-relate, rather than just managing one system at a time.

It uses no agents and no additional custom security infrastructure, so it’s easy to deploy - and most importantly, it uses a very simple language (YAML, in the form of Ansible Playbooks) that allow you to describe your automation jobs in a way that approaches plain English.

In this section, we’ll give you a really quick overview of how Ansible works so you can see how the pieces fit together.

Modules

Ansible works by connecting to your nodes and pushing out small programs, called “Ansible Modules” to them. These programs are written to be resource models of the desired state of the system. Ansible then executes these modules (over SSH by default), and removes them when finished.

Your library of modules can reside on any machine, and there are no servers, daemons, or databases required. Typically you’ll work with your favorite terminal program, a text editor, and probably a version control system to keep track of changes to your content.

Plugins

Plugins are pieces of code that augment Ansible’s core functionality. Ansible ships with a number of handy plugins, and you can easily write your own.

Inventory

By default, Ansible represents what machines it manages using a very simple INI file that puts all of your managed machines in groups of your own choosing.

To add new machines, there is no additional SSL signing server involved, so there’s never any hassle deciding why a particular machine didn’t get linked up due to obscure NTP or DNS issues.

If there’s another source of truth in your infrastructure, Ansible can also plugin to that, such as drawing inventory, group, and variable information from sources like EC2, Rackspace, OpenStack, and more.

Here’s what a plain text inventory file looks like:

```
---
[webservers]
 www1.example.com
 www2.example.com

[dbservers]
 db0.example.com
 db1.example.com
```

Once inventory hosts are listed, variables can be assigned to them in simple text files (in a subdirectory called ‘group_vars/’ or ‘host_vars/’ or directly in the inventory file. Or, as already mentioned, use a dynamic inventory to pull your inventory from data sources like EC2, Rackspace, or OpenStack.

Playbooks

Playbooks can finely orchestrate multiple slices of your infrastructure topology, with very detailed control over how many machines to tackle at a time. This is where Ansible starts to get most interesting. Ansible's approach to orchestration is one of finely-tuned simplicity, as we believe your automation code should make perfect sense to you years down the road and there should be very little to remember about special syntax or features. Here's what a simple playbook looks like:

```
---
- hosts: webservers
  serial: 5 # update 5 machines at a time
  roles:
    - common
    - webapp

    - hosts: content_servers
  roles:
    - common
    - content
```

Extending Ansible with Plug-ins and the API

Should you want to write your own, Ansible modules can be written in any language that can return JSON (Ruby, Python, bash, etc). Inventory can also plug in to any datasource by writing a program that speaks to that datasource and returns JSON. There's also various Python APIs for extending Ansible's connection types (SSH is not the only transport possible), callbacks (how Ansible logs, etc), and even for adding new server side behaviors.

Ansible Production Modules

Ansible ships with a number of modules (called the 'module library') that can be executed directly on remote hosts or through Playbooks.

Users can also write their own modules. These modules can control system resources, like services, packages, or files (anything really), or handle executing system commands.

Lab 7 - Docker

Connectivity Instructions:

IP	Cluster IP
Username	Cluster User
Password	Cluster Pass

Lab Overview

In this lab participants will learn how to install and configure Docker. Once Docker is installed and stable, participants will learn to create and deploy a container… Once completed, participants will containerize and deploy a 3 Tier LAMP application.

Requirements:

- CentOS Server v7 VM created: centos-setup guide for docker.
- Docker-CE 17.x Installed: docker-installation guide for docker.
- NTNX Docker Plugin Installed: ntnx-plugin-setup guide for docker.
- Docker Hub Account: https://hub.docker.com/

Glossary:

- **Image:** An image is a lightweight, stand-alone, executable package that includes everything needed to run a piece of software, including the code, a runtime, libraries, environment variables, and config files.
- **Dockerfile:** Dockerfile will define what goes on in the environment inside your container.
- **Container:** A container is a runtime instance of an image—what the image becomes in memory when actually executed. It runs completely isolated from the host environment by default, only accessing host files and ports if configured to do so.
- **Service:** In a distributed application, different pieces of the app are called "services." For example, if you imagine a video sharing site, it probably includes a service for storing application data in a database, a service for video transcoding in the background after a user uploads something, a service for the front-end, and so on. Services are really just "containers in production." A service only runs one image, but it codifies the way that image runs—what ports it should use, how many replicas of the container should run so the service has the capacity it needs, and so on. Scaling a service changes the number of container instances running that piece of software, assigning more computing resources to the service in the process.

Create Dockerfile:

Create an empty directory. Change directories (cd) into the new directory, create a file called Dockerfile, copy-and-paste the following content into that file, and save it. Take note of the comments that explain each statement in your new Dockerfile.

```
# Use an official Python runtime as a parent image
FROM python:2.7-slim

# Set the working directory to /app
WORKDIR /app

# Copy the current directory contents into the container at /app
ADD . /app

# Install any needed packages specified in requirements.txt
```

```
RUN pip install --trusted-host pypi.python.org -r requirements.txt

# Make port 80 available to the world outside this container
EXPOSE 80

# Define environment variable
ENV NAME World

# Run app.py when the container launches
CMD ["python", "app.py"]
```

Proxy servers can block connections to your web app once it’s up and running. If you are behind a proxy server, add the following lines to your Dockerfile, using the ENV command to specify the host and port for your proxy servers:

```
# Set proxy server, replace host:port with values for your servers
ENV http_proxy host:port
ENV https_proxy host:port
```

This Dockerfile refers to a couple of files we haven’t created yet, namely app.py and requirements.txt. Let’s create those next.

Create the Application:

Create two more files, **requirements.txt** and **app.py**, and put them in the same folder with the Dockerfile. This completes our app, which as you can see is quite simple. When the above Dockerfile is built into an image, app.py and requirements.txt will be present because of that Dockerfile’s ADD command, and the output from app.py will be accessible over HTTP thanks to the EXPOSE command.

requirements.txt

```
Flask
Redis
```

app.py

```
from flask import Flask
from redis import Redis, RedisError
import os
import socket

# Connect to Redis
redis = Redis(host="redis", db=0, socket_connect_timeout=2, socket_timeout=2)

app = Flask(__name__)

@app.route("/")
def hello():
    try:
        visits = redis.incr("counter")
    except RedisError:
        visits = "<i>cannot connect to Redis, counter disabled</i>"

    html = "<h3>Hello {name}!</h3>" \
           "<b>Hostname:</b> {hostname}<br/>" \
           "<b>Visits:</b> {visits}"
    return html.format(name=os.getenv("NAME", "nucalm"),
hostname=socket.gethostname(), visits=visits)
```

```
if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80)
```

Now we see that *pip install -r requirements.txt* installs the Flask and Redis libraries for Python, and the app prints the environment variable NAME, as well as the output of a call to *socket.gethostname()*. Finally, because Redis isn't running (as we've only installed the Python library, and not Redis itself), we should expect that the attempt to use it here will fail and produce the error message.

Note: Accessing the name of the host when inside a container retrieves the container ID, which is like the process ID for a running executable.

That's it! You don't need Python or anything in requirements.txt on your system, nor will building or running this image install them on your system. It doesn't seem like you've really set up an environment with Python and Flask, but you have.

Build the Application

We are ready to build the app. Make sure you are still at the top level of your new directory. Here's what ls should show:

```
$ ls
  Dockerfile          app.py                  requirements.txt
```

Now run the build command. This creates a Docker image, which we're going to tag using -t so it has a friendly name.

```
$ docker build -t calmWorkshop .
```

Where is your built image? It's in your machine's local Docker image registry:

```
$ docker images

  REPOSITORY          TAG                 IMAGE ID
  calmWorkshop        latest              326387cea398
```

Tip: You can use the commands docker images or the newer docker image ls list images. They give you the same output.

Run the Application

Run the app, mapping your machine's port 4000 to the container's published port 80 using -p:

```
$ docker run -p 4000:80 calmWorkshop
```

You should see a message that Python is serving your app at http://0.0.0.0:80. But that message is coming from inside the container, which doesn't know you mapped port 80 of that container to 4000, making the correct URL http://localhost:4000.

Go to that URL in a web browser to see the display content served up on a web page, including "Hello World" text, the container ID, and the Redis error message.

*You can also use the curl command in a shell to view the same content.

```
$ curl http://localhost:4000

  <h3>Hello nucalm!</h3><b>Hostname:</b> 8fc990912a14<br/><b>Visits:</b>
<i>cannot connect to Redis, counter disabled</i>
```

This port remapping of 4000:80 is to demonstrate the difference between what you EXPOSE within the Dockerfile, and what you publish using docker run -p. In later steps, we'll just map port 80 on the host to port 80 in the container and use http://localhost.

Hit CTRL+C in your terminal to quit.

Now let's run the app in the background, in detached mode:

```
$ docker run -d -p 4000:80 calmWorkshop
```

You get the long container ID for your app and then are kicked back to your terminal. Your container is running in the background. You can also see the abbreviated container ID with docker container ls (and both work interchangeably when running commands):

```
$ docker container ls
  CONTAINER ID        IMAGE               COMMAND             CREATED
  1fa4ab2cf395        calmWorkshop        "python app.py"     28 seconds ago
```

You'll see that CONTAINER ID matches what's on http://localhost:4000.

Now use docker container stop to end the process, using the CONTAINER ID, like so:

```
$ docker container stop 1fa4ab2cf395
```

Image sharing

To demonstrate the portability of what we just created, let's upload our built image and run it somewhere else. After all, you'll need to learn how to push to registries when you want to deploy containers to production.

A registry is a collection of repositories, and a repository is a collection of images—sort of like a GitHub repository, except the code is already built. An account on a registry can create many repositories. The docker CLI uses Docker's public registry by default.

Note: We'll be using Docker's public registry here just because it's free and pre-configured, but there are many public ones to choose from, and you can even set up your own private registry using Docker Trusted Registry.

Log in with your Docker ID

If you don't have a Docker account, sign up for one at cloud.docker.com. Make note of your username.

Log in to the Docker public registry on your local machine.

```
$ docker login
```

Tag the image

The notation for associating a local image with a repository on a registry is username/repository:tag. The tag is optional, but recommended, since it is the mechanism that registries use to give Docker images a version. Give the repository and tag meaningful names for the context, such as get-started:part2. This will put the image in the get-started repository and tag it as part2.

Now, put it all together to tag the image. Run docker tag image with your username, repository, and tag names so that the image will upload to your desired destination. The syntax of the command is:

```
$ docker tag image username/repository:tag
```

For example:

```
$ docker tag calmWorkshop dogfish/get-started:part2
```

Run docker images to see your newly tagged image. (You can also use docker image ls.)

```
$ docker images
  REPOSITORY             TAG                 IMAGE ID            CREATED
SIZE
  almWorkshop            latest              d9e555c53008        3 minutes ago
195MB
  dogfish/get-started    part2               d9e555c53008        3 minutes ago
195MB
  python                 2.7-slim            1c7128a655f6        5 days ago
183MB
  ...
```

Publish the image

Upload your tagged image to the repository:

```
$ docker push username/repository:tag
```

Once complete, the results of this upload are publicly available. If you log in to Docker Hub, you will see the new image there, with its pull command.

Pull and run the image from the remote repository From now on, you can use docker run and run your app on any machine with this command:

```
$ docker run -p 4000:80 username/repository:tag
```

If the image isn’t available locally on the machine, Docker will pull it from the repository.

```
$ docker run -p 4000:80 dogfish/get-started:part2
  Unable to find image 'dogfish/get-started:part2' locally
  part2: Pulling from dogfish/get-started
  10a267c67f42: Already exists
  f68a39a6a5e4: Already exists
  9beaffc0cf19: Already exists
  3c1fe835fb6b: Already exists
  4c9f1fa8fcb8: Already exists
  ee7d8f576a14: Already exists
  fbccdcced46e: Already exists
  Digest:
sha256:0601c866aab2adcc6498200efd0f754037e909e5fd42069adeff72d1e2439068
  Status: Downloaded newer image for dogfish/get-started:part2
```

```
 * Running on http://0.0.0.0:80/ (Press CTRL+C to quit)
```

Note: If you don’t specify the :tag portion of these commands, the tag of :latest will be assumed, both when you build and when you run images. Docker will use the last version of the image that ran without a tag specified (not necessarily the most recent image).

No matter where docker run executes, it pulls your image, along with Python and all the dependencies from requirements.txt, and runs your code. It all travels together in a neat little package, and the host machine doesn’t have to install anything but Docker to run it.