ATTENZIONE

I doppi - consecutivi dei comandi per via di come Google Documenti gestisce i simboli sono mostrati come uno singolo, inoltre non è consigliato copiare direttamente i comandi da qui sulla bash

Docker

Per far partire un container tramite un'immagine (la cerca prima in locale e poi su Docker Hub), con un nome custom (univoco) e collegarlo alla rete del Docker host (-d per farlo partire in background [detached mode]):

docker run -d –port portaEsterna:portaInterna –name nomeContainer nomeImmagine

Oppure

[Non consigliato, perché sappiamo solo la porta interna]

docker run -d –port portaInterna – name nomeContainer nomeImmgina

Porta Esterna: porta del container [accedibile da localhost]

Porta Interna: porta del server [data dal container, non è da cambiare]

Esempio:

Se un Docker Engine (PC su cui è installato Docker) contiene un container che è un server HTTP (nginx) (in esecuzione sulla porta 80), e vogliamo che dalla rete locale (dove è collegato Docker Engine) si possa accedere al container devo:

- avere l'indirizzo del Docker Engine;
- avere la porta su dove gira nginx sul Docker Engine (8080)

Quindi:

docker -run -d –port 8080:80 –name miohttp nginx

Possiamo, dopo aver creato un container, creare un'immagine di esso in modo da poter duplicare velocemente lo stesso container [Utile all'esame per quando si chiede di creare tanti nodi]:

docker commit idContainer nomeImmagine

Successivamente per farlo partire si fanno gli stessi comandi sopracitati.

Network

[MACVLAN non sarà nell'esame]

Ogni container viene linkato ad una rete [docker network ls] che di default è bridge; c'è la possibilità di creare reti custom partendo da modelli base:

docker network create –driven bridge alpine-net

Per visionare nel dettaglio ogni rete (bridge in questo caso):

docker network inspect bridge

Per esempio è possibile vedere i container al suo interno.

Per connettere un container ad una rete (colleghiamo alpine ad bridge):

docker network connect bridge alpine

Un container può collegarsi a più reti.

Per far partire un container in una rete specifica (host in questo caso), it serve per interfacciare il nostro container con il Docker Engine, con una immagine specifica che permette di eseguire un comando appena fatto partire:

docker run -it --network=host alpine:latest /bin/ash

Posso anche unire i flag, es: -dit (d per background, it per interfacciare)

Questa immagine se non gli si dà un comando all'avvio viene creata e distrutta subito, perché non ha nessun compito.

Dopo la creazione se vogliamo entrare nel container interattivo (grazie ad -it) posso usare:

docker attach alpine

Ricordarsi di usare CTRL + P + Q per uscire.

Se si usa exit o CTRL+D l'interprete dei comandi si spegne e così il container.

Mantenere stato

Bind mount

Collega una cartella/file della nostra macchina (Docker Engine) nel container.

Per fare mount di una cartella dalla nostra macchina sul container:

docker run -dit -v /percorso/macchina:/percorso/container alpine:latest /bin/ash

Esempio:

docker run -dit -v C:\Users\matte\Desktop\prova:/tmp alpine:latest /bin/ash

RIcordarsi di mettere " " se il percorso ha spazi

Ora da dentro il container in /tmp sarà possibile accedere alla cartella \prova e ai suoi file.

Volume

Salva sempre le cartelle/file su un'area della nostra macchina ma è gestito completamente da Docker ed hanno un ciclo di vita al di fuori e non dipendente dal container.

Per usare un volume:

docker run -dit -v nomeVolume:/percorso/container alpine:latest /bin/ash

Esempio:

docker run -dit -v volumeEsempio:/tmp alpine:latest /bin/ash

Sostanzialmente se non si mette un path specifico crea un volume con quel nome, possiamo vedere i volumi creati con:

docker volume ls

e per vederli nel dettaglio:

docker volume inspect nomeVolume

Per eliminare i container che non usano volumi facciamo:

docker volume prune

Jenkins

Automation server per creare delle pipeline in fase di sviluppo, in modo da automatizzare processi che vengono fatti in maniera ripetuta.

Per scaricare l'immagine di jenkins e la chiamiamo jenkinsMaster:

docker run --name jenkinsMaster -p 8080:8080 -p 50000:50000 -d -v jenkins_home:/var/jenkins_home jenkins/jenkins

Successivamente possiamo ottenere la password per accedere all'account admin tramite [da fare dopo averli fatto partire]:

docker logs nomeContainer

Pipeline

La sintassi per le pipeline è:

```
pipeline {
    agent any
    stages {
        stage('Stage 1') {
            steps {
```

```
                echo 'Hello world!'
            }
        }
    }
}
```

Dove any, indica in quali nodi eseguire la pipeline, in questo caso tutti.

Git

Per avviare uno script residente su GitHub possiamo seguire i seguenti passaggi:

1. Andiamo nella pagina di creazione;
2. Cliccare: Pipeline Syntax, collocato sotto la sezione di inserimento della pipeline.
3. In Sample Step mettere git: Git.
4. Inserire l'URL, il nome del ramo e le credenziali.
5. Generare la pipeline e copiarla.
6. Creare uno script con diversi stage, uno con il comando copiato e uno per avviare il file.

Esempio:

```
pipeline {
    agent any
    stages {
        stage('inizio') {
            steps {
                echo 'pipeline avviata correttamente'
            }
        }
        stage('fetch') {
            steps {
                git branch: 'main', credentialsId: 'c6d7c2d5-8649-4650-981c-c6b80e94d27e', url: 'https://github.com/scastagnoli/PythonDevOps.git'
            }
        }
        stage('esecuzione') {
            steps {
```

```
                sh 'python3 liste.py'
            }
        }
    }
}
```

Triggered pipeline

Esistono due modalità:

- cron: come il daemon dentro Linux permette di eseguire la pipeline ad intervalli regolari;
- pollSCM: la pipeline controlla dal sorgente SCM (git o altro) se ci sono modifiche, in caso viene eseguita.

La pipeline deve essere eseguita manualmente una volta prima che il trigger venga preso in considerazione !!!

Struttura generale:

```
pipeline {
    agent any
    triggers {
            cron('H * */4 * * 1-5')
    }
    stages {
        stage('Example') {
            steps {
                echo 'Hello World'
            }
        }
    }
}
```

L'H indica un hash della pipeline corrente, che rende leggermente casuale l'esecuzione, così anche se ci sono tante pipeline che vengono eseguite con gli stessi intervalli evita conflitti.

Esempi di trigger:

Funzionamento	Comando
ogni 15 minuti (ad esempio a :07, :22, :37, :52)	triggers{ cron('H/15 * * * *') }
ogni 10 minuti nella prima mezz'ora di ogni ora (ad esempio a :04, :14, :24)	triggers{ cron('H(0-29)/10 * * * *') }
ogni due ore a :45 dale 9AM all 15:59 di ogni giorno da lunedì a venerdì	triggers{ cron('45 9-16/2 * * 1-5') }
una volta ogni due ore fra le 9 AM e le 5 PM di ogni giorno (esempio 10:38 AM, 12:38 PM, 2:38 PM, 4:38 PM)	triggers{ cron('H H(9-16)/2 * * 1-5') }
una volta al giorno l'1 e il 15 di ogni mese eccetto dicembre	triggers{ cron('H H 1,15 1-11 *') }

MINUTE	HOUR	DOM	MONTH	DOW
Minutes within the hour (0–59)	The hour of the day (0–23)	The day of the month (1–31)	The month (1–12)	The day of the week (0–7) where 0 and 7 are Sunday.

Si può controllare se la combinazione è giusta qui: https://crontab.guru/

Master-Slave

Dopo aver creato un nuovo nodo in Permanent Node (Gestisci Jenkins -> Nodes -> New Node -> nome nodo e mode) e aver impostato l'ind. IP per URL di Jenkins (dashboard->gestisci Jenkins->System->URL di Jenkins) possiamo creare un container che ospiterà il nostro slave e scaricare al suo interno jdk (per compilare java), curl e git:

- docker run -dit --name nomeSlave debian
- docker exec -it nomeSlave /bin/bash
- apt update
- apt install default-jdk curl git

Questo container funge da nodo slave.

Per connettersi a Jenkins eseguiamo i comandi che prendiamo dalla pagina del nodo direttamente da Jenkins.

Se tutto va come previsto nella bash dovrebbe uscire:

INFO: Connected

e come prova del nove se andiamo sulla pagina dei nodi in Jenkins notiamo che dice connesso e l'icona non è barrata.

In caso il nodo debba essere connesso nuovamente rifare i comandi dati da Jenkins (in particolare quello java con l'indirizzo ip).

Ora possiamo eseguire pipeline sui diversi nodi slave, mettendo negli agent i nomi di Jenkins e NON quello del container.

Ansible

Permette di centralizzare e automatizzare la gestione e i processi su dei nodi e il controllo del loro stato.

Gli elementi fondamentali di questa tecnologia sono:

- Control node: nodo dove vengono eseguiti i comandi per controllare gli altri nodi.
- Inventory: elenco dei nodi controllati, sta nel nodo di controllo.
- Managed node: nodi controllati dal Control node

Da ricordare: il Control Node NON è un server, anzi è lui che si collegherà ai vari nodi !!!

Configurazione nodi/ssh

Per preparare i nodi:

- docker run -dit –name nomeNodo debian
- apt update [Dentro al container, per entrare usa docker exec -it nome bash]
- apt install python3 openssh-server nano ansible

Solo sul nodo di controllo (Control node)

Successivamente vanno generate le key pair (dentro al Control Node), la chiave privata rimarrà nel nodo di controllo e i managed node avranno quella pubblica:

- [Control Node] ssh-keygen: successivamente verrà chiesto ->
- nome: /root/.ssh/nomeFile;
- passphrase (da ricordare): password.

```
root@7260142c3b9f:/# ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/root/.ssh/id_rsa): /root/.ssh/key
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /root/.ssh/key
Your public key has been saved in /root/.ssh/key.pub
The key fingerprint is:
SHA256:RvbAv2CnBU2OAkBa6Cnl4r/KBZWlTQb2siAkmAvqq6o root@7260142c3b9f
The key's randomart image is:
+---[RSA 3072]----+
|o=++o+    .      |
|Boo O. . =       |
|*=.= o. B o      |
|*o+ o  + =       |
|+o .    S =      |
| o.    o = .     |
|  o.    . .      |
|....             |
|Eo...            |
+----[SHA256]-----+
```

Per salvare la passphrase per evitare che venga chiesta ad ogni connessione [Facoltativo] [Control Node]:

- exec ssh-agent $SHELL

- ssh-add /root/.ssh/nomeFile
- [Comando di verifica] ssh-add -l

Ora bisogna distribuire la chiave pubblica ai Managed node, per farlo dobbiamo modificare il file di configurazione di ssh :

- [Managed Node] nano /etc/ssh/sshd_config: e modifichiamo le seguenti voci (cercale con CTRL + W):

- PermitRootLogin yes
- PubkeyAuthentication yes
- AuthorizedKeysFile .ssh/nomeFile.pub [ricordarsi di prendere la chiave dal control node (cat root/.ssh/nomeFile.pub), copiarla, e inserirla nel file in questa posizione]
- PasswordAuthentication no

Per poi restartare il server ssh [Managed Node]:

- /etc/init.d/ssh restart
- /etc/init.d/ssh status

Per verificare l'accessibilità dei nodi va fatto:

- [Control Node] ssh -i /root/.ssh/nomeFile root@host

Si prende facendo docker inspect bridge nomeContainer sul Docker Engine (IP Address)

NOTA BENE: se si decide di usare docker commit, per duplicare i nodi velocemente, usare il comando:

ssh-keygen -q -N " " -t ed25519 -f /etc/ssh/ssh_host_ed25519_key

in modo da generare un nuovo fingerprint per il server ssh (ricordarsi di fare il restart).

Altrimenti ssh non riuscirà a connettersi al host duplicato perché è settato per evitare un attacco di tipo Man in The Middle.

Funzionamento

Ora che abbiamo creato i container e configurato SSH possiamo passare ad usare Ansible.

Per iniziare bisogna creare l'inventory, quindi dentro il Control Node creiamo un file .ini dove inseriamo i gruppi con ciascun nodo, esempio:

[gruppo1]

172.17.0.4

172.17.0.5

[gruppo2]

172.17.0.7

[vm]

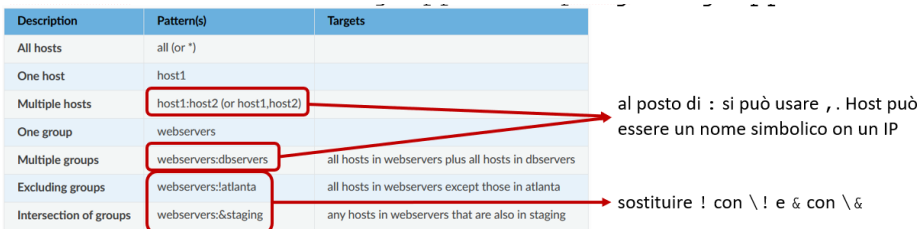
192.168.178.123

Possiamo vedere come sia possibile inserire anche VM oltre che host (non è necessario).

Successivamente è possibile fare diversi comandi di Ansible, esempio:

- ansible gruppo1 -m ping -i gruppi.ini => dove faccio ping a tutti i nodi del gruppo 1, prendendo le informazioni dal file gruppo.ini.

Esistono diversi pattern per far eseguire i comandi a diversi nodi:



In generale un comando Ansible è così strutturato:

ansible target -i inventory [-m module [-a “module options”]]

Dove:

- target: i nodi destinatari del comando;
- inventory: il file da dove prendere i target/gruppi/ecc;
- module: nome del modulo da usare (il comando);
- module options: dipende dal comando usato, se non uso -m posso inserire uno script da usare.

Ecco alcuni esempi:

- Modulo `apt` (https://docs.ansible.com/archive/ansible/2.3/apt_module.html):
  - `ansible gruppo1 -i gruppi.ini -m apt -a "update_cache=yes"`
  - `ansible gruppo1 -i gruppi.ini -m apt -a "name=tree"`
  - `ansible gruppo1 -i gruppi.ini -m apt -a "name=tree state=absent"`
- Moduli vari:
  - `ansible all -i gruppi.ini -m copy -a "src=./file.txt dest=~/myfile.txt"`
  - `ansible all -i gruppi.ini -m copy -a "src=~/myfile.txt remote_src=yes dest=./file.txt"`
  - `ansible all -i gruppi.ini -m file -a "dest=/var/www/file.txt mode=600 owner=prova group=prova"`
  - `ansible all -i gruppi.ini -m file -a "dest=/var/www/file.txt mode=600 owner=prova group=prova"`
  - `ansible gruppo1 -i gruppi.ini -m service -a "name=nginx state=restarted"`
  - `ansible gruppo2 -i gruppi.ini -m setup`
  - `ansible gruppo1 -i gruppi.ini -m setup -a "gather_subset=min"`
  - `ansible gruppo1 -i gruppi.ini -m setup -a "filter=*ipv*"`
- Esecuzione script
  - `ansible gruppo1 -i gruppi.ini -a "tail /var/log/apt/history.log"`
  - `ansible all -i inventory -a "df -h"`
  - `ansible webservers -i inventory -a "/sbin/reboot"`

Playbook

Servono per eseguire comandi Ansible in modo semplice e flessibile.

I due formati sono:

- .ini: semplici ma meno flessibili;
- .yaml: usati praticamente sempre quando la complessità aumenta.

Esempio di sintassi, ricordarsi che è necessario rispettare gli spazi:

```yaml
---
- name: installazione e avvio di Nginx
  hosts: vm
  tasks:
    - name: installazione ultima versione disponibile
      apt:
        name: nginx
        state: latest
    - name: avvio Nginx
      service:
        name: nginx
        state: restarted
        enabled: yes
```

Si possono avviare con il comando:

ansible-playbook -i inventory.ini nomeFile.yaml

Creare un VM con docker: https://secgroup.dais.unive.it/teaching/vm-with-docker/

Kubernetes

Sistema di orchestrazione di container. Composto da Nodi (macchina che esegue attività), cluster (nodi logicamente correlati), pod (uno o più container distribuiti su un singolo nodo, con stesso IP, IPC, nome host ecc)) e Control plane (controlla i nodi e i cluster).

Set up

Per vedere come creare e settare la macchina virtuale: https://unibo.cloud.panopto.eu/Panopto/Pages/Viewer.asp
78e0-4816-a740-b22600f9d90f

Comandi

Per creare un namespace:

kubectl create namespace nomeNamespace

Visualizzare i namespace;

kubectl get namespace

Creare dei deployment di un'immagine in un namespace specifico:

kubectl create deployment nomeDeploy –image=nomeImage -n=nomeNamespace

Visualizzare i deployment di un namespace:

kubectl get deployment -n nomeNamespace

Visualizzare tutti i deployment di tutti i namespace:

kubectl get deployment -A

Cambiare namespace corrente:

kubectl config set-context - -current - -namespace=nomeNamespace

per assicurarsi di essere nel giusto:

kubectl config get-contexts

Ottenere informazioni su un deployment [DA FARE DENTRO IL RISPETTIVO NAMESPACE]:

kubectl describe deployment nomeDeploy

Visualizzare il log di un deployment, utile in caso di errore [DA FARE DENTRO IL RISPETTIVO NAMESPACE]:

kubectl logs deployment/nomeDeploy

Cancellare tutti i deployment di un namespace:

kubectl delete deployment –all -n nomeNamespace

Visualizzare in quali nodi sono in esecuzione i pods:

kubectl get pods -o wide

MQTT