16/12/19

**Linux(Bash)**

Bash file should start with:

#!/bin/bash

\*/bin/bash can be replaced by the which bash you use

Bash file not necessarily have to have a .sh extension, it’s visually more clear that a file with .sh is a bash file.

Environment variables in scripts:

Start with “$”, use “env” to check environment variable

Exp: $HOME in bash file

Don’t overwrite existing variables

Environment variables are should be uppercase

Exp:

FIRSTNAME = “Liang” (no space around equal sign)

Export FIRSTNAME

**[Export]** is a built-in command of the Bash shell. It is used to mark variables and functions to be passed to child processes. Basically, a variable will be included in child process environments without affecting other environments.

<https://www.journaldev.com/28251/export-command-linux>

In simple terms, environment variables are set when you open a new shell session. at any time if you change any of the variable values, the shell has no way of picking that change. The export command, on the other hand, provides the ability to update the current shell session about the change you made to the exported variable. You don’t have to wait until new shell session to use the value of the variable you changed.

<https://www.geeksforgeeks.org/export-command-in-linux-with-examples/>

**export FIRSTNAME=”Liang” (** “” stores normal value**)**

**export TODAYDATE=`date`** (``back ticks store the results of using command **date,** \*run the same command again to update date value)

COMMENTS in bash scripts

Start with # sign

Exp:

#COMMENTS

**Docker**

In docker-compose file

“Build .” build from DockerFile in current location

Swarm Mode

A server clustering service

To check swarm is active or not: **docker swarm | grep Swarm**

By default, swarm is inactive, to active swarm: **docker swarm init** (create a manager node)

Create a new service:

**docker service create [OPTIONS] IMAGE [COMMAND] [ARG...]**

Exp:

docker service create alpine ping 8.8.8.8 (creates 1 alpine replica)

ID NAME MODE REPLICAS IMAGE PORTS

mjfnz592fd8g awesome\_bohr replicated 1/1 alpine:latest

**Usage: docker update [OPTIONS] CONTAINER [CONTAINER...]**

Update configuration of one or more containers

**Usage: docker service update [OPTIONS] SERVICE**

Update a service

When one of the containers failed, Swarm will automatically start a new container.

Steps:

1. docker swarm init --advertise-addr 192.168.0.48(current manager node IP) on manager node
2. docker swarm join --token SWMTKN-1-0v9wupgq9ni1m0t9y83b8g0xo9vz3g7ntoune0w6ldo5bcf93k-4zdxl7xqxmlb87xbk7nydawb9 192.168.0.48:2377 (Join swarm as a worker) on worker node
3. docker node ls(check node list)
4. Change a worker to Manager node:
   1. docker node update –role manager xxx(node name)
   2. docker swarm join-token manager(manage node) to get command for changing worker node to manager node(worker node)
5. docker service create --replicas 3 alpine ping 8.8.8.8 (create a servicewith 3 replicas on manager node)

\* docker node

\* docker swarm

\* docker service

what’s the relation between node, swarm and service?

A **node** is an instance of the Docker engine participating in the swarm.

A **task** is a running container which is part of a swarm service and managed by a swarm manager, as opposed to a standalone container.

A **service** is the definition of the tasks to execute on the manager or worker nodes. It is the central structure of the swarm system and the primary root of user interaction with the swarm. When you create a service, you specify which container image to use and which commands to execute inside running containers. In the replicated services model, the swarm manager distributes a specific number of replica tasks among the nodes based upon the scale you set in the desired state.

<https://docs.docker.com/engine/swarm/key-concepts/>

swarm mode that enables you to create a cluster of one or more Docker Engines called a swarm. A swarm consists of one or more nodes.

<https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/>

What’s the difference between a work node and a manager node?

To deploy your application to a swarm, you submit a service definition to a **manager node**. The manager node dispatches units of work called tasks to worker nodes. Manager nodes also perform the orchestration and cluster management functions required to maintain the desired state of the swarm. Manager nodes elect a single leader to conduct orchestration tasks. **Worker nodes** receive and execute tasks dispatched from manager nodes. By default, manager nodes also run services as worker nodes, but you can configure them to run manager tasks exclusively and be manager-only nodes.

17/12/19

**Linux(Bash)**

Command Substitution, two methods

1. static data:

USERFILES=`find /home –user adminuser`

Echo “All Files Owned by adminuser:$USERFILES”

1. dynamic data:

shopt –s expand\_aliases

alias TODAY=”date”

D=TODAY

Echo “With Alias, TODAY is: $D”

**Shopt –s expand\_aliases** (expand alias in sub shell, alias only exist in sub shall -> more secure?)

Exit Status

**Echo $? (**check last command’s exit status, command success returns 0, if return not 0, means failed/error**)**

**Set –e (**when a bash file occurs an error, stop the process, exit the shell, this is to prevent the wrong script overwrite important values/scripts**)**

Arithmetic Operations (add, subtract, multiply, divide …)

**Expr expression**

Exp:

**Expr 2 + 2 (put space around “+” sign)**

Use “\\*” times to avoid “\*” wildcard, same happens to “()”, it should be “\( 2 + 2 \) / 4” result is 1

Global and Local Environment Variables

**Env and printenv (**Check global environment variables**)**

**Set (**Check Local environment variables, only specific to current login user**)**

**Docker**

Overlay Multi-Host Networking

**--driver overlay**

For container-to-container traffic inside a single Swarm

Exp:

docker network create --driver overlay mydrupal (create an overlay network called mydrupal)

docker service create --name drupal --network mydrupal -p 80:80 drupal

docker service create --name psql --network mydrupal -e POSTGRES\_PASSWORD=mypass postgres

(drupal and postgres can talk to each other)

Routing Mesh

Routes ingress(incoming) packets for a Service to proper Task, spands all nods in Swarm. It uses IPVS from Linux Kernel. Acts like a Load Balance across all tasks.

Two ways:

1. container-to-container in an overlay network (uses VIP)
2. External traffic incoming to published ports (all nodes listen)

Demo commands:

Swarm:

docker swarm init --advertise-addr 192.168.0.8 ( two nodes joined as worker nodes)

Networks:

docker network create -d overlay backend

docker network create -d overlay frontend

Services:

docker service create --name vote -p 80:80 --network frontend --replicas 2 bretfisher/examplevotingapp\_vote

docker service create --name redis --network frontend redis:3.2

docker service create --name db --network backend --mount type=volume,source=db-data,target=/var/lib/postgresql/data postgres:9.4

docker service create --name worker --network frontend --network backend bretfisher/examplevotingapp\_worker:java

docker service create --name result --network backend -p 5001:80 bretfisher/examplevotingapp\_result

Stacks

Production grade compose

Stacks accept Compose files as their declarative definition for services, networks and volumes

Stack manages all those objects for users, including overlay network per stack.

**Docker stack deploy**

Exp: Docker stack deploy –c xxxx.yml xxx

(Can’t do build just like in docker-compose.yml file, this is because building shouldn’t happen on production Swarm)

on local machine Compose ignores deploy, Swarm ignores build

Stack reads docker-compose file without needing docker-compose CLI

Once use a configuration file to manage all the configurations, only change the conguration file.

To update a current stack, change the configuration file(.yml), then

Docker stack deploy –c xxxx.yml xxx (same as the first time deploy stack, the system will recognise the changes in the file)

Secrets Storage

Secure solution for storing secrets in Swarm

What is a Secret?

1. Usernames and passwords
2. TLS certificates and keys
3. SSH keys

Secret only in Swarm, but local docker-compose can use file-based secrets, but not secure (testing purpose?)

Secrets are first stored in Swarm, then assigned to a Service(s)

Only stored on disk on Manager nodes(encrypted) as key-value pair, can have alias

Exp:

/run/secrets/<secret-name> or /run/secrets/<secret-alias>

Two ways to create secrets:

1. Have a .txt file with secret value, run command **docker secret create xxx(secret variable) xx.txt**
2. **Echo “xxxx”(secret value) | docker secret create xxx(secret variable) –**

both will restore secrets locally, which is not recommended

Can also have secret files already and associated in stack(docker-compose.yml) file.

18/12/19

**Linux(Bash)**

Special Characters - Quotes and Escapes

Back slash **\** escapes a single character

Exp:

Echo “\$HOME” gets result-> $HOME (if no \, gets the current user home directory)

Single quote **‘’** escapes more than one character

Exp:

Echo ‘$HOME $PWD’ gets result -> $HOME $PWD

Double quotes **“”** escape first character

/dev/null

anything write to /dev/null will be ignored

echo “abc” >>/dev/null

no return value

Read Statement

**Read**

Exp:

In a shell script

Echo “Entre your first name:”

Read FIRSTNAME

Echo “Your first name is $FIRSTNAME”

Shell Expansion

Echo sh{ot,ort,oot} -> shot short shoot

Echo st{il,al}l -> still stall

Echo ~ -> /root (~ represents user home directory)

Echo ~+ (current working directory, same with PWD)

Echo ~- (Old PWD)

Echo “${!HO\*}” (what does **$** and **!** mean in here?)

The ‘**$**’ character introduces parameter expansion, command substitution, or arithmetic expansion.

${!prefix\*}

${!prefix@}

Expands to the names of variables whose names begin with prefix, separated by the first character of the IFS special variable. When ‘@’ is used and the expansion appears within double quotes, each variable name expands to a separate word.

<https://www.gnu.org/software/bash/manual/html_node/Shell-Expansions.html>

Echo and sign value at one time

Echo “${FIRSTNAME:=LIANG}”

return LIANG

Types of Variables (Implicit vs. Explicit Definition)

Declare

**Declare –i xxx=xxx** (implicitly declare a variable as an integer)

**Declare +i xxx** (to unset a implicitly signed variable type, for integer)

**Declare –r xxx=xxx** or **readyonly xxx=xxx** (declare a read-only variable)

Arrays

Declare an array:

ArrayName=(value1 value2 value3 …)

No “,” between values like in programming languages

Use index to see a value

Echo ${ArrayName[0]} (echo first value, just like in all programming languages)

Echo ${ArrayName[\*]} (echo all values)

<https://www.geeksforgeeks.org/array-basics-shell-scripting-set-1/>

**Docker**

App lifecycle with docker-compose

If docker-compose.override.yml and docker-compose.yml in same file, system will pick docker-compose.yml first then use docker-compose.override.yml to override docker-compose.yml file.

**docker stack deploy --compose-file docker-compose.yml vossibility** (a single docker compose file)

**docker stack deploy --compose-file docker-compose.yml -c docker-compose.prod.yml vossibility** (a base docker compose file + environment-specific override compose file)

<https://docs.docker.com/engine/reference/commandline/stack_deploy/>

Service Updates

**Docker service update –image xxapp:1.0.1 <servicename>** (updates a service with an newer image)

**Docker service update –en-add NODE\_ENV=production –publish-rm 8080** (Adding env variable and remove a port in one command)

**Docker service scale web=8 api=6** (Change number of replicas of two services)

**Docker stack deploy –c xxx.yml <stackname>** (Update stack using newer yml file)

<https://docs.docker.com/engine/reference/commandline/service_update/>

**Docker service update –force xx** (Balancing service, force service update)

**Docker-Kubernetes**

Swarm vs Kubernetes

Swarm: easier to deploy/manage

Kubernetes: more features and flexibility

**Advantages of Swarm:**

1. Comes with Docker, single vendor container platform
2. Easiest orchestrator to deploy/manage yourself
3. Follows 80/20 rule, 20% of features for 80% of use cases
4. Runs anywhere Docker does: local, Cloud, datacentre; ARM, Windows, 32bit
5. Secure by default
6. Easier to troubleshoot

**Advantages of Kubernetes:**

1. Clouds will deploy/manage K8S for you
2. Infrastructure vendors are making their own distributions
3. Widest adoption and community
4. Flexible: Covers widest set of use cases
5. “K8S first” vendor support

K8S basic terms:

K8S: the whole orchestration system

Kubectl: CLI to configure K8S and manage apps

Node: Single server in the K8S cluster (Similar to Worker)

A Pod always runs on a Node. A Node is a worker machine in Kubernetes and may be either a virtual or a physical machine, depending on the cluster. Each Node is managed by the Master. A Node can have multiple pods, and the Kubernetes master automatically handles scheduling the pods across the Nodes in the cluster.

<https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/>

Kubelet: K8S agent running on nodes

Control Plane: Set of containers (Master, similar to Manager in Swarm) that manage the cluster, includes API server, scheduler, controller manager, etcd and more

Pod: one or more containers running together on one Node. Containers are always in pods

A Pod is a Kubernetes abstraction that represents a group of one or more application containers (such as Docker or rkt), and some shared resources for those containers.

<https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/>

Controller: for creating/updating pods and other objects. Types of controllers: deployment, replicaSet, StatefulSet, DaemoSet, Job, CronJob …

Services: network endpoint to connect to pod

Namespace: Filtered group of objects in cluster

Container(s) -> Pod -> Node (Worker) -> Control Plane (Master)

Two ways to deploy Pods(containers) in K8S: commands or YAML

Use command:

Kubectl version (Check is K8S is working, should have two versions)

Kubectl run my-nginx –image nginx (run a pod of the nginx web server)

Kubectl get pods (list the pods)

Kubectl get all (see all objects)

When a pod is created, a deployment controller is also created, then replicaSet controller is also created. Deployment controller manages replicaSet controller, replicaSet manages pods.

**$ deployment NAME --image=image [--dry-run]**

Kubectl delete deployment my-nginx (similar to remove(rm) in docker, it deletes deployment controller, replicaSet controller and dops)

Scaling ReplicaSets

Kubectl scale deploy/my-apache --replicas 2

Kubectl scale deployment my-apache --replicas 2

Above two commands are same, in K8S, deploy=deployment=deployments

Kubectl get pods

Kubectl logs deployment/my-apache --follow --tail 1 (get container logs, return last line only)

Kubectl describe pod/my-apache-xx-xxx (use selector, based on the label containers shared. Get a bunch of details about an object, including events, labels are given when cubectl run xxx, similar to inspect in docker/swarm, in the end of output, can see all the event details)

Kubectl get pods –w (like watch command)

Kubectl delete pod/my-apache-xxx-xxx (delete a pod)

Kuberctl delete deploy/xxxx

kubectl Cheat Sheet

<https://kubernetes.io/docs/reference/kubectl/cheatsheet/>

kubectl Commands

[https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands - -strong-getting-started-strong-](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#-strong-getting-started-strong-)

19/12/19

**Linux(Bash)**

Passing Variables to Scripts at the Command Line

In the script:

Using **$1 $2** …to refer to the variables that passed at command line

Exp:

**$ abc.sh “Liang” “Chen”** (command line)

**echo “First name is $1 second name is $2”** (in script) -> return First name is Liang second name is Chen

If/Then/Else

To check if a file exist: -a

Exp:

If [ -a “mytxt.txt” ]; then

**If [ condition ];**

**Then**

**#commands to be run if true**

**elif [ condition ];**

**then**

**#commands to be run if true**

**else**

**#commands to be run if true**

**fi**

For loop

**For arg in [List]**

**Do**

**#commands**

**Done**

**For ((expression1; expression2; expression3)) do**

**#commands**

**Done**

Exp:

For ((i=1;i<=100;i++)) do

Echo $i

Done

While loop

**While [ condition ] do**

**#commands**

**Done**

Case statement (same as switch in C#)

**Case “$VAR” in**

**Pattern\_1 )**

**#commands**

**;; (;;means break)**

**Pattern\_2 )**

**#commands**

**;;**

**\* )**

**#default statement**

**;;**

**esac**

**Docker-Kubernetes**

**Kubectl expose** (creates a service for existing pods)

A service is a stable address for pod(s)

Use service to connect pod(s)

CoreDNS allows us to resolve services by name

Service types:

ClusterIP, NodePort, LoadBalancer, ExternName

ClusterIP (Default):

Single, internal virtual IP allocated

Only reachable from within cluster (nodes and pods)

Pods can reach service on apps port number

NodePort:

High port allocated on each node

Port is open on every node’s IP

Anyone can connect (if they can reach node)

ClusterIP and NodePort are always available in K8S

LoadBalancer:

Mostly used in Cloud

Controls a LB endpoint external to the cluster

Only available when infrastructure provider gives you a LB (AWS ELB…)

Creates NodePorts + ClusterIP services, tells LB to send to NodePort, only for traffic coming in from external sources

ExternalName:

Adds CNAME DNS record to CoreDNS only

Not used for Pods, but for giving pods a DNS name to use for something outside K8S

Create ClusterIP & NodePort Service

**kubectl create deployment httpenv --image=bretfisher/httpenv**

**kubectl expose deployment/httpenv --port 8888** (open port 8888 for deployment httpenv)

**kubectl expose deployment --port 8888 --name httpenv-np --type NodePort httpenv** (NodePort, if no –type -> default is ClusterIP)

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

service/httpenv ClusterIP 10.96.30.81 <none> 8888/TCP 4m44s

service/httpenv-np NodePort 10.102.76.156 <none> 8888:31351/TCP 27s

httpenv-np’s port: 8888:31351 (8888 is inside the cluster, 31351 is on node expose to outside, range between 30000-32767)

Create LoadBalancer

**kubectl expose deployment --port 8888 --name httpenv-lb --type LoadBalancer httpenv**

DNS in K8S

**kubectl get namespaces**

NAME STATUS AGE

default Active 22h

docker Active 22h

kube-node-lease Active 22h

kube-public Active 22h

kube-system Active 22h

Generators

Commands use helper templates called “generators”

**kubectl create deployment test --image nginx --dry-run -o yaml (**output the result of the command to a yaml file without making any actual change.)

apiVersion: apps/v1

kind: Deployment

metadata:

creationTimestamp: null

labels:

app: test

name: test

spec:

replicas: 1

selector:

matchLabels:

app: test

strategy: {}

template:

metadata:

creationTimestamp: null

labels:

app: test

spec:

containers:

- image: nginx

name: nginx

resources: {}

status: {}

what’s the difference between K8S client version and server version?

I have upgraded my client version to 1.17.0, but my server version is still 1.14.8.

Follow link below, only upgraded client version. (already posted my question on Udemy, waiting for answer) <https://kubernetes.io/docs/tasks/tools/install-kubectl/#install-kubectl-on-macos>

Bret replied:

Client: the cli binary you are running commands on

Server: the API service running on the node

How you control their versions depends on how you installed them and where they are running.

<https://www.udemy.com/course/docker-mastery/learn/lecture/15126234#questions>

Imperative VS Declarative

Imperative: Focus on how a program operates (do things step by step)

Declarative: Focus on what a program should accomplish (system take care of the process)

Imperative command in K8S:

Kubectl run, kuberctl create deployment, kubectl update

Declarative:

Kubectl apply –f my-resources.yaml

Automation, need to understand YAML keys and values

Three management approaches:

1. Imperative commands: run, expose, scale, edit, create deployment…

Best for dev/learning/personal projects

Easy to learn, hardest to manager over time

1. Imperative objects (middle ground): create –f files.yml, replace –f file.yml, delete…

Good for prod of small environments, single file per command

Store your changes in git-based yaml files

Hard to automation

1. Declarative objects: apply –f file.yml, dir\, diff…

Best for prod, easier to automate

Harder to understand and predict changes

Watched a video talks about Cloud Computing in the Year 2020

<https://www.youtube.com/watch?v=1pBuwKwaHp0>

In the video, it states that serverless will play a big role in 2020, I’m agreed with that, therefore, need to start doing AWS lambda studying.

Holiday Docker and Kubernetes Q&A: DevOps and Docker Show (Ep 66) By Bret Fisher Docker and DevOps

<https://www.youtube.com/watch?v=Oj6wDZKbLoU>

20/12/19

**Linux(Bash)**

Reading Files

**echo "Entre a file name to read:"**

**read FILE**

**while read -r SUPERHERO; do**

**echo "Superhero Name: $SUPERHERO"**

**done < "$FILE"**

(read a file, then output its content with prefixed text, read –r read from a file)

File Descriptors and Handles

**echo "Entre a filename to read:"**

**read FILE**

**exec 5<>$FILE**

**while read -r SUPERHERO; do**

**echo "Superhero name: $SUPERHER"**

**done <&5**

**echo "THis file Read On: `date`" >&5**

**exec 5>&-**

(use file descriptor to represents a file, has to be equal or greater to 3, < read, > write, in this case 5 represents the file, in the end, exec 5>&- to close the file)

IFS and Delimiting

**echo "Enter filename to parse: "**

**read FILE**

**echo "Enter the Delimiter: "**

**read DELIM**

**IFS="$DELIM"**

**while read -r CPU MEMORY DISK; do**

**echo "CPU: $CPU"**

**echo "Memory: $MEMORY"**

**echo "Disk: $DISK"**

**done <"$FILE"**

(IFS represents delimiter, by default its “ “ space)

Traps and Signals

**trap 'echo " - Please press Q to exit.."' SIGINT SIGTERM SIGTSTP**

**clear**

**while [ "$CHOICE" != "Q" ] && [ "$CHOICE" != "q" ]; do**

**echo "MAIN MENU"**

**echo "1) ONE"**

**echo "2) TWO"**

**echo "3) THREE"**

**echo "Q) Quite"**

**echo CHOICE**

**clear**

**done**

**Docker-Kubernetes**

**Kubectl apply**

**Kubectl apply -f filename.yaml** (create/update resources in a file)

**Kubectl apply -f myyaml/** (create/update a whole directory of yaml)

**Kubectl apply -f https://xxx.xxx/xxx.yml** (create/update from a URL)

**Curl -L** [**https://xxx.xxx/xxx**](https://xxx.xxx/xxx)(look at the yaml file first)

Configuration YAML

YAML or JSON?

YAML is more human readable, in the background, the system converts YAML to JSON.

Each file contains one or more manifests, each manifest describes an API object (deployment, job, secret), each manifest needs four parts (four root key: values in the file)

apiVersion: get the API versions the cluster supports

**kubectl api-version**

kind: get a list of resources the cluster supports

**kubectl api-resources**

metadata: only name is required

spec: where all the action is at

exp YAML file:

apiVersion: v1

kind: Pod

metadata:

name: nginx

spec:

containers:

- name: nginx

image: nginx:1.17.3

ports:

- containerPort: 80

**kubectl explain services –recursive** (display all the keys each kind supports)

**kubectl explain services.spec** (ServiceSpec describes the attributes that a user creates on a service)

**kubectl explain services.spec** (type determines how the Service is exposed. Defaults to ClusterIP. Valid options are ExternalName, ClusterIP, NodePort, and LoadBalancer)

kubectl apply –f app.yml

vim app.yml (change app.yml)

kubectl diff –f app.yml (use diff to see the changes)

Labels:

Labels are key/value pairs that are attached to objects, such as pods. Labels are intended to be used to specify identifying attributes of objects that are meaningful and relevant to users, but do not directly imply semantics to the core system. Labels can be used to organize and to select subsets of objects. Labels can be attached to objects at creation time and subsequently added and modified at any time. Each object can have a set of key/value labels defined. Each Key must be unique for a given object.

<https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/>

Labels goes under metadata in YAML

Simple list of key:value for identifying your resource later by selecting, grouping, or filtering for it

Common examples include tier:frontend, app:api, env:prod, customer:xxx.co

Filter exp:

Kubectl get pods –l app=nginx (only get pods with nginx label)

Kubectl applu –f xx.yaml –l app=nginx ( only apply certain parts of the yaml file)

Label Selectors

Unlike names and UIDs, labels do not provide uniqueness. In general, we expect many objects to carry the same label(s).

Via a label selector, the client/user can identify a set of objects. The label selector is the core grouping primitive in Kubernetes.

The API currently supports two types of selectors: equality-based and set-based. A label selector can be made of multiple requirements which are comma-separated. In the case of multiple requirements, all must be satisfied so the comma separator acts as a logical AND (&&) operator.

<https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/>

What I need to do during Christmas holiday? (23 – 27/12/2019)

1. Markdown (need to start changing existing notes format to with Markdown)
2. YAML (need to start preparing write YAML)
3. Python (syntax, need to start LeetCode, record problems and solutions)
4. Review my notes daily

28/12/19

**Docker-Kubernetes**

Storage

StatefulSets is a new resource type, making Pods more sticky

Volumes

2 types volumes:

Volumes:

Tied to lifecycle to a Pod

All containers in a single Pod can share them

PersistentVolumes:

Create at the cluster level, outlives a Pod

Separates storage config from Pod using it

Multiple Pods can share them

CSI: Container Storage Interface

CSI plugins are the new way to connect to storage

Ingress

None of Service types work at OSI Layer 7(HTTP)

How do we route outside connections based on hostname or URL:

Ingress Controllers(optional) do this with 3rd party proxies

Nginx, Traefik, HAProxy, F5, Envoy, Istio…

CRD’s and The Operator Pattern

Can add 3rd party Resources and Controllers, which extends K8S API and CLI

Operator Pattern: automate deployment and management of complex apps: Databases, monitoring tools, backups and custom ingresses…

Higher Abstractions for Apps

Deployment options

All kubectl commands just talk to K8S API

K8S has limited built-in templating, versioning, tracking and management of your apps

Helm: create templating YAML

K8S dashboard: a web GUI for cluster management

Some distributions have their own GUI (Rancher, Docker Ent, OpenShift)

Kubectl Namespaces and Context

Namespaces limit scope, aka “virtual clusters”

29/12/19

**Docker-Kubernetes**

**Kubectl run --restart=OnFailure**

Create a job (one time thing to do, such as batch job), don’t need restart if it has healthy exit

**Kubectl run --restart=Never**

Whatever happened to the pod (doesn’t create whole development) it won’t restart

**Kubectl run --schedule=…**

Create **cronjobs** (a job that will be executed at specific intervals)

**kubectl run --schedule="\*/3 \* \* \* \*" --restart=OnFailure --image=alpine sleep 10**

run alpine every 3 mins (\*/3 \* \* \* \* is same what crond in UNIX)

To create explicit things, use kubectl create

**Kubectl create deployment**

**Kubectl create job**

**Kubectl create cronjob**

**Kubectl run** will be used only to start one-shot pods

Kubeclt logs has limitation, it can only have 5 connections. (too many connections can be overwhelming to K8S API)

Use **Stern** for showing logs of multiple pods.

03/02/2020

**Docker-Kubernetes**

Headless services

A headless service is obtained by setting the clusterIP field to None. (--cluster-ip=None or by providing a custom YAML)

As a result, the service doesn’t have a virtual IP address, there is not load balancer either

CoreDNS will return the pods’IP addresses as multiple A records, this gives us an easy way to discover all the replicas for a deployment

**kubectl create -f whatever.yaml**

* creates resources if they don't exist
* if resources already exist, don't alter them (and display error message)

**kubectl apply -f whatever.yaml**

* creates resources if they don't exist
* if resources already exist, update them (to match the definition provided by the YAML file)
* stores the manifest as an annotation in the resource

04/02/2020

**Docker-Kubernetes**

DaemonSet

A DaemonSet ensures that all (or some) Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected. Deleting a DaemonSet will clean up the Pods it created.

(<https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/)>

Taints and Tolerations

Node affinity, described here, is a property of pods that attracts them to a set of nodes (either as a preference or a hard requirement). Taints are the opposite – they allow a node to repel a set of pods.

Taints and tolerations work together to ensure that pods are not scheduled onto inappropriate nodes. One or more taints are applied to a node; this marks that the node should not accept any pods that do not tolerate the taints. Tolerations are applied to pods, and allow (but do not require) the pods to schedule onto nodes with matching taints.

(<https://kubernetes.io/docs/concepts/configuration/taint-and-toleration/)>

The "mission" of a replica set is:

"Make sure that there is the right number of pods matching this spec!"

The "mission" of a daemon set is:

"Make sure that there is a pod matching this spec on each node!"

Add enabled=yes to all pods that have app=rng:

kubectl label pods -l app=rng enabled=yes

kubectl label ... foo= doesn't remove a label (it sets it to an empty string)

to remove label foo, use kubectl label ... foo-

to change an existing label, we would need to add –overwrite

the label(s) of a resource (in the metadata block in the beginning)

the selector of a resource (in the spec block)

the label(s) of the resource(s) created by the first resource (in the template block)

kubectl edit

This is equivalent to first get the resource, edit it in text editor, and then apply the resource with the updated version

kubectl edit deployment/my-nginx

kubectl get deployment my-nginx -o yaml > /tmp/nginx.yaml

vi /tmp/nginx.yaml

# do some edit, and then save the file

kubectl apply -f /tmp/nginx.yaml

deployment.apps/my-nginx configured

rm /tmp/nginx.yaml

Write YAML file

Find the resource kind you want to create (api-resources)

Find the latest apiVersion your cluster supports for kind (api-versions)

Give it a name in metadata (minimum)

Dive into the spec of that kind

kubectl explain <kind>.spec

kubectl explain <kind> --recursive

05/02/2020

**Docker-Kubernetes**

Rolling updates

Two parameters determine the pace of the rollout: **maxUnavailable** and **maxSurge**

They can be specified in absolute number of pods, or percentage of the replicas count

At any given time ...

there will always be at least replicas - maxUnavailable pods available

there will never be more than replicas + maxSurge pods in total

there will therefore be up to maxUnavailable + maxSurge pods being updated

Rolling updates:

Kubectl edit deploy xxx

Or

Kubectl set image deploy xxx

**kubectl describe deployments xxx**

exp:

Replicas: 10 desired | 5 updated | 13 total | 8 available | 5 unavailable

to check rollout replicas

**kubectl rollout undo deploy xxxx**

to undo rolling update once

**kubectl rollout status deploy xxxx**

check rolling update status

Rollout history

**kubectl rollout history**

Roll back to the "known good" deployment version:

**kubectl rollout undo deployment worker --to-revision=1**

Healthchecks

Healthchecks are probes that apply to containers (not to pods)

Each container can have three (optional) probes:

**liveness** = is this container dead or alive? (most important probe)

**readiness** = is this container ready to serve traffic? (only needed if a service)

**startup** = is this container still starting up? (alpha in 1.16)

Different probe handlers are available (HTTP, TCP, program execution)

Liveness probe

Indicates if the container is dead or alive

A dead container cannot come back to life

If the liveness probe fails, the container is killed

(to make really sure that it's really dead; no zombies or undeads!)

What happens next depends on the pod's restartPolicy:

Never: the container is not restarted

OnFailure or Always: the container is restarted

When to use a liveness probe

To indicate failures that can't be recovered

deadlocks (causing all requests to time out)

internal corruption (causing all requests to error)

Anything where our incident response would be "just restart/reboot it"

Do not use liveness probes for problems that can't be fixed by a restart

Otherwise we just restart our pods for no reason, creating useless load

Readiness probe

Indicates if the container is ready to serve traffic

If a container becomes "unready" it might be ready again soon

If the readiness probe fails:

the container is not killed

if the pod is a member of a service, it is temporarily removed

it is re-added as soon as the readiness probe passes again

When to use a readiness probe

To indicate failure due to an external cause

database is down or unreachable

mandatory auth or other backend service unavailable

To indicate temporary failure or unavailability

application can only service N parallel connections

runtime is busy doing garbage collection or initial data load

Startup probe

Kubernetes 1.16 introduces a third type of probe: startupProbe

(it is in alpha in Kubernetes 1.16)

It can be used to indicate "container not ready yet"

process is still starting

loading external data, priming caches

Before Kubernetes 1.16, we had to use the initialDelaySeconds parameter

(available for both liveness and readiness probes)

initialDelaySeconds is a rigid delay (always wait X before running probes)

startupProbe works better when a container start time can vary a lot

Timing and thresholds

Probes are executed at intervals of periodSeconds (default: 10)

The timeout for a probe is set with timeoutSeconds (default: 1)

If a probe takes longer than that, it is considered as a FAIL

A probe is considered successful after successThreshold successes (default: 1)

A probe is considered failing after failureThreshold failures (default: 3)

A probe can have an initialDelaySeconds parameter (default: 0)

Questions to ask before adding healthchecks

Do we want liveness, readiness, both?

(sometimes, we can use the same check, but with different failure thresholds)

Do we have existing HTTP endpoints that we can use?

Do we need to add new endpoints, or perhaps use something else?

Are our healthchecks likely to use resources and/or slow down the app?

Do they depend on additional services?

06/02/2020

**Terraform**

Common commands:  
apply: Builds or changes infrastructure  
console: Interactive console for Terraform interpolations  
destroy: Destroys Terraform-managed infrastructure  
fmt: Rewrites configuration files to canonical format  
get: Downloads and installs modules for the configuration  
graph: Creates a visual graph of Terraform resources  
import: Imports existing infrastructure into Terraform  
init: Initializes a new or existing Terraform configuration  
output: Reads an output from a state file  
plan: Generates and shows an execution plan  
providers: Prints a tree of the providers used in the configuration  
push: Uploads this Terraform module to Terraform Enterprise to run  
refresh: Updates local state file against real resources  
show: Inspects Terraform state or plan  
taint: Manually marks a resource for recreation  
untaint: Manually unmarks a resource as tainted  
validate: Validates the Terraform files  
version: Prints the Terraform version  
workspace: Workspace management

07/02/2020

**Terraform**

Terraform commands:

The terraform **init** command is used to initialize a working directory containing Terraform configuration files. This is the first command that should be run after writing a new Terraform configuration or cloning an existing one from version control. It is safe to run this command multiple times.

**taint**: Manually mark a resource for recreation untaint: Manually unmark a resource as tainted

**terraform taint [NAME]**

**terraform untaint [NAME]**

The terraform taint command manually marks a Terraform-managed resource as tainted, forcing it to be destroyed and recreated on the next apply.

This command will not modify infrastructure, but does modify the state file in order to mark a resource as tainted. Once a resource is marked as tainted, the next plan will show that the resource will be destroyed and recreated and the next apply will implement this change.

**lookup** retrieves the value of a single element from a map, given its key. If the given key does not exist, a the given default value is returned instead.

**lookup(map, key, default)**

**exp:**

**# Download the latest Ghost Image**

**resource "docker\_image" "image\_id" {**

**name = lookup(var.image\_name, var.env)**

**}**

**# Start the Container**

**resource "docker\_container" "container\_id" {**

**name = lookup(var.container\_name, var.env)**

**image = "${docker\_image.image\_id.latest}"**

**ports {**

**internal = "${var.int\_port}"**

**external = "lookup(var.ext\_port, var.env)**

**}**

**}**

**Terraform** commands:

workspace: New, list, select and delete Terraform workspaces

Workspace subcommands:

**delete**: Delete a workspace list:

**List** Workspaces

**new**: Create a new workspace

**select**: Select a workspace

**show**: Show the name of the current workspace

Switching between different environments (dev, prod…)

Workspace default/dev/prod

**Provisioners**

Provisioners can be used to model specific actions on the local machine or on a remote machine in order to prepare servers or other infrastructure objects for service.

**local-exec Provisioner**

The local-exec provisioner invokes a local executable after a resource is created. This invokes a process on the machine running Terraform, not on the resource. See the remote-exec provisioner to run commands on the resource.

**Null Resource**

The null\_resource resource implements the standard resource lifecycle but takes no further action.rh

**Exp:**

**resource "null\_resource" "null\_id" {**

**provisioner "local-exec" {**

**command = "echo ${docker\_container.container\_id.name}:${docker\_container.container\_id.ip\_address} >> container.txt"**

**}**

**}**

10/02/2020

**Terraform**

Modules

A module is a container for multiple resources that are used together.

Every Terraform configuration has at least one module, known as its root module, which consists of the resources defined in the .tf files in the main working directory.

A module can call other modules, which lets you include the child module's resources into the configuration in a concise way. Modules can also be called multiple times, either within the same configuration or in separate configurations, allowing resource configurations to be packaged and re-used.

(<https://www.terraform.io/docs/configuration/modules.html)>

Modules are reusable Terraform configurations that can be called and configured by other configurations. Most modules manage a few closely related resources from a single provider. The Terraform Registry makes it easy to use any provider or module.

11/02/2020

**Terraform**

Data Sources

Data sources allow data to be fetched or computed for use elsewhere in Terraform configuration. Use of data sources allows a Terraform configuration to make use of information defined outside of Terraform, or defined by another separate Terraform configuration.

(<https://www.terraform.io/docs/configuration/data-sources.html>)

14/02/2020

**Ansible**

Ansible configuration file:

/etc/ansible/ansible.cfg

controls how Ansible behaviours

ansible-config list

list all current configs reading lib/constants.py and shows env and config file setting names

Ansible inventory file:

/etc/ansible/hosts

Ansible works against multiple managed nodes or “hosts” in your infrastructure at the same time, using a list or group of lists know as inventory. Once your inventory is defined, you use patterns to select the hosts or groups you want Ansible to run against.

<https://docs.ansible.com/ansible/latest/user_guide/intro_inventory.html#inventory-basics-formats-hosts-and-groups>

ad-hoc commands

An Ansible ad-hoc command uses the /usr/bin/ansible command-line tool to automate a single task on one or more managed nodes. Ad-hoc commands are quick and easy, but they are not re-usable. So why learn about ad-hoc commands first? Ad-hoc commands demonstrate the simplicity and power of Ansible. The concepts you learn here will port over directly to the playbook language. Before reading and executing these examples, please read How to build your inventory.

(<https://docs.ansible.com/ansible/latest/user_guide/intro_adhoc.html#why-use-ad-hoc-commands>)

$ ansible **[**pattern**]** -m **[**module**]** -a "[module options]"

exp:

**ansible -i inv remote -m ping**

return ping results from nodes under remote group defined in local inv file.

**ansible -i inv centos -b -m yum -a "name=elinks state=installed"**

use yum install elinks in nodes under centos group defined in local inv file

-b = sudo

**ansible -i inv centos -b -m yum -a "name=elinks state=absent"**

use yum uninstall elinks

Modules

Modules (also referred to as “task plugins” or “library plugins”) are discrete units of code that can be used from the command line or in a playbook task. Ansible executes each module, usually on the remote target node, and collects return values.

Module index

(<https://docs.ansible.com/ansible/latest/modules/modules_by_category.html)>

check modules in nodes:

**man ansible-doc**

check available ansible modules

**ansible-doc xxx(module name)**

check specific module

Ansible command module vs shell module vs script module:

In the most use cases both modules lead to the same goal. Here are the main differences between these modules.

* With the **Command module** the command will be executed without being proceeded through a shell. As a consequence some variables like $HOME are not available. And also stream operations like <, >, | and & will not work.
* **The Shell module** runs a command through a shell, by default /bin/sh. This can be changed with the option executable. Piping and redirection are here therefor available.
* The command module is more secure, because it will not be affected by the user’s environment.
* **The Script module** the given script will be processed through the shell environment on the remote node.

check details of managed nodes

**ansible -i inv remote -m setup | less**

**ansible -i inv remote -m setup –a “filter=xxx” (can use wildcard)**

**exp:**

**ansible -i inv remote -m setup –a “filter=\*dist\*”**

return distribution information of managed nodes in remote group

**ansible -i inv remote -m setup --tree facts**

output results to local file facts

17/02/2020

**Ansible**

File module:

**acl** – Set and retrieve file ACL information

**archive** – Creates a compressed archive of one or more files or trees

**assemble** – Assemble configuration files from fragments

**blockinfile** – Insert/update/remove a text block surrounded by marker lines

**copy** – Copy files to remote locations

**fetch** – Fetch files from remote nodes

**file** – Manage files and file properties

**find** – Return a list of files based on specific criteria

**ini\_file** – Tweak settings in INI files

**iso\_extract** – Extract files from an ISO image

**lineinfile** – Manage lines in text files

**patch** – Apply patch files using the GNU patch tool

**read\_csv** – Read a CSV file

**replace** – Replace all instances of a particular string in a file using a back-referenced regular expression

**stat** – Retrieve file or file system status

**synchronize** – A wrapper around rsync to make common tasks in your playbooks quick and easy

**tempfile** – Creates temporary files and directories

**template** – Template a file out to a remote server

**unarchive** – Unpacks an archive after (optionally) copying it from the local machine

**xattr** – Manage user defined extended attributes

**xml** – Manage bits and pieces of XML files or strings

**get\_url** – Downloads files from HTTP, HTTPS, or FTP to node

**user** – Manage user accounts

**group** – Add or remove groups

**package** – Generic OS package manager

**service** – Manage services (Controls services on remote hosts. Supported init systems include BSD init, OpenRC, SysV, Solaris SMF, systemd, upstart.)

18/02/2020

**Ansible-Playbooks**

Playbooks

Playbooks are Ansible’s configuration, deployment, and orchestration language. They can describe a policy you want your remote systems to enforce, or a set of steps in a general IT process.

If Ansible modules are the tools in your workshop, playbooks are your instruction manuals, and your inventory of hosts are your raw material.

At a basic level, playbooks can be used to manage configurations of and deployments to remote machines. At a more advanced level, they can sequence multi-tier rollouts involving rolling updates, and can delegate actions to other hosts, interacting with monitoring servers and load balancers along the way.

While there’s a lot of information here, there’s no need to learn everything at once. You can start small and pick up more features over time as you need them.

Playbooks are designed to be human-readable and are developed in a basic text language. There are multiple ways to organize playbooks and the files they include, and we’ll offer up some suggestions on that and making the most out of Ansible.

Exp:

**---**

**-** hosts**:** webservers

remote\_user**:** root

tasks**:**

**-** name**:** test connection

ping**:**

remote\_user**:** yourname

Executing A Playbook

ansible-playbook playbook.yml -f 10

-K ask for become password

-k ask for the connect(SSH) password

-C run in check mode(dry run)

Tasks list

Each play contains a list of tasks. Tasks are executed in order, one at a time, against all machines matched by the host pattern, before moving on to the next task. It is important to understand that, within a play, all hosts are going to get the same task directives. It is the purpose of a play to map a selection of hosts to tasks.

tasks**:**

**-** name**:** make sure apache is running

service**:**

name**:** httpd

state**:** started

19/02/2020

**Ansible-Playbooks**

Registering variables

Another major use of variables is running a command and **registering** the result of that command as a variable. When you execute a task and save the return value in a variable for use in later tasks, you create a registered variable. There are more examples of this in the Conditionals chapter.

**stat** – Retrieve file or file system status

exp:

- hosts: web\_servers

tasks:

- shell: /usr/bin/foo

register: foo\_result

ignore\_errors: True

- shell: /usr/bin/bar

when: foo\_result.rc == 5

exp2:

--- # Ansible conditional example

- hosts: remote

vars:

target\_file: /home/ansible/hostname

tasks:

- name: Gather file information

**stat**:

path: "{{ target\_file }}"

**register**: hostname

- name: Rename hostname when found

command: mv "{{ target\_file }}" /home/ansible/net-info

when: hostname.stat.exists

**hostname** has the return value of stat: path: "{{ target\_file }}"

Loops

Exp1:

**-** name**:** add several users

user**:**

name**:** "{{ **item** }}"

state**:** present

groups**:** "wheel"

loop**:**

**-** testuser1

**-** testuser2

Can define the list in a variables file, or in the ‘vars’ section of your play, then refer to the name of the list in the task:

exp2:

loop2.yml

--- # loop example

- hosts: localhost

become: yes

vars\_files:

- vars.yml

tasks:

- name: Check services

service:

name: "{{ item }}"

state: started

loop: "{{ service\_list }}"

var.yml:

working\_dir: /home/ansible/working

service\_list:

- httpd

- nfs

- mariadb

share\_paths:

nfs: /mnt/nfs

cifs: /mnt/cifs

iscsi: /mnt/iscsi

Handlers:

Running Operations On Change

Modules should be idempotent and can relay when they have made a change on the remote system. Playbooks recognize this and have a basic event system that can be used to respond to change.

These ‘**notify’** actions are triggered at the end of each block of tasks in a play, and will only be triggered once even if notified by multiple different tasks.

Exp1:

**-** name**:** template configuration file

template**:**

src**:** template.j2

dest**:** /etc/foo.conf

notify**:**

**-** restart memcached

**-** restart apache

**Handlers** are lists of tasks, not really any different from regular tasks, that are referenced by a globally unique name, and are notified by notifiers. If nothing notifies a handler, it will not run. Regardless of how many tasks notify a handler, it will run only once, after all of the tasks complete in a particular play.

(listen should be same as notify)

Exp2:

tasks**:**

**-** name**:** restart everything

command**:** echo "this task will restart the web services"

notify**:** "restartwebservices"

handlers**:**

**-** name**:** restart memcached

service**:**

name**:** memcached

state**:** restarted

listen**:** "restartwebservices"

**-** name**:** restart apache

service**:**

name**:** apache

state**:** restarted

listen**:** "restartwebservices"

exp3:

--- # handler example

- hosts: scoldham2

become: yes

vars:

httpd\_log\_level: error

tasks:

- name: install httpd

yum:

name: httpd

state: latest

- name: update configuration

template:

src: /home/ansible/httpd.conf.j2

dest: /etc/httpd/conf/httpd.conf

notify: httpd service

handlers:

- name: httpd service

service:

name: httpd

state: restarted

listen: httpd service

20/02/2020

**Ansible-Playbooks**

Ansible Vault

**Ansible Vault** is a feature of ansible that allows you to keep sensitive data such as passwords or keys in encrypted files, rather than as plaintext in playbooks or roles. These vault files can then be distributed or placed in source control.

A **vault ID** is an identifier for one or more vault secrets; Ansible supports multiple vault passwords.

Vault IDs provide labels to distinguish between individual vault passwords.

To use vault IDs, you must provide an ID label of your choosing and a source to obtain its password (either prompt or a file path):

Blocks

**Blocks** allow for logical grouping of tasks and in play error handling. Most of what you can apply to a single task (with the exception of loops) can be applied at the block level, which also makes it much easier to set data or directives common to the tasks. This does not mean the directive affects the block itself, but is inherited by the tasks enclosed by a block. i.e. a when will be applied to the tasks, not the block itself.

**Blocks error handling**

Blocks also introduce the ability to handle errors in a way similar to exceptions in most programming languages. Blocks only deal with ‘failed’ status of a task. A bad task definition or an unreachable host are not ‘rescuable’ errors.

Exp:

--- # Error handling example #2

- hosts: localhost

become: yes

vars:

target\_service: openvpn-client

tasks:

- name: Install Software

block:

- service:

name: "{{ target\_service }}"

state: started

register: service\_status

rescue:

- debug:

var: service\_status

always:

- debug:

msg: "Tried to ensure service was running"

**block** – Group tasks into a “block”

**rescue** - A special block that is executed when the preceding block fails

**always** – A special block that is always executed after the preceding

Asynchronous Actions and Polling

Time-limited background operations

You can run long-running operations in the background and check their status later. For example, to execute long\_running\_operation asynchronously in the background, with a timeout of 3600 seconds (**-B**), and without polling (**-P**):

Exp:

$ ansible all -B 3600 -P 0 -a "/usr/bin/long\_running\_operation --do-stuff"

Exp2:

--- # Async Task Example

- hosts: localhost

tasks:

- name: Run sleep.sh

command: /home/ansible/sleep.sh

async: 60

poll: 10

ansible will wait for up to 60 sec, poll every 10 sec.

When **poll** is 0, Ansible will start the task and immediately move on to the next one without waiting for a result.

Delegation

If you want to perform a task on one host with reference to other hosts, use the ‘**delegate\_to**’ keyword on a task. This is ideal for placing nodes in a load balanced pool, or removing them. It is also very useful for controlling outage windows. Be aware that it does not make sense to delegate all tasks, debug, add\_host, include, etc always get executed on the controller. Using this with the ‘serial’ keyword to control the number of hosts executing at one time is also a good idea:

--- # Delegation Example

- hosts: centos

tasks:

- name: Run sleep.sh

command: /home/ansible/sleep.sh

async: 60

poll: 0

delegate\_to: scoldham2c

- name: Install mariadb

package:

name: mariadb

state: absent

become: yes

task Run sleep.sh will only run on scoldham2c host

local\_action

shorthand syntax for delegating **localhost**(127.0.0.1)

tasks**:**

**-** name**:** take out of load balancer pool

local\_action**:** command /usr/bin/take\_out\_of\_pool {{ **inventory\_hostname** }}

Parallelism

By default, Ansible will try to manage all of the machines referenced in a play in parallel. For a rolling update use case, you can define how many hosts Ansible should manage at a single time by using the **serial** keyword:

--- # Serial execution Example

- hosts: all

max\_fail\_percentage: 1

serial:

- 5

- "30%"

- "50%"

become: yes

tasks:

- name: Install elinks

package:

name: elinks

state: latest

**serial** keyword may be used to control forks(-F) in a playbook. It can be integer or percentage of the total hosts

**max\_fail\_percentage**: all a certain percentage to fail

Roles

Roles are ways of automatically loading certain vars\_files, tasks, and handlers based on a known file structure. Grouping content by roles also allows easy sharing of roles with other users.

Roles expect files to be in certain directory names. Roles must include at least one of these directories, however it is perfectly fine to exclude any which are not being used. When in use, each directory must contain a **main.yml** file, which contains the relevant content:

**tasks** - contains the main list of tasks to be executed by the role.

**handlers** - contains handlers, which may be used by this role or even anywhere outside this role.

**defaults** - default variables for the role (see Using Variables for more information).

**vars** - other variables for the role (see Using Variables for more information).

**files** - contains files which can be deployed via this role.

**templates** - contains templates which can be deployed via this role.

**meta** - defines some meta data for this role. See below for more details.

Dynamic vs. Static

Ansible has two modes of operation for reusable content: dynamic and static.

In Ansible 2.0, the concept of dynamic includes was introduced. Due to some limitations with making all includes dynamic in this way, the ability to force includes to be static was introduced in Ansible 2.1. Because the include task became overloaded to encompass both static and dynamic syntaxes, and because the default behavior of an include could change based on other options set on the Task, Ansible 2.4 introduces the concept of include vs. import.

If you use any include\* Task (include\_tasks, include\_role, etc.), it will be dynamic. If you use any import\* Task (import\_playbook, import\_tasks, etc.), it will be static.

How to Use Ansible Roles to Abstract your Infrastructure Environment

<https://www.digitalocean.com/community/tutorials/how-to-use-ansible-roles-to-abstract-your-infrastructure-environment>