

Appunti Sistemi Cloud - Laboratorio

1. Vagrant, [1], [2]

Vagrant Set-Up Cheat Sheet	
by Mary F. Smith (boogie) via cheatography.com/86236/cs/20080/	
Vagrant Setup	Vagrant Share/Remote Access
Vagrant installs at c:\hashicorp\vagrant	vagrant share share your environment with anyone
Vagrantfile is located at c:\users\users name \vagrantfile	vagrant connect connect to a shared end point
Hyper-V & Virtual Box cannot run simultaneously	vagrant hosts all hostnames managed by plugin
Initial Configuration	vagrant hostmanager manages /etc/hosts within a multi machine environment
vagrant init [Initializes new vagrantfile] vagrant init hashicorp/precise64	vagrant ssh-config provides connect config for machine
Only 1 vagrantfile can exist	vagrant rdp connects to other machine via rdp
rm vagrantfile [deletes vagrantfile]	
rm vagrantfile hashicorp/precise64	
General Info & Day-to-Day Maintenance	Customization
vagrant box [all available vagrant boxes on this list endpoint]	vagrant.configure ("2") do config config.vm.box = "hashicorp/precise64" # guest is the VM; host is your computer end config.vm.network "forwarded_port", guest: 80, host: 8080 config.vm.provision :shell, path: "my_bash_script.sh" # path is relative to your Vagrantfile end
vagrant status [status of current vagrant endpoint]	By default ./ on your computer is shared as /vagrant on the vm; allowing other to access your VM
vagrant box outdated [checks for box updates]	
vagrant global-status	
i.e. c:\users\msmith097>	
id name provider state directory	
b2bcba9 default hyperv running	
C:/hashicorp/Vagrant	
a226748 default hyperv running	
C:/Users/msmith097	
Vagrant Use Commands [Run in Project Directory]	Create Base Boxes
vagrant init	vagrant provision
vagrant init hashicorp/precise64	vagrant push deploys code to a configured destination
vagrant up	vagrant package creates a box from a working environment
vagrant ssh	pe-build command related to a pe installation
vagrant suspend	
vagrant resume	
vagrant halt	
vagrant reload	
vagrant destroy	
vagrant --version	
	Website Resources
	https://learn.hashicorp.com/tutorials/vagrant/getting-started-boxes
	https://app.vagrantup.com/boxes/search
	https://www.vagrantbox.es/
	Advanced Vagrant Box Mgmt
	vagrant global-status status of all vagrant endpoints
	vagrant global-status prune status of all boxes; dropping invalid entries
	vagrant box remove deletes a copied end point
	vagrant provision forces end point to re-provision
	vagrant reload --provision restart end point forcing provisioning

Advanced Vagrant Box Mgmt (cont)

```
vagrant      | increases verbosity during debug
provision --
debug

vagrant up --provision | tee provision.log
enable VBGuest auto update

vagrant      PS C:\Users\msmith097> vagrant global-status
global-status
[displays
status for
all endpoints]

id name provider state directory -----
----- b2bcba9
default hyperv running C:/hashicorp/Vagrant
a226748 default hyperv running C:/Users/msmith097
```

Customize the Vagrantfile

```
Configure Base Boxes
Vagrant.configure("2") do |config|
  config.vm.box = "hashicorp/precise64" end
Set base box with version number
Vagrant.configure("2") do |config|
  config.vm.box = "hashicorp/precise64"
  config.vm.box_version = "1.1.0"
end
via url
Vagrant.configure("2") do |config|
  config.vm.box = "hashicorp/precise64"
  config.vm.box_url = "https://vagrantcloud.com/hashicorp/precise-64"
```

Vagrant Base Boxes [standard templates]

```
$ vagrant box add {title} or {url}
                               hashicorp/precise64
                               bento/centos-7
                               bento/ubuntu-16.04
                               ubuntu/trusty64
```

Vagrant Plugins

```
vagrant plugin | manages all plugins
vagrant plugin install vagrant-vbguest
vagrant plugin install vagrant-hostmanager
- manages /etc/hosts within a multi machine environment
vagrant plugin install vagrant-env
```

Windows Base Box [vagrant box add]

```
http://aka.ms/msedge.win10.vagrant
http://aka.ms/ie8.xp.vagrant
http://aka.ms/ie10.win7.vagrant
```

Vagrant Logs

```
vagrant up --provision | tee provision.log      forces provisioning and writes a log
vagrant _log=info vagrant up      use env var VAGRANT_LOG to
set verbosity
```

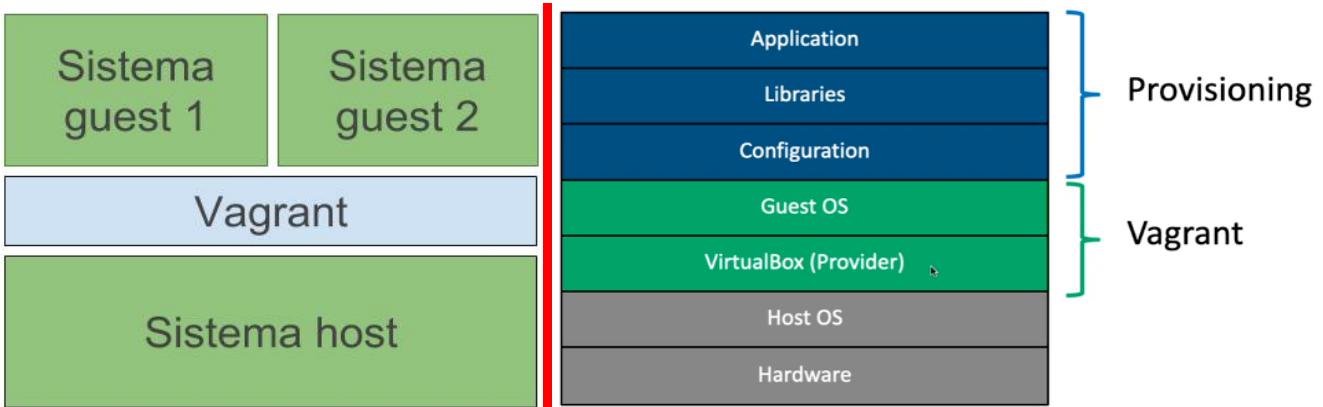
Vagrant è uno strumento che permette di costruire macchine virtuali e non solo. Tramite dei file di configurazione, mediante uno stile dichiarativo, permette la gestione e la configurazione rapida di macchine virtuali (anche molte). Inoltre è possibile utilizzare SSH per: salvare lo stato della macchina, salvare lo stato, spegnerla, ecc...

Vagrant è uno strumento multi-provider, ovvero funziona con molti software di tipo **hypervisor**, denominati **providers**, tra cui VirtualBox, VMware e KVM.

1.1 Virtualizzazione

La virtualizzazione è una tecnica che permette di astrarre l'hardware di un computer affinché questo sia disponibile come risorsa virtuale. Il sistema operativo iniziale, su cui è installato il software di virtualizzazione è detto **sistema host**, mentre il sistema operativo virtuale è detto **sistema guest**.

Poiché l'hardware è una risorsa virtuale, è possibile creare macchine guest che sfruttino solo una parte delle risorse messe a disposizione dall'host, ad esempio una macchina virtuale potrà essere configurata per funzionare con soli 2GB di RAM anche se il computer fisico che la tiene in vita ne ha a disposizione 8.



Le macchine virtuali possono essere “accese” e “spente” proprio come normali computer, liberando quindi risorse preziose per il sistema host, le “immagini” (ovvero i dischi rigidi) di una macchina virtuale possono essere salvate e scambiate tra i membri di un team in modo tale da avere un ambiente comune sul quale sviluppare o effettuare dei test.

1.2 Perché Vagrant?

Sebbene le macchine virtuali facciano già tanto, le sole non basterebbero a rendere il nostro viaggio liscio come la seta. Uno degli scenari più noiosi, infatti, è proprio la preparazione dell'ambiente di una macchina virtuale: dobbiamo avviare la macchina, entrarci, fare tutti i nostri setup, rigenerare un pacchetto di diversi gigabyte e condividerlo con il gruppo.

Se un giorno ci rendessimo conto di dover fare un upgrade di una componente: stesso giro e stesso scambio di gigabyte. Decisamente poco pratico...

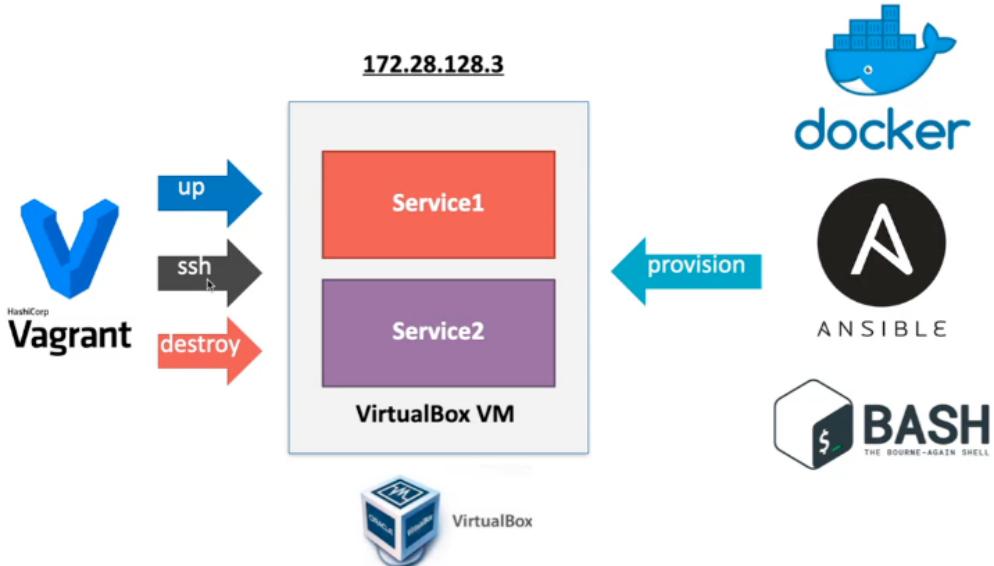
Inoltre, la cosa peggiore, è che tutto questo processo di configurazione della macchina virtuale, anche noto come **provisioning**, non è tracciabile (a meno di qualche passaggio manuale) e quindi non facilmente ripetibile.

Vagrant invece, consente di avere delle macchine virtuali completamente “scriptabili” sia in fase di configurazione (RAM, networking, spazio disco...) che di provisioning (installazione di MongoDB, RabbitMQ...). Ognuno di questi script potrà poi essere aggiunto al nostro source control preferito e arrivare a versionare i vari scenari realizzando quella pratica nota come **Infrastructure as code**.

I vantaggi di questo approccio sono evidenti visto che non avremo più necessità di dover trovare noi un modo per condividere le virtual machine. Per comprendere la semplicità di utilizzo, basta pensare che una macchina virtuale Linux è possibile aviarla con due semplici comandi inseriti in un prompt di Windows.

Un altro vantaggio di Vagrant risiede nella capacità di virtualizzare la virtualizzazione. Infatti sebbene il provider di virtualizzazione di default è Oracle Virtual Box, nessuno ci vieta di cambiarlo e passare, ad esempio a Microsoft Hiper-V, o VMware. Questo passaggio a noi sarà completamente indolore visto che non dovremo toccare nulla nei nostri script Vagrant.

Non solo è possibile astrarre il provider di virtualizzazione sottostante e passare così dall'uno all'altro in maniera trasparente, Vagrant può cambiare completamente tipo di virtualizzazione, passando da quella classica di cui abbiamo parlato finora a quella più moderna basata su container.



1.3 Virtual machine vs Container

La differenza tra le due tecnologie di virtualizzazione è che mentre la prima realizza un isolamento completo tra il sistema operativo host e quello guest, quest'ultima condivide lo stesso kernel. Sebbene i container siano ovviamente più leggeri della virtualizzazione classica, a causa della condivisione del kernel tra host e guest non sarà possibile, ad esempio, avere un container Windows su Linux.

Docker è uno degli esempi più famosi nel panorama dei container: volendo Vagrant può path prestabilito.

1.4 Vagrant, init e up, i comandi fondamentali, [1]

Creiamo una cartella con:

```
$ mkdir vagrant
```

Portiamoci al suo interno e lanciamo uno dei principali comandi dell'ecosistema:

```
$ cd vagrant
$ vagrant init hashicorp/precise32
```

Il comando è diviso in tre parti fondamentali:

- **vagrant** - è il comando principale;
- **init** - è il sotto-comando. In questo caso è stato richiesto di inizializzare un nuovo progetto Vagrant all'interno del path della cartella.
- **hashicorp/precise32** - segnaliamo che il progetto è un sistema operativo di tipo Ubuntu 12.04 Precise Pangolin a 32 bit, in futuro l'immagine di tale sistema (o meglio box) andrà recuperata dal repository Hashicorp.

Lanciato il comando, se tutto andrà per il verso giusto, Vagrant ci informerà dell'aggiunta di un **Vagrantfile** alla directory corrente.

```
(base) pierpaolo@pierpaolo-mint:~/Scaricati/vagrant$ vagrant init hashicorp/precise32
A `Vagrantfile` has been placed in this directory. You are now
ready to `vagrant up` your first virtual environment! Please read
the comments in the Vagrantfile as well as documentation on
`vagrantup.com` for more information on using Vagrant.
```

I sistemi operativi da utilizzare possiamo recuperarli dal repository ufficiale, cercando quello che fa al caso nostro, in alternativa possiamo creare un box personalizzato ed usarlo per i nostri progetti. Per ora Ubuntu è più che sufficiente a soddisfare le nostre esigenze.

1.4.1 Vagrantfile

Questo file è il cuore di un progetto Vagrant e racchiude le caratteristiche della nostra futura macchina virtuale, qui potremo specificare la quantità di RAM, le configurazioni di rete e tanto altro ancora. Possiamo aprire il file con un qualunque editor di testo: il contenuto è scritto in Ruby ma non è necessario conoscere il linguaggio per capire cosa voglia significare, inoltre è fortemente commentato.

Se un Vagrantfile diventa molto verboso, fino a risultare fastidioso, è possibile aggiungere il flag **--minimal** al comando init. Questo è l'output generato dal comando init precedente privato di tutti i commenti:

```
Vagrant.configure(2) do |config|
  config.vm.box = "hashicorp/precise32"
end
```

La prima e la terza riga creano un blocco entro cui la configurazione della macchina virtuale è agganciata alla variabile config. La seconda riga specifica di quale box Vagrant avrà bisogno per avviare la macchina virtuale (e coincide con quanto scritto nel comando init precedentemente).

1.4.2 Vagrant up

Per avviare la nostra macchina virtuale è sufficiente lanciare vagrant up da riga di comando (assicuriamoci di eseguire il comando nella stessa directory in cui abbiamo generato il Vagrantfile). Il comando cercherà sul nostro disco se è presente il box precise32.box (ovviamente ciò non avverrà), altrimenti lo preleverà dal repository centrale, in tal caso l'avvio della macchina virtuale non sarà dei più rapidi.

Per avere informazione sullo stato delle nostre virtual machine, quali sono in esecuzione, quali stoppatte e così via, possiamo digitare il comando:

```
$ vagrant status
```

Possiamo fermare una macchina virtuale utilizzando il comando:

```
$ vagrant halt
```

e farla tornare di nuovo in vita ridigitando:

```
$ vagrant up
```

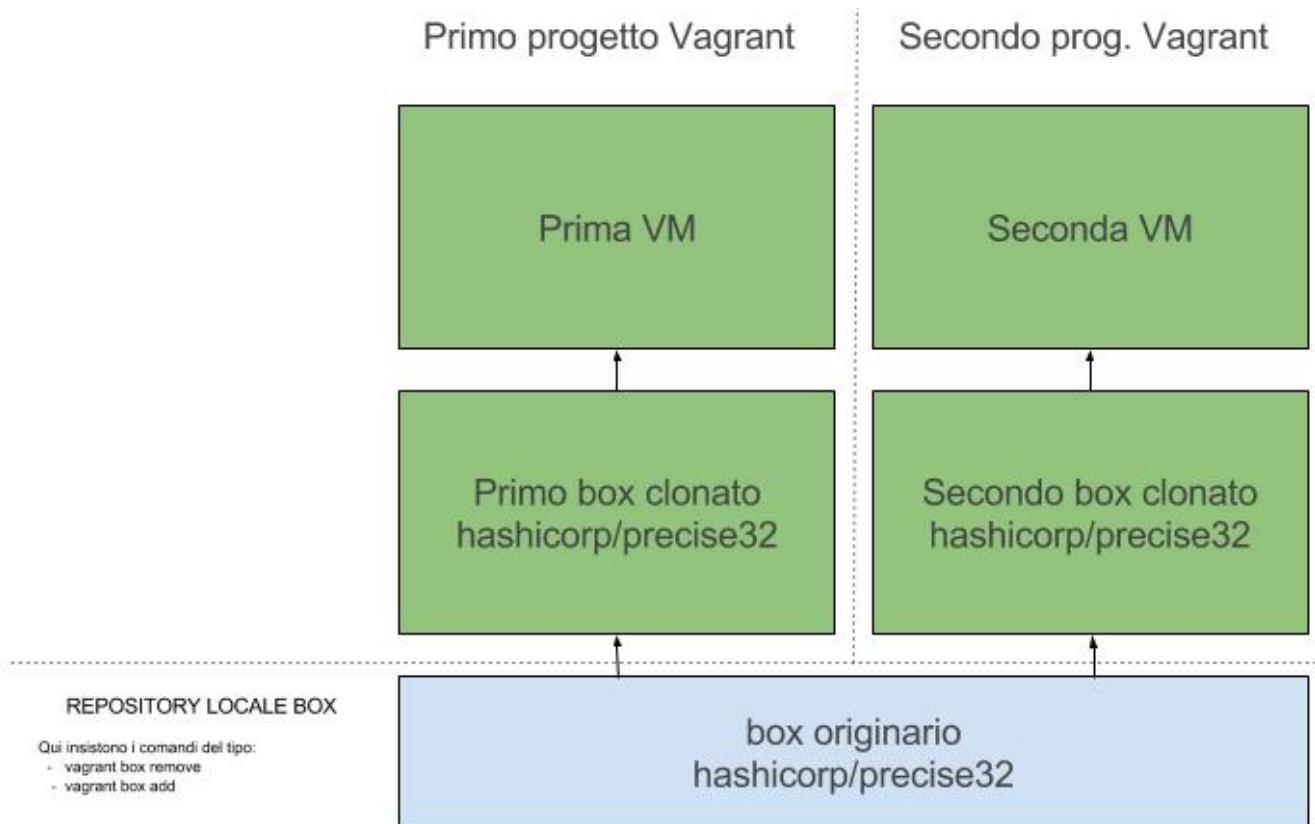
vedremo che i tempi di avvio della macchina saranno molto più veloci rispetto alla prima esecuzione. Il motivo, come si intuisce, sta nel fatto che Vagrant questa volta non ha dovuto scaricare il box del sistema operativo in quanto già presente sul disco locale.

1.5 Vagrant, box e macchine virtuali, [1]

In Vagrant un box è un file .tar compresso, in esso troviamo le immagini di una virtual machine, un Vagrantfile ed un file JSON di metadati. Il suo ruolo è quello di fare da prototipo per tutti i progetti futuri che andremo a creare con Vagrant.

Poiché ogni box contiene un intero sistema operativo, le dimensioni possono variare da qualche centinaio di MB per un sistema Linux headless (ovvero senza interfacce utente) a vari GB per un sistema Windows.

Come già accennato, alla prima esecuzione del comando `vagrant up` un box viene scaricato dal repository specificato nel Vagrantfile e salvato nel nostro hard disk. Questa operazione avviene una sola volta per ogni box: se creiamo un secondo progetto avente come box, ancora una volta, hashicorp/precise32 (come nell'esempio precedente), esso non verrebbe scaricato essendo già presente sul disco.



Quando eseguiamo il comando `vagrant up` per la prima volta per un dato progetto il box, se presente nel nostro repository locale, viene clonato in una directory di progetto. In questo modo avremo due box, quello originario e quello clonato.

La fase di startup di una virtual machine avviene sempre usando il box clonato, tale operazione permette a più virtual machine aventi per origine lo stesso box di essere in esecuzione contemporaneamente senza interferire tra loro.

Ogni modifica apportata ad una macchina virtuale renderà man mano il relativo box clonato sempre più diverso da quello originario.

Qualora la macchina virtuale venisse arrestata, grazie al comando `vagrant halt`, le modifiche effettuate all'interno della macchina virtuale sarebbero ancora visibili al successivo avvio in quanto tali modifiche sono persistite nel box clonato che continuerà ad esistere, sebbene la macchina risulti spenta.

Se vogliamo liberarci del box clonato (ma non di quello originario) e ripartire dall'immagine pulita utilizziamo il comando:

```
$ vagrant destroy
```

in questo caso al successivo avvio non avremo più alcuna traccia di eventuali modifiche, in quanto il box clonato sarà identico a quello presente nel repository locale.

1.6 Vagrant, i comandi principali, [1]

Oltre init e up ci sono una serie di comandi molto importanti per gestire le virtual machines: halt, reload, status, global-status, suspend, resume, destroy.

vagrant init

L'abbiamo già incontrato qualche lezione fa, è uno dei due comandi fondamentali, in quanto si occupa di aggiungere un Vagrantfile alla directory corrente facendola così diventare una directory di progetto Vagrant.

```
$ vagrant init [nome-box] [url-box]
```

Il primo parametro specifica quale box vogliamo usare, nelle lezioni precedenti abbiamo sempre usato *hashicorp/precise32*. Il secondo parametro specifica l'URL da cui scaricare il box, se non fornito verrà utilizzato l'URL del repository di default.

Esistono inoltre tre flag:

- **--force** - se esiste già un Vagrantfile nella directory corrente questo viene sovrascritto;
- **--minimal** - il Vagrantfile generato non conterrà alcun tipo di commento;
- **--output NOMEFILE** - il risultato del comando non verrà scritto su un file di nome Vagrantfile ma sul file specificato.

vagrant up

L'altro comando fondamentale. Ci permette di creare e/o avviare una macchina virtuale in base alle specifiche fornite nel Vagrantfile. Dato un progetto, alla prima esecuzione, il comando:

```
$ vagrant up
```

scaricherà (se non ancora presente) il box e lo salverà sul disco, dopodiché lo clonerà e utilizzerà il box clonato per ogni successivo avvio.

Alcune caratteristiche del comando sono personalizzabili grazie all'utilizzo di [flag specifici](#).

vagrant halt

Questo comando spegne la macchina virtuale. Se possibile, Vagrant proverà a spegnere la macchina attraverso le procedure di spegnimento proprie del sistema operativo guest, altrimenti ne effettuerà uno spegnimento forzato.

Se non siamo interessati ad uno spegnimento *graceful* della macchina possiamo passare direttamente il parametro **-f** o **--force**.

vagrant reload

Ha l'effetto di un vagrant halt seguito da un vagrant up. Utile a far sì che vengano applicate delle modifiche apportate al Vagrantfile, equivale ad un reboot di un computer. Tale comando di default non rieffettua il provisioning dell'ambiente (per ulteriori dettagli sull'argomento è disponibile una lezione in proposito).

vagrant suspend/vagrant resume

L'utilizzo di *vagrant suspend* permette di sospendere la macchina virtuale per poi riprenderla in un secondo momento con *vagrant resume*.

Sospendendo la macchina virtuale, il computer host diverrà più scarico sia a livello di RAM che a livello di CPU, mentre verrà richiesto dello spazio extra su disco per salvare lo stato della macchina in modo che potrà essere successivamente ripresa.

vagrant destroy

Questo comando termina la virtual machine con un *vagrant halt*, dopodiché libera lo spazio occupato dalle risorse associate, in primis il box clonato utilizzato nella fase di startup. Il box originario (o anche base-box) non sarà toccato, perciò un successivo *vagrant up* comporterebbe la clonazione del box ma non il suo download.

Se invece vogliamo eliminare anche il base-box è necessario un *vagrant box remove*.

vagrant status

Recuperare lo stato della macchina virtuale non è un'operazione immediata. Tale comando, già presentato nella lezione precedente, fornisce le informazioni sul progetto corrente, informandoci se la macchina è in esecuzione (running), spenta (poweroff) o mai creata (not created).

Se il nostro progetto Vagrant si trova nello stato *not created* vuol dire che esiste il Vagrantfile di progetto ma non esiste il box clonato, ragion per cui è necessario un *vagrant up*.

Essendo questo comando un comando di progetto non fornirà nessun output utile se all'interno della directory corrente non esiste un Vagrantfile.

vagrant global-status

Se siamo interessati alla situazione complessiva delle nostre macchine questo comando è quello che fa al caso nostro, inoltre, essendo globale può essere invocato da qualunque posizione. L'output è più o meno simile a quello precedente ma oltre alle informazioni classiche sullo stato della macchina, fornisce anche un *id* univoco che in futuro potrà essere molto utile. Per motivi di efficienza l'output del comando è fornito da una cache, per cui alcune invocazioni potrebbero produrre dati non più validi, in tal caso potremo usare il flag *--prune* per eliminare tali occorrenze.

1.7 Connettersi alla macchina virtuale con SSH, [1]

Se non disponiamo di un'interfaccia grafica sulla macchina virtuale, il modo più pratico per connetterci ad essa è tramite un client SSH. Durante la fase di creazione della macchina virtuale (sebbene non specificato esplicitamente all'interno del Vagrantfile) viene configurato l'accesso SSH utilizzando dei parametri di default.

Usare il seguente comando per visualizzare la struttura ad albero:

```
$ tree -a
```

```
(base) pierpaolo@pierpaolo-mint:~/Scaricati/vagrant$ tree -a
.
└── .vagrant
    ├── machines
    │   └── default
    │       └── virtualbox
    │           ├── action_provision
    │           ├── action_set_name
    │           ├── box_meta
    │           ├── creator_uid
    │           ├── id
    │           ├── index_uuid
    │           ├── private_key ←
    │           ├── synced_folders
    │           └── vagrant_cwd
    └── rgloader
        └── loader.rb
Vagrantfile
```

Vagrant creerà per noi anche una chiave da usare al momento dell'autenticazione e la salverà all'interno della directory di progetto.

A questo punto potremmo usare il nostro client SSH di fiducia per connetterci alla macchina fornendo i default che Vagrant ha generato, oppure, in maniera molto più rapida potremo lanciare il comando `vagrant ssh` e attendere:

```
$ vagrant ssh
```

```
(base) pierpaolo@pierpaolo-mint:~/Scaricati/vagrant$ vagrant ssh
Welcome to Ubuntu 12.04.5 LTS (GNU/Linux 3.13.0-32-generic i686)

 * Documentation:  https://help.ubuntu.com/
 New release '14.04.5 LTS' available.
 Run 'do-release-upgrade' to upgrade to it.

vagrant@precise32:~$
```

1.8 Synced folder, [1]

Durante la fase di startup della virtual machine di esempio, una delle ultime operazioni di cui abbiamo notizia è qualcosa di simile:

```
--> default: Mounting shared folders...
      default: /vagrant => /home/pierpaolo/Scaricati/vagrant
```

In questo caso, Vagrant ci sta informando del mapping realizzato tra la directory `/vagrant` sul computer guest e `/home/pierpaolo/Scaricati/vagrant` sul computer host.

Così come nel caso di SSH, non abbiamo scritto nulla di simile nel `Vagrantfile`. Utilizzando una configurazione di `default`, Vagrant ha reso possibile la realizzazione di quella che viene chiamata `synced folder` (`shared folder` nelle versioni precedenti).

A questo punto, se non fosse ancora chiaro, una `synced folder` è un punto di incontro tra il file system della macchina virtuale e quello della macchina locale: in questo caso, tutto il contenuto presente all'interno del path `/vagrant` della macchina guest sarà accessibile anche dal path `/home/pierpaolo/Scaricati/vagrant` della macchina host (e viceversa).

Questo è perfettamente in linea con i dettami del Tao di Vagrant, in quanto grazie a questo meccanismo possiamo modificare dei file presenti sulla macchina virtuale come se fossero sul nostro computer locale, utilizzando quindi i nostri strumenti preferiti. Inoltre i file nella `synced folder` non verranno persi qualora effettuassimo il `destroy` della macchina.

Sebbene molto utili, le synced folder non sono molto performanti per cui non è consigliabile appoggiare al loro interno file su cui andranno eseguite molte operazioni di I/O. Una macchina guest può avere più di una synced folder.

1.8.1 Synced folder oltre i default

Come ogni altra funzionalità di Vagrant, anche questa può essere personalizzata per venire incontro alle nostre esigenze. Per intenderci, se i default del paragrafo precedente non dovessero fare al caso nostro possiamo modificarli accedendo al Vagrantfile e scrivendo qualcosa del genere:

```
[...]
config.vm.synced_folder ".", "/etc/synced_folder"
[...]
```

Il punto indica che, lato host, la synced folder sarà la directory corrente, ovvero, la stessa che contiene il Vagrantfile. Il secondo parametro è invece il path della synced folder per quanto riguarda la macchina guest, se tale path non esistesse questo verrebbe interamente ricreato. Inoltre, il primo parametro può essere scritto sia in maniera relativa (partendo cioè dalla directory di progetto) che assoluta, in entrambi i casi però, questo deve essere un path realmente esistente. Ad esempio, questa definizione:

```
[...]
config.vm.synced_folder "src", "/etc/src"
[...]
```

al momento del *vagrant up* (o *vagrant reload* se la macchina è già in *running*) solleverà un errore in quanto la directory `src` non esiste nella directory di progetto. A questo punto abbiamo due strade: crearla a mano, o aggiungere una proprietà che lo faccia al posto nostro:

```
[...]
config.vm.synced_folder "src", "/etc/src", create: true
[...]
```

A questo punto se andassimo a spulciare l'output della fase di *up* della nostra macchina noteremo due synced folder: quella di default che punta alla directory di progetto (quella col Vagrantfile) e quella che punta ad `src`. Se volessimo sbarazzarci della prima utilizzeremo per questa la proprietà `disabled`, e il nostro Vagrantfile diventerebbe:

```
[...]
config.vm.synced_folder ".", "/vagrant", disabled: true
config.vm.synced_folder "src", "/etc/src", create: true
[...]
```

Al successivo avvio della macchina vedremo solo la seconda synced folder.

1.9 Provisioning di una macchina virtuale, [1]

Una delle fasi più noiose nella preparazione di un ambiente, sia questo di sviluppo, collaudo o quant'altro, è sicuramente l'installazione e la configurazione delle componenti software. Che versione installiamo? A quale URL trovo il pacchetto? Come si chiama il database che devo creare? Esiste un ordine preciso in cui le singole componenti vanno installate? Tra qualche mese sarò in grado di rifare tutto il processo senza errori?

Queste sono solo alcune delle domande che emergono e il più delle volte la risposta è: "Ok, scriviamo un documento!". Perfetto, in alcune situazioni la scrittura di un documento che guida

passo passo la procedura di provisioning è la soluzione più pratica. Spesso però i documenti non vengono adeguatamente condivisi, non sempre è presente un criterio di organizzazione degli stessi e non è difficile che questi siano completamente disallineati da quella che è poi diventata, col tempo, l'infrastruttura. Ma se anche così non fosse c'è un problema ancora peggiore: seguire una procedura passo passo è un'esperienza noiosa, la concentrazione cala e la possibilità di introdurre errori o saltare qualche passaggio è alta.

Quanto appena detto vale per noi esseri umani, non per il nostro computer, lui è un vero fenomeno in questo tipo di operazioni... purché riscriviamo il documento di installazione in modo che possa comprenderlo.

1.9.1 Shell provisioning

In Vagrant abbiamo diversi modi per effettuare il provisioning di una macchina, il primo e più immediato è chiamato shell provisioning in quanto si affida alla shell del sistema ospite. Nel nostro caso, avendo creato una macchina virtuale Linux la shell sarà una classica Bash. Supponiamo, ad esempio, che la nostra applicazione abbia bisogno per funzionare di MongoDB 2.6.4. Avendo bene in mente i passi per l'installazione (magari sono scritti nel famigerato documento di cui parlavamo ad inizio lezione) creeremo il relativo script di installazione. Il risultato sarà qualcosa del genere:

```
#!/usr/bin/env bash
apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10
echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist
10gen' | tee /etc/apt/sources.list.d/mongodb.list
apt-get update
apt-get install mongodb-org=2.6.4 mongodb-org-server=2.6.4
mongodb-org-shell=2.6.4 mongodb-org-mongos=2.6.4 mongodb-org-tools=2.6.4 -y
echo "mongodb-org hold" | dpkg --set-selections
echo "mongodb-org-server hold" | dpkg --set-selections
echo "mongodb-org-shell hold" | dpkg --set-selections
echo "mongodb-org-mongos hold" | dpkg --set-selections
echo "mongodb-org-tools hold" | dpkg --set-selections
sed -i '/bind_ip = 127.0.0.1/,/bind_ip\ |=\ 127\.0\.0\.1/s/^/#/'
/etc/mongod.conf
service mongod restart
```

Salviamo il documento nella nostra directory di progetto assegnandogli estensione .sh.

Nulla di trascendentale, ma qualcosa già è cambiato: prima di tutto abbiamo una chiara visione di insieme della lista di comandi da eseguire, partendo dall'update del repository di Ubuntu fino all'avvio del demone di MongoDB. In secondo luogo lo script è versionabile e sarà quindi possibile condividerlo, migliorarlo e correggerlo tenendo sempre traccia di chi ha modificato cosa.

A questo punto, non ci resta che spiegare a Vagrant, dove trovare lo script e questo comando da scrivere nel Vagrantfile è tutto quello che serve:

```
config.vm.provision "shell", path: "mongo-provision.sh"
```

In questo modo stiamo istruendo Vagrant per effettuare uno shell provisioning dal file chiamato mongo-provision.sh. Poiché questo file risiede nella directory di progetto non avremo bisogno di specificare un vero e proprio path, basterà solo il nome.

Verifichiamo con il comando `vagrant status` la situazione della nostra macchina, se questa risulterà *not created*, al primo `vagrant up` sarà eseguito lo script di provisioning. Se la macchina è in *running* o in *poweroff* dovremo forzare il provisioning rispettivamente con:

```
$ vagrant reload --provision
```

oppure:

```
$ vagrant up --provision
```

1.9.2 Shell provisioning oltre i default

Il comando di provisioning è abbastanza flessibile e ci permette di personalizzare il suo comportamento. Possiamo decidere, ad esempio, di eseguire un determinato comando ogni volta che si avvia la macchina virtuale, grazie al modificatore *always* che impone l'esecuzione dello script ogni volta che lanciamo `vagrant up` o `vagrant reload`:

```
config.vm.provision "shell", path: "service-start.sh", run: "always"
```

Potremmo anche non avere necessità di utilizzare un file `.sh`, magari perché il comando da lanciare è abbastanza breve. Supponiamo di voler inserire in un file `session.log` la data completa della macchina virtuale quando questa viene avviata, un comando del genere è quello che può fare al caso nostro:

```
config.vm.provision "shell", inline: "echo $(date) >> session.log", run: "always"
```

Sebbene del tutto simile al precedente, non possiamo fare a meno di notare l'opzione `inline` che ha preso il posto di `path`, al cui interno non è più presente un riferimento al file con i comandi ma direttamente il comando da eseguire.

2. Bash, [1], [2], [3]

Cheatography

BASH Scripting
by datamansam via cheatography.com/139410/cs/32170/

Lists and List Options <code>ls</code> List all values in present working directory <code>ls -</code> List all files in sub-directories as well <code>R</code> <code>ls -</code> List hidden files as well <code>a</code>	For loop (cont) <code>2</code> <code>5</code> <code># For loop; Three expression</code> <code>for ((x=2;x<=4;x+=2))</code> <code>do</code> <code> echo \$x</code> <code>done</code> Output: <code>2</code> <code>4</code>	Directory with a for loop <code># Search for books in the book directory that contain air</code> <code>for book in \$(ls books/ grep -i 'air')</code> <code>do</code> <code> echo \$book</code> <code>done</code> <code>AirportBook.txt</code> <code>FairMarketBook.txt</code>
Moving and Renaming Files <code>mv file "new file path"</code> Moves the files to the new location <code>mv filename</code> Renames the file to a new_file_name new filename	Home Directory <code>cd</code> Navigate to home directory <code>cd ..</code> Move one level up <code>cd /</code> Move to root directory	Case Statements <code>case 'STRING' in</code> <code>PATTERN1)</code> <code>COMMAND1;;</code> <code>PATTERN2)</code> <code>COMMAND2;;</code> <code>*)</code> <code>DEFAULT COMMAND;;</code> <code>esac</code> <code>case \$(cat \$1) in</code> <code>sydney)</code> <code>mv \$1 sydney/ ;;</code> <code>melbourne brisbane)</code> <code>rm \$1 ;;</code> <code>canberra)</code> <code>mv \$1 "IMPORTANT_\$1" ;;</code> <code>*)</code> <code>echo "No cities found" ;;</code> <code>esac</code>
Insert mode <code>i</code> insert at cursor <code>a</code> Write after cursor <code>A</code> Write at end of line <code>ESC</code> Terminate insert mode <code>u</code> Undo last change <code>U</code> undo change to entire last line	Process Management <code>ps</code> Display currently running processes <code>ps -</code> Display currently running processes on system	Search Files <code>grep pattern files</code> <code>grep -i</code> Case insensitive search <code>grep -R 'httpd'</code> Look for all files in the current directory and in all of its subdirectories <code>grep -c 'nixcraft'</code> Search and display the total number of times that the string 'nixcraft' appears in a file <code>frontpage.md</code>
For loop <code># For loop in Bash; Basic</code> <code>for x in 1 2 3</code> <code>do</code> <code> echo \$x</code> <code>done</code> <code>1</code> <code>2</code> <code>3</code> <code># For loop in Bash; Range {START..STOP..INCREMENT}</code> <code>for x in {1..5..2}</code> <code>do</code> <code> echo \$x</code> <code>done</code> Output: <code>1</code>		Creating and viewing a file <code>Creates a new file</code> <code>cat > filename</code> <code>Displays the file content</code> <code>cat filename</code>



By datamansam

cheatography.com/datamansam/

Published 28th May, 2022.
Last updated 30th May, 2022.
Page 1 of 2.

Sponsored by [CrosswordCheats.com](#)
Learn to solve cryptic crosswords!
<http://crosswordcheats.com>

BASH variables	
Set with an equal sign	greeting="Hello"
Access with a \$	echo \$greeting

Control Flow	
Direct Output of Second command to first	1st_command \$ 2nd_Command
Runs the 2nd command only if the 1st command runs successfully.	1stcommand && 2nd Command
Runs the 2nd command only if the 2nd command does not run successfully.	A B # Run B if and only if A failed

Navigating With Cursor	
Alt + b	move backward one word
Alt + f	move forward one word
Alt + u	make entire word after cursor uppercase
Alt + c	make first letter after cursor uppercase
Alt + d	delete word after cursor
^a	beginning of line
^e	end of line

While Statement	
Set a condition which is tested at each iteration:	x=1 while [\$x -le 3]; do echo \$x ((x+=1)) done

if, then, else statement	
if then else: x="Queen" if [CONDITION]; if [\$x == "King"]; then # SOME CODE then echo "\$x is a King!" else # SOME OTHER CODE else echo "\$x is not a King!" fi fi	



By datamansam

cheatography.com/datamansam/

Published 28th May, 2022.
Last updated 30th May, 2022.
Page 2 of 2.

Sponsored by [CrosswordCheats.com](#)
Learn to solve cryptic crosswords!
<http://crosswordcheats.com>

Bash è una “Unix shell”, vale a dire un’interfaccia a linea (o riga) di comando che permette l’interazione con sistemi operativi derivati da Unix. È disponibile su gran parte dei sistemi operativi moderni di questo tipo, essendo la shell predefinita di molte distribuzioni GNU/Linux, nonché di Mac OS X.

Cos’è lo Shell Scripting?

In aggiunta alla modalità interattiva, che permette all’utente di eseguire un comando alla volta e ricevere il risultato in tempo reale, Bash (così come molte altre shell) ha la capacità di eseguire interi script di comandi, conosciuti come “Bash shell script” (“Bash script”, “shell script” o più semplicemente “script”). Uno script può contenere una semplice lista di comandi – o anche un singolo comando – così come funzioni, cicli, istruzioni condizionali, e tutti gli altri costrutti tipici della programmazione imperativa. In definitiva, un Bash shell script è un programma scritto nel linguaggio di programmazione Bash.

Lo Shell Scripting è l’arte di creare e modificare tali script.

I Bash script possono essere richiamati dalla modalità interattiva descritta in precedenza, oppure possono essere eseguiti da altre parti del sistema: uno script può essere richiamato ogni volta che il sistema si avvia, un altro ogni giorno alle 2:30 del mattino, un altro ancora ogni qualvolta un utente si autentica nel sistema.

Perché programmare la shell?

I Bash script vengono comunemente usati per molte attività di amministrazione di sistema, come l’analisi dei log, il backup dei dischi, e così via. Sono anche molto usati come script di installazione di programmi complessi. Insomma, la conoscenza pratica dello Shell Scripting è essenziale per tutti coloro che desiderano diventare degli amministratori di sistema esperti, poiché questi permettono di risolvere problemi complessi pur richiedendo spesso istruzioni semplici. Per esempio, se si ha bisogno di richiamare due programmi esterni, è possibile creare uno script composto solamente da due linee; al tempo stesso, la shell ci mette a disposizione tutta la potenza e abilità decisionale di un linguaggio di programmazione completo.

2.1 Un primo esempio

```
#!/bin/bash

# Questo è un commento
MONDO="World"                      # dichiarazione di una variabile
echo "Hello $MONDO"                  # $MONDO viene sostituita con il valore della variabile MONDO
echo "Today: `date +%d/%m/%Y`"       # il comando tra apici inversi viene sostituito con il suo risultato: `comando`
echo "Today: $(date +%d/%m/%Y)"      # analogo al precedente per sostituire il comando con il suo risultato: $(comando)
echo 'Hello `echo $MONDO`'           # tra apici singoli non vengono valutati i caratteri speciali come $ e `
echo "Hello"; echo "$MONDO"          # più comandi possono essere separati da ;

# Output
# Hello World
# Today: 15/07/2021
# Today: 15/07/2021
# Hello `echo $MONDO`
# Hello
# World
```

La prima riga informa il sistema che il file deve essere eseguito da `/bin/bash`, che corrisponde al percorso standard della Bourne-Again Shell su qualsiasi sistema Unix based. Esempio di script `hello.sh` (nota, l’estensione `.sh` è solo una convenzione). Il file con lo script può essere passato in input all’interprete bash: `bash hello.sh`

Eseguire lo script:

Una volta creato il file, avremo la necessità di eseguire lo script che contiene. Per farlo, apriamo il terminale e posizioniamoci nella directory contenente il file appena creato, tramite il comando `cd` seguito dal percorso completo. Adesso eseguiamo il comando `chmod` per rendere il file di testo eseguibile, utilizzando la sintassi seguente:

```
chmod +rx hello.sh
```

Infine eseguiamo il nostro script, direttamente dal terminale, come segue:

```
./hello.sh
```

Variabili:

In seguito lo script di esempio utilizza la variabile `MONDO` per memorizzare il valore `world`, e stampa il solito messaggio "Hello World":

```
MONDO=World  
echo "Hello ${MONDO}"
```

La stringa `${MONDO}` nella terza riga è stata sostituita da "World" prima dell'effettiva esecuzione del comando; questa operazione è conosciuta come *espansione di variable*, ed è molto più flessibile di come ci si aspetti. Ad esempio, si può anche memorizzare il nome del comando da eseguire:

```
cmd_to_run=echo  
${cmd_to_run} "Hello World"
```

2.2 Redirezione dell'I/O, [1]

```
#!/bin/bash  
  
echo "Benvenuto!"          # stampa di una stringa nello standard output  
  
read -p "inserisci: " str  # lettura di una stringa dallo standard input (es. input da tastiera)  
echo "letta: $str"  
  
echo "riga_1" > file.txt   # redirigo lo standard output nel file  
echo "riga_2" >> file.txt  # redirigo lo standard output in append al file  
  
echo "gino" 1> out.txt     # redirigo lo standard output nel file  
rm file_inesistente.txt 2> err.txt  # redirigo lo standard error nel file  
rm file_inesistente.txt &> out_err.txt # redirigo sia lo standard error che lo standard output su file  
  
wc -l < file.txt          # invio il contenuto del file nello standard input del comando (output: 2)  
  
read -p "inserisci: " str < file.txt  # leggo solo la prima riga  
echo "letta $str"           # output: letta riga_1  
  
cat file.txt | grep "_2" | wc -l    # uso la pipe per concatenare più comandi (output: 1)
```

2.3 Espressioni condizionali, [1]

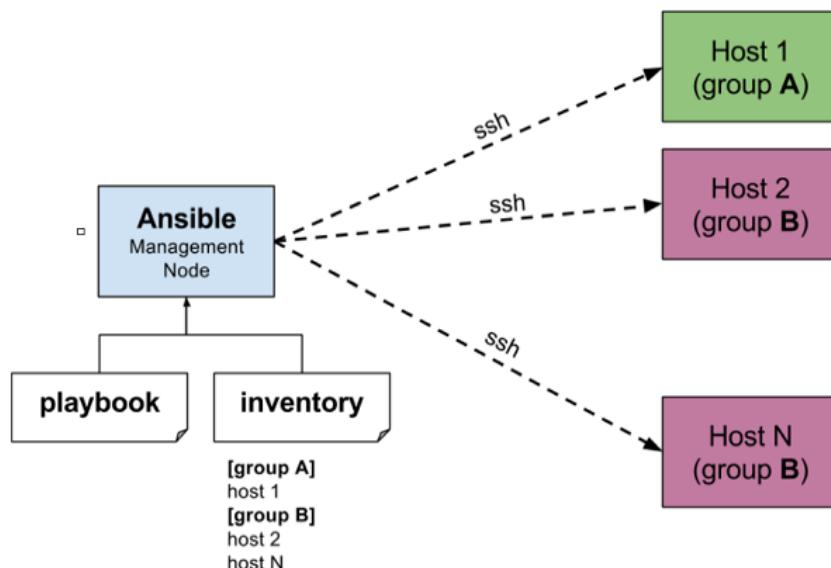
.....

3. Ansible, [1], [2], [3]

Ansible è uno strumento open source per l'automazione IT che consente di automatizzare il provisioning, la gestione della configurazione, il deployment delle applicazioni, l'orchestrazione e molti altri processi IT manuali. Gli utenti di Ansible (amministratori di sistema, sviluppatori e devops) possono sfruttare l'automazione Ansible per installare i software, automatizzare le attività quotidiane, eseguire il provisioning dell'infrastruttura, migliorare i livelli di sicurezza e conformità, applicare patch ai sistemi e condividere l'automazione in tutta l'azienda.

In particolare:

- Ansible è un software free che consente di automatizzare le procedure di configurazione e gestione sui sistemi unix e derivati.
- È scritto in *python* ed è multipiattaforma.
- È *agentless*, ovvero non abbiamo la necessità di installare demoni sui nodi da controllare.
- Tutti i file ansible possono essere scritti in [YAML](#).
- Il "node manager", il quale è l'unico a cui è richiesto che sia installato Ansible, si collega ai "nodes" da controllare attraverso *SSH*.
- Il file denominato *Hosts* contiene l'elenco dei nodi che andremo a controllare.
- Il file denominato *Playbook* contiene tutti gli step che andremo ad eseguire sui nostri nodi/o.



Ansible automatizza la gestione dei sistemi remoti e ne controlla lo stato desiderato. Un ambiente Ansible di base ha tre componenti principali:

- Control node (nodo di controllo). Un sistema su cui è installato Ansible. Su un nodo di controllo si eseguono comandi Ansible come `ansible` o `ansible-inventory`.
- Managed node (nodo gestito): Un sistema remoto, o host, che Ansible controlla.
- Inventory: Un elenco di nodi gestiti organizzati logicamente. Si crea un inventario sul nodo di controllo per descrivere le distribuzioni degli host ad Ansible.

Installazione di Ansible:

NB: Non è possibile installare ansible in una macchina Windows.

- Per installare Ansible su RHEL/CentOS possiamo utilizzare l' Extra Packages for Enterprise Linux (**EPEL**) repository. Quindi possiamo lanciare i seguenti comandi:


```
sudo dnf install epel-release
sudo dnf install ansible
```
- Per installare Ansible su Debian/Ubuntu possiamo utilizzare il Personal Package Archive (**PPA**). Quindi possiamo lanciare i seguenti comandi:


```
sudo apt-add-repository -y ppa:ansible/ansible
sudo apt-get update
sudo apt-get install -y ansible
```
- Infine per verificare la corretta installazione, usiamo il comando:


```
sudo ansible --version
```

Configurazioni delle chiavi SSH:

- Ci permette di collegarci alla macchina target senza l'utilizzo di password
- Per prima cosa generiamo le chiavi sulla nostra macchina, con il comando:


```
ssh-keygen -t rsa -N '' -f ~/.ssh/id_rsa
```
- Questo comando genererà i file: id_rsa.pub e id_rsa
- Adesso dobbiamo copiare la chiave pubblica **id_rsa.pub** su tutte le macchine target su cui vogliamo effettuare il collegamento
- E' possibile utilizzare il tool **ssh-copy-id**, e quindi il comando:


```
ssh-copy-id root@192.168.0.XXX
```
- Per un rapido test:


```
ssh root@192.168.0.XXX "hostname -I"
```

3.1 Gli Inventory, [1]

- E' possibile creare una lista di host (ip/dns), o raggrupparli
- I formati del file più comuni sono **INI** e **YAML**
- La posizione predefinita per l'inventory è **/etc/ansible/hosts**, ma è anche possibile cambiare il path. Un'altra opzione è quella di estrarre l'inventory dinamicamente dal cloud
- Raggruppa gli host in modo logico, utilizzando la **best practice what, where, when**
- Un esempio di file inventory di tipo **INI** è mostrato qui di fianco
- Possiamo anche evitare di elencare tutti gli ip sequenziali manualmente definendo un range:


```
192.168.1.[10:12]
```
- E' possibile anche inserire variabili nell'inventory, sia per gli host che per i gruppi. Es: **mail.example.com ansible_connection=ssh ansible_user=myuser**

```
mail.example.com
[webservers]
foo.example.com
bar.example.com

[dbservers]
192.168.1.10
192.168.1.11
192.168.1.12
```

Gli inventory organizzano i nodi gestiti in file centralizzati che forniscono ad Ansible informazioni sul sistema e sulle posizioni di rete. Utilizzando un file di inventory, Ansible può gestire un gran numero di host con un solo comando. Gli inventory aiutano anche a utilizzare Ansible in modo più efficiente, riducendo il numero di opzioni della riga di comando da specificare. Ad esempio, gli inventory di solito contengono l'utente SSH, quindi non è necessario includere il flag **-u** quando si eseguono i comandi di Ansible.

Nella sezione precedente, si sono aggiunti i nodi gestiti direttamente al file `/etc/ansible/hosts`.

Ora creiamo un file inventory da aggiungere al controllo sorgente per garantire flessibilità, riutilizzo e condivisione con altri utenti.

I file di inventory possono essere in formato INI o YAML. A scopo dimostrativo, questa sezione utilizza solo il formato YAML.

Eseguire i seguenti passaggi:

1. Apri il terminale del control node;
2. Crea un nuovo file inventory denominato `inventory.yaml` all'interno di qualsiasi cartella e aprilo per modificarlo.
3. Aggiungere un nuovo gruppo per gli host, quindi specificare l'indirizzo IP o il nome di dominio completamente qualificato (FQDN) di ogni nodo gestito con il campo `ansible_host`. L'esempio seguente aggiunge gli indirizzi IP di tre macchine virtuali in KVM:

```
virtualmachines:  
  hosts:  
    vm01:  
      ansible_host: 192.0.2.50  
    vm02:  
      ansible_host: 192.0.2.51  
    vm03:  
      ansible_host: 192.0.2.52
```

4. verificare l'inventory. Se l'inventory è stato creato in una directory diversa dalla propria home directory, specificare il percorso completo con l'opzione `-i`.

```
ansible-inventory -i inventory.yaml --list
```

5. Eseguire il ping dei nodi gestiti nell'inventario. In questo esempio, il nome del gruppo è `virtualmachines`, che si può specificare con il comando `ansible` invece di `all`.

```
ansible virtualmachines -m ping -i inventory.yaml
```

```
vm03 | SUCCESS => {  
  "ansible_facts": {  
    "discovered_interpreter_python": "/usr/bin/python3"  
  },  
  "changed": false,  
  "ping": "pong"  
}  
vm02 | SUCCESS => {  
  "ansible_facts": {  
    "discovered_interpreter_python": "/usr/bin/python3"  
  },  
  "changed": false,  
  "ping": "pong"  
}  
vm01 | SUCCESS => {  
  "ansible_facts": {  
    "discovered_interpreter_python": "/usr/bin/python3"  
  },  
  "changed": false,  
  "ping": "pong"  
}
```

3.1.1 Suggerimenti per la creazione degli inventory, [1]

Assicurarsi che i nomi dei gruppi siano significativi e unici. I nomi dei gruppi sono sensibili alle maiuscole e alle minuscole.

Evitare spazi, trattini e numeri precedentemente ai nomi dei gruppi.

Raggruppare gli host nell'inventario in modo logico in base a What (Cosa), Where (Dove) e Quando (When).

- Cosa. Raggruppa gli host in base alla topologia, ad esempio: db, web, leaf, spine.
- Dove. Raggruppa gli host in base alla posizione geografica, ad esempio: datacenter, regione, piano, edificio.
- Quando. Raggruppa gli host in base alla fase, ad esempio: sviluppo, test, staging, produzione.

Usare i metagroups, [1]

Creare un metagrupo che organizzi più gruppi nell'inventario con la seguente sintassi:

```
metagroupname:  
  children:
```

Il seguente inventario illustra la struttura di base di un data center. Questo esempio di inventario contiene un metagrupo **network** che include tutti i dispositivi di rete e un **metagrupo** datacenter che include il gruppo **rete** e tutti i server web.

```
leafs:  
  hosts:  
    leaf01:  
      ansible_host: 192.0.2.100  
    leaf02:  
      ansible_host: 192.0.2.110  
  
spines:  
  hosts:  
    spine01:  
      ansible_host: 192.0.2.120  
    spine02:  
      ansible_host: 192.0.2.130  
  
network:  
  children:  
    leafs:  
    spines:  
  
webservers:  
  hosts:  
    webserver01:  
      ansible_host: 192.0.2.140  
    webserver02:  
      ansible_host: 192.0.2.150  
  
datacenter:  
  children:  
    network:  
    webservers:
```

Creare variabili, [1]

Le variabili impostano i valori per i nodi managed, come l'indirizzo IP, l'FQDN, il sistema operativo e l'utente SSH, in modo da non doverli passare quando si eseguono i comandi di Ansible.

Le variabili possono essere applicate a host specifici:

```
webservers:  
  hosts:  
    webserver01:  
      ansible_host: 192.0.2.140  
      http_port: 80  
    webserver02:  
      ansible_host: 192.0.2.150  
      http_port: 443
```

Le variabili possono anche essere applicate a tutti gli host di un gruppo:

```
webservers:  
  hosts:  
    webserver01:  
      ansible_host: 192.0.2.140  
      http_port: 80  
    webserver02:  
      ansible_host: 192.0.2.150  
      http_port: 443  
  
vars:  
  ansible_user: my_server_user
```

3.2 Playbooks, [1]

- Contengono i passaggi che l'utente desidera eseguire su una o più macchine
- Sono file scritti in YAML
- Vengono eseguiti dall'alto verso il basso
- I Playbook contengono Plays, i Plays contengono Tasks e i Tasks chiamano i Modules
- Il comando per lanciare un playbook è:
`ansible-playbook -i hosts ping_all_hosts.yaml`
- Qui di fianco un esempio di playbook

```
---  
- name: update db servers  
hosts: databases  
remote_user: root  
  
tasks:  
- name: ensure postgresql is at the latest version  
  yum:  
    name: postgresql  
    state: latest  
- name: ensure that postgresql is started  
  service:  
    name: postgresql  
    state: started
```

3.2 Conditionals, [1]

- Alcune volte abbiamo la necessità di variare il comportamento del nostro playbook in base ad alcune condizioni
- L'istruzione condizionale più semplice si applica al singolo task, utilizzando la parola chiave **when**
- Quando verrà lanciato quel task, Ansible valuterà quella condizione per tutti gli hosts
- Su tutti gli hosts che soddisferanno quella condizione, verrà eseguito quel task
- Possiamo applicare ad esempio condizioni basate su:
ansible_facts: attributi di singoli host
registered variables: risultati di task passati
variables: di playbook o inventory

In un playbook, è possibile che si vogliano eseguire attività diverse o avere obiettivi diversi, a seconda del valore di un fatto (dati sul sistema remoto), di una variabile o del risultato di

un'attività precedente. È possibile che il valore di alcune variabili dipenda dal valore di altre variabili. Oppure si possono creare altri gruppi di host in base alla corrispondenza con altri criteri. Tutte queste cose si possono fare con i conditionals.

La dichiarazione condizionale più semplice si applica a un singolo task. Si crea il task, quindi si aggiunge un'istruzione `when` che applica un test. La clausola `when` si usa senza doppie parentesi graffe. Quando si esegue il task o il playbook, Ansible valuta il test per tutti gli host. Su ogni host in cui il test viene superato (restituisce il valore True), Ansible esegue il task. Ad esempio, se si sta installando mysql su più macchine, alcune delle quali hanno SELinux abilitato, si potrebbe avere un task per configurare SELinux per consentire l'esecuzione di mysql. Si vorrebbe che questo task venisse eseguito solo sulle macchine che hanno SELinux abilitato:

```
tasks:
  - name: Configure SELinux to start mysql on any port
    ansible.posix.seboolean:
      name: mysql_connect_any
      state: true
      persistent: yes
    when: ansible_selinux.status == "enabled"
    # all variables can be used directly in conditionals without double curly braces
```

3.2.1 Condizionali basati su `ansible_facts`, [1]

Spesso si desidera eseguire o saltare un'attività in base a dei fatti o condizioni. I fatti sono attributi dei singoli host, tra cui l'indirizzo IP, il sistema operativo, lo stato di un filesystem e molti altri. Con i condizionali basati sui fatti, per esempio:

- È possibile installare un determinato pacchetto solo se il sistema operativo è di una determinata versione.
- Si può saltare la configurazione di un firewall su host con indirizzi IP interni.
- È possibile eseguire operazioni di pulizia solo quando un filesystem è pieno.

Non tutti i fatti esistono per tutti gli host. Ad esempio, il fatto 'lsb_major_release' usato nell'esempio seguente esiste solo quando il pacchetto lsb_release è installato sull'host di destinazione. Per vedere quali fatti sono disponibili sui vostri sistemi, aggiungete un task di debug al vostro playbook:

```
- name: Show facts available on the system
  ansible.builtin.debug:
    var: ansible_facts
```

Ecco un esempio di condizionale basato su un fatto:

```
tasks:
  - name: Shut down Debian flavored systems
    ansible.builtin.command: /sbin/shutdown -t now
    when: ansible_facts['os_family'] == "Debian"
```

Se si hanno più condizioni, è possibile raggrupparle con le parentesi:

```
tasks:
  - name: Shut down CentOS 6 and Debian 7 systems
    ansible.builtin.command: /sbin/shutdown -t now
    when: (ansible_facts['distribution'] == "CentOS" and ansible_facts['distribution_major_version'] == "6") or
          (ansible_facts['distribution'] == "Debian" and ansible_facts['distribution_major_version'] == "7")
```

È possibile utilizzare gli operatori logici per combinare le condizioni. Quando si hanno più condizioni che devono essere tutte vere (cioè un "and" logico), è possibile specificarle come un elenco:

```
tasks:  
  - name: Shut down CentOS 6 systems  
    ansible.builtin.command: /sbin/shutdown -t now  
    when:  
      - ansible_facts['distribution'] == "CentOS"  
      - ansible_facts['distribution_major_version'] == "6"
```

Se un fatto o una variabile è una stringa e si deve eseguire un confronto matematico su di essa, utilizzare un filtro per assicurarsi che Ansible legga il valore come un numero intero:

```
tasks:  
  - ansible.builtin.shell: echo "only on Red Hat 6, derivatives, and later"  
    when: ansible_facts['os_family'] == "RedHat" and ansible_facts['lsb']['major_release'] | int >= 6
```

3.2.2 Condizionali basati su registered variables, [1]

Spesso in un playbook si desidera eseguire o saltare un'attività in base al risultato di un'attività precedente. Ad esempio, si potrebbe voler configurare un servizio dopo che è stato aggiornato da un'attività precedente. Per creare una condizione basata su una variabile registrata:

- Registrare il risultato dell'attività precedente come variabile.
- Creare un test condizionale basato sulla variabile registrata.

Il nome della variabile registrata viene creato con la parola chiave `register`. Una variabile registrata contiene sempre lo stato dell'attività che l'ha creata e l'output generato dall'attività. È possibile utilizzare le variabili registrate nei modelli e nelle linee di azione, nonché nelle istruzioni condizionali `when`. È possibile accedere al contenuto della stringa della variabile registrata utilizzando `variable.stdout`. Ad esempio:

```
- name: Test play  
  hosts: all  
  
  tasks:  
  
    - name: Register a variable  
      ansible.builtin.shell: cat /etc/motd  
      register: motd_contents  
  
    - name: Use the variable in conditional statement  
      ansible.builtin.shell: echo "motd contains the word hi"  
      when: motd_contents.stdout.find('hi') != -1
```

È possibile utilizzare i risultati registrati nel ciclo di un task se la variabile è un elenco. Se la variabile non è un elenco, è possibile convertirla in un elenco, con `stdout_lines` o con `variable.stdout.split()`. È anche possibile dividere le righe in base ad altri campi:

```

- name: Registered variable usage as a loop list
  hosts: all
  tasks:

    - name: Retrieve the list of home directories
      ansible.builtin.command: ls /home
      register: home_dirs

    - name: Add home dirs to the backup spooler
      ansible.builtin.file:
        path: /mnt/bkspool/{{ item }}
        src: /home/{{ item }}
        state: link
      loop: "{{ home_dirs.stdout_lines }}"
      # same as loop: "{{ home_dirs.stdout.split() }}"

```

Il contenuto della stringa di una variabile registrata può essere vuoto. Se si vuole eseguire un altro task solo su host in cui lo stdout della variabile registrata è vuoto, si deve controllare che il contenuto della stringa della variabile registrata non sia vuoto:

```

- name: check registered variable for emptiness
  hosts: all

  tasks:

    - name: List contents of directory
      ansible.builtin.command: ls mydir
      register: contents

    - name: Check contents for emptiness
      ansible.builtin.debug:
        msg: "Directory is empty"
      when: contents.stdout == ""

```

Ansible registra sempre qualcosa in una variabile registrata per ogni host, anche su quelli in cui un task fallisce o Ansible salta un task perché una condizione non è soddisfatta. Per eseguire un'attività successiva su questi host, interrogare la variabile registrata per gli host saltati (non per "undefined" o "default"). Ecco un esempio di condizionali basati sul successo o sul fallimento di un task: (Ricordate di ignorare gli errori se volete che Ansible continui l'esecuzione su un host quando si verifica un fallimento)

```

tasks:
  - name: Register a variable, ignore errors and continue
    ansible.builtin.command: /bin/false
    register: result
    ignore_errors: true

  - name: Run only if the task that registered the "result" variable fails
    ansible.builtin.command: /bin/something
    when: result is failed

  - name: Run only if the task that registered the "result" variable succeeds
    ansible.builtin.command: /bin/something_else
    when: result is succeeded

  - name: Run only if the task that registered the "result" variable is skipped
    ansible.builtin.command: /bin/still/something_else
    when: result is skipped

```

3.2.3 Condizionali basati su variabili, [1]

È anche possibile creare condizionali basati su variabili definite nei playbook o nell'inventario. Poiché le condizioni richiedono un input booleano (un test deve essere valutato come Vero per attivare la condizione), è necessario applicare il filtro `| bool` alle variabili non booleane, come le variabili string con contenuto come 'yes', 'on', '1' o 'true'. Si possono definire variabili come questa:

```

vars:
  epic: true
  monumental: "yes"

```

Con le variabili di cui sopra, Ansible eseguirà uno di questi task e salterà l'altro:

```

tasks:
  - name: Run the command if "epic" or "monumental" is true
    ansible.builtin.shell: echo "This certainly is epic!"
    when: epic or monumental | bool

  - name: Run the command if "epic" is false
    ansible.builtin.shell: echo "This certainly isn't epic!"
    when: not epic

```

Utilizzo dei condizionali nei cicli:

Se si combina un'istruzione `when` con un ciclo, Ansible elabora la condizione separatamente per ogni elemento. Questo è il motivo per cui è possibile eseguire l'attività su alcuni elementi del ciclo e saltarla su altri. Ad esempio:

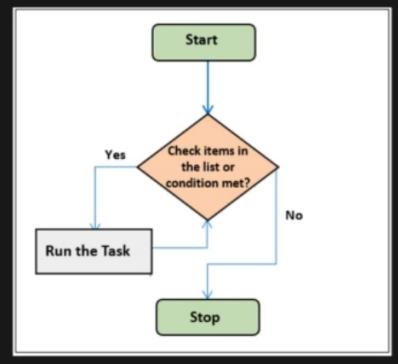
```

tasks:
  - name: Run with items greater than 5
    ansible.builtin.command: echo {{ item }}
    loop: [ 0, 2, 4, 6, 8, 10 ]
    when: item > 5

```

3.3 Loops (cicli), [1]

- Tutte le volte che abbiamo la necessità di eseguire un task più volte, possiamo utilizzare i **Loops**
- Possiamo iterare su **liste**, **liste di hash** e **dizionari**
- Quando iteriamo su una semplice lista, possiamo riferirci all'elemento x tramite la parola chiave `{{ item }}`
- Quando iteriamo su una lista di hash, possiamo riferirci alle sottochiavi attraverso le parole chiavi `{{ item.key1 }}`, `{{ item.key2 }}` ...
- Per iterare su un dizionario, possiamo utilizzare la parola chiave `dict2items` per trasformare un dizionario in un elenco, e poterlo iterare. Esempio: `dict2items`



Ansible offre le parole chiave `loop`, `with_<lookup>` e `until` per eseguire un'attività più volte.

Esempi di loop comunemente utilizzati sono la modifica della proprietà di diversi file e/o directory con il modulo file, la creazione di più utenti con il modulo user e la ripetizione di un passaggio di polling fino al raggiungimento di un determinato risultato.

3.3.1 Iterazione su una lista semplice, [1]

I task ripetitivi possono essere scritti come cicli standard su un semplice elenco di stringhe. È possibile definire l'elenco direttamente nel task.

```
- name: Add user testuser1
  ansible.builtin.user:
    name: "testuser1"
    state: present
    groups: "wheel"

- name: Add user testuser2
  ansible.builtin.user:
    name: "testuser2"
    state: present
    groups: "wheel"
```

utilizzando i loop diventa →

```
- name: Add several users
  ansible.builtin.user:
    name: "{{ item }}"
    state: present
    groups: "wheel"
  loop:
    - testuser1
    - testuser2
```

3.3.2 Iterazione su una lista di hash, [1]

Se si dispone di un elenco di hash, è possibile fare riferimento alle sottochiavi in un ciclo. Ad esempio:

```
- name: Add several users
  ansible.builtin.user:
    name: "{{ item.name }}"
    state: present
    groups: "{{ item.groups }}"
  loop:
    - { name: 'testuser1', groups: 'wheel' }
    - { name: 'testuser2', groups: 'root' }
```

3.3.3 Iterazione su un dictionary, [1]

Per eseguire un ciclo su un dict, utilizzare [dict2items](#):

```

- name: Using dict2items
  ansible.builtin.debug:
    msg: "{{ item.key }} - {{ item.value }}"
  loop: "{{ tag_data | dict2items }}"
  vars:
    tag_data:
      Environment: dev
      Application: payment

```

3.4 Blocks, [1]

- I **Blocks** vengono utilizzati per raggruppare tasks, oppure per gestire gli errori nei tasks
- Tutti i tasks in un Block ereditano le direttive applicate a livello di Block
- Puoi controllare la gestione degli errori relativi ai tasks utilizzando i Blocks con le sezioni **rescue** e **always**
- Similmente alla gestione delle eccezioni nei linguaggi di programmazione, Ansible esegue le attività indicate in **rescue** quando uno dei task precedenti nel Block è **fallito**
- Le attività nella sezione **always** vengono eseguite indipendentemente dallo stato dei tasks del Block precedente
- **ansible_failed_task** e **ansible_failed_result** sono due variabili che è possibile utilizzare con **rescue** per ottenere informazioni relative al task fallito

I blocchi creano gruppi logici di attività. I blocchi offrono anche la possibilità di gestire gli errori dei task, in modo simile alla gestione delle eccezioni in molti linguaggi di programmazione.

- Raggruppare i task con i blocks.
- Gestione degli errori con i blocks.

3.4.1 Raggruppare i task con i blocks, [1]

Tutti i task di un blocco ereditano le direttive applicate a livello di blocco. La maggior parte di ciò che si può applicare a un singolo task (con l'eccezione dei loop) può essere applicato a livello di blocco, quindi i blocchi rendono molto più semplice impostare dati o direttive comuni ai task. La direttiva non influisce sul blocco stesso, ma viene ereditata solo dai task racchiusi da un blocco. Ad esempio, un'istruzione `when` viene applicata ai task all'interno di un blocco, non al blocco stesso.

```

tasks:
  - name: Install, configure, and start Apache
    block:
      - name: Install httpd and memcached
        ansible.builtin.yum:
          name:
          - httpd
          - memcached
          state: present

      - name: Apply the foo config template
        ansible.builtin.template:
          src: templates/src.j2
          dest: /etc/foo.conf

      - name: Start service bar and enable it
        ansible.builtin.service:
          name: bar
          state: started
          enabled: True
        when: ansible_facts['distribution'] == 'CentOS'
        become: true
        become_user: root
        ignore_errors: yes

```

Nell'esempio precedente, la condizione `when` sarà valutata prima che Ansible esegua ciascuno dei tre task del blocco. Tutti e tre i task ereditano anche le direttive per l'escalation dei privilegi, venendo eseguiti come utente root. Infine, `ignore_errors: yes` assicura che Ansible continui a eseguire il playbook anche se alcuni dei task falliscono.

3.4.2 Gestione degli errori con i blocks, [1]

È possibile controllare il modo in cui Ansible risponde agli errori delle attività utilizzando blocchi con sezioni `rescue` e `always`.

I blocchi `rescue` specificano i task da eseguire quando un task precedente di un blocco fallisce. Questo approccio è simile alla gestione delle eccezioni in molti linguaggi di programmazione. Ansible esegue i blocchi di `rescue` solo dopo che un task restituisce lo stato "failed". Le definizioni sbagliate dei task e gli host irraggiungibili non attivano il blocco `rescue`.

```

tasks:
- name: Handle the error
block:
- name: Print a message
  ansible.builtin.debug:
    msg: 'I execute normally'

- name: Force a failure
  ansible.builtin.command: /bin/false

- name: Never print this
  ansible.builtin.debug:
    msg: 'I never execute, due to the above task failing, :-('
rescue:
- name: Print when errors
  ansible.builtin.debug:
    msg: 'I caught an error, can do stuff here to fix it, :-)'

```

È anche possibile aggiungere una sezione **always** a un blocco. Le attività nella sezione always vengono eseguite indipendentemente dallo stato delle attività del blocco precedente.

```

- name: Always do X
block:
- name: Print a message
  ansible.builtin.debug:
    msg: 'I execute normally'

- name: Force a failure
  ansible.builtin.command: /bin/false

- name: Never print this
  ansible.builtin.debug:
    msg: 'I never execute :-('
always:
- name: Always do this
  ansible.builtin.debug:
    msg: "This always executes, :-)"

```

3.5 Handlers, [1]

- Le operazioni associate agli **handler** si differenziano dai task poiché non vengono sempre eseguite, ma vengono eseguite solo se la configurazione di quel servizio cambia.
- Un semplice esempio è quello di avviare un servizio dopo aver installato il software necessario, o riavviarlo dopo la modifica del suo file di configurazione.
- Vengono eseguiti solo se ricevono la specifica '**notify**'
- Per impostazione predefinita, gli handler vengono eseguiti dopo che tutti i tasks in una particolare play sono state completati.

```

1 - name: example handler
2 hosts: webservers
3 tasks:
4   - name: Install nginx
5     package:
6       name: nginx
7       state: present
8       notify:
9         Start nginx
10
11 handlers:
12   - name: Start nginx
13     service:
14       name: nginx
15       state: started

```

A volte si desidera che un'attività venga eseguita solo quando viene apportata una modifica a un computer. Ad esempio, si potrebbe voler riavviare un servizio se un task aggiorna la configurazione di quel servizio, ma non se la configurazione rimane invariata. Ansible utilizza gli

handler per risolvere questo caso d'uso. Gli handler sono task che vengono eseguiti solo quando vengono notificati.

```
- name: Verify apache installation
  hosts: webservers
  vars:
    http_port: 80
    max_clients: 200
  remote_user: root
  tasks:
    - name: Ensure apache is at the latest version
      ansible.builtin.yum:
        name: httpd
        state: latest

    - name: Write the apache config file
      ansible.builtin.template:
        src: /srv/httpd.j2
        dest: /etc/httpd.conf
      notify:
        - Restart apache

    - name: Ensure apache is running
      ansible.builtin.service:
        name: httpd
        state: started

  handlers:
    - name: Restart apache
      ansible.builtin.service:
        name: httpd
        state: restarted
```

In questo esempio di playbook, il server Apache viene riavviato dal'handler al termine di tutte le attività della play.

I task possono indicare l'esecuzione di uno o più handler utilizzando la parola chiave `notify`. La parola chiave `notify` può essere applicata a un'attività e accetta un elenco di nomi di handler che vengono richiamati al cambiamento dell'attività. In alternativa, è possibile fornire anche una stringa contenente un singolo nome di handler. L'esempio seguente mostra come più handler possano essere notificati da un singolo task:

```

tasks:
- name: Template configuration file
  ansible.builtin.template:
    src: template.j2
    dest: /etc/foo.conf
  notify:
    - Restart apache
    - Restart memcached

handlers:
- name: Restart memcached
  ansible.builtin.service:
    name: memcached
    state: restarted

- name: Restart apache
  ansible.builtin.service:
    name: apache
    state: restarted

```

Nell'esempio precedente gli handler vengono eseguiti al cambiamento del task nel seguente ordine: **Restart memcached**, **Restart apache**. Gli handler vengono eseguiti nell'ordine in cui sono definiti nella sezione **handlers**, non nell'ordine elencato nell'istruzione **notify**. La notifica dello stesso handler più volte comporta l'esecuzione del handler una sola volta, indipendentemente dal numero di task che lo notificano. Ad esempio, se più task aggiornano un file di configurazione e notificano a un handler di riavviare Apache, Ansible esegue Apache una sola volta per evitare riavvii non necessari.

3.6 Tags, [1]

- Con l'utilizzo dei **Tag**, possiamo decidere di eseguire solo parti del nostro playbook, evitando un'esecuzione completa
- Utilizzando i Tag possiamo **eseguire o skippare** determinati task dal nostro playbook
- Gli step da seguire sono due:
 - Inserire il tag a livello di task*
 - Lanciare il playbook specificando quali tag eseguire o skippare*
- Ansible mette a disposizione anche due tag "speciali" **always** e **never**
- Per eseguire il playbook con i tag, possiamo utilizzare il parametro:
 - tags "tag1, tag2"** per eseguire tag1 e tag2
 - skip-tags "tag1, tag2"** per skippare l'esecuzione di tag1 e tag2

Se si dispone di un playbook di grandi dimensioni, può essere utile eseguire solo parti specifiche di esso invece di eseguire l'intero playbook. È possibile farlo con i tag di Ansible. L'uso dei tag per eseguire o escludere attività selezionate è un processo composto da due fasi:

- Aggiungere i tag ai task, singolarmente o con l'ereditarietà dei tag da un blocco, una rappresentazione, un ruolo o un'importazione.
- Selezionare o saltare i tag quando si esegue il playbook.

3.6.1 Aggiunta di tag con la parola chiave tags, [1]

È possibile aggiungere tag ad un singolo task. È anche possibile aggiungere tag a più task definendoli a livello di block, play, role o import. La parola chiave tags risponde a tutti questi casi d'uso. È possibile selezionare o saltare i task in base ai tag attraverso la riga di comando quando si esegue un playbook.

Aggiunta di tag ai singoli task.

Al livello più semplice, è possibile applicare uno o più tag a un singolo task. È possibile aggiungere tag ai task nei playbook, nei file task o all'interno di un ruolo. Ecco un esempio che assegna a due task tag diversi:

```
tasks:  
- name: Install the servers  
  ansible.builtin.yum:  
    name:  
    - httpd  
    - memcached  
    state: present  
tags:  
- packages  
- webservers  
  
- name: Configure the service  
  ansible.builtin.template:  
    src: templates/src.j2  
    dest: /etc/foo.conf  
tags:  
- configuration
```

È possibile applicare lo stesso tag a più di un singolo task. Questo esempio assegna a diversi task lo stesso tag, "ntp":

```

---
# file: roles/common/tasks/main.yml

- name: Install ntp
  ansible.builtin.yum:
    name: ntp
    state: present
  tags: ntp

- name: Configure ntp
  ansible.builtin.template:
    src: ntp.conf.j2
    dest: /etc/ntp.conf
  notify:
    - restart ntpd
  tags: ntp

- name: Enable and run ntpd
  ansible.builtin.service:
    name: ntpd
    state: started
    enabled: yes
  tags: ntp

- name: Install NFS utils
  ansible.builtin.yum:
    name:
      - nfs-utils
      - nfs-util-lib
    state: present
  tags: filesharing

```

3.6.2 Tag speciali: always e never, [1]

Ansible riserva due nomi di tag per dei comportamenti speciali: **always** e **never**. Se si assegna il tag **always** a un task o a un play, Ansible eseguirà sempre quel task o quel play, a meno che non lo si salti specificamente (tramite **--skip-tags always**). Per esempio:

```

tasks:
- name: Print a message
  ansible.builtin.debug:
    msg: "Always runs"
  tags:
    - always

- name: Print a message
  ansible.builtin.debug:
    msg: "runs when you use tag1"
  tags:
    - tag1

```

Se si assegna il tag **never** a un task o ad un play, Ansible lo salterà a meno che non lo si richieda espressamente (tramite **--tags never**). Per esempio:

```

tasks:
  - name: Run the rarely-used debug task
    ansible.builtin.debug:
      msg: '{{ showmevar }}'
      tags: [ never, debug ]

```

Il task di debug, utilizzato raramente nell'esempio precedente, viene eseguito solo quando si richiede specificamente il tag debug o never.

3.6.3 Selezionare o ignorare i tag durante l'esecuzione di una playbook, [1]

Una volta aggiunti i tag ai task, agli include, ai blocks, ai play, ai roles e agli imports, è possibile eseguire o saltare selettivamente i task in base ai loro tag quando si esegue ansible-playbook. Ansible esegue o salta tutti i task con tag che corrispondono a quelli passati attraverso la riga di comando.

Ansible-playbook offre cinque opzioni da riga di comando relative ai tag:

- **--tags all** - Esegue tutti i task, ignorando i tag (default behavior).
- **--tags [tag1, tag2]** - Esegue solo i task con il tag1 o il tag2.
- **--skip-tags [tag3, tag4]** - Esegue tutti i task tranne quelli con il tag3 o il tag4.
- **--tags tagged** - Esegue solo i task con almeno un tag.
- **--tags untagged** - Esegue solo i task senza tag.

Ad esempio, per eseguire solo i task e i blocchi etichettati come **configuration** e **packages** in un playbook molto lungo:

```
ansible-playbook example.yml --tags "configuration,packages"
```

Per eseguire tutti i compiti tranne quelli contrassegnati con **packages**:

```
ansible-playbook example.yml --skip-tags "packages"
```

Quando si esegue un ruolo o un playbook, è possibile che non si sappia o non si ricordi quali task hanno quali tag, o quali tag esistono proprio. Ansible offre due flag da riga di comando per ansible-playbook che aiutano a gestire i playbook con tag:

- **--list-tags** - Genera una lista di tag disponibili.
- **--list-tasks** - Quando viene usato con **--tags tagname** o **--skip-tags tagname**, genera un'anteprima dei task taggati.

Ad esempio, se non si sa se il tag per le task di configurazione è **config** o **conf** in un file di playbook, role o task, è possibile visualizzare tutti i tag disponibili senza eseguire alcuna task:

```
ansible-playbook example.yml --list-tags
```

Se non si sa quali task hanno i tag **configuration** e **packages**, si possono passare questi tag e aggiungere **--list-tasks**. Ansible elenca i task, ma non ne esegue nessuno.

```
ansible-playbook example.yml --tags "configuration,packages" --list-tasks
```

3.7 Utilizzo delle variabili, [1]

- Per creare una variabile semplice, basta seguire la sintassi YAML `key: value`
- Un esempio di variabile: `config_path: /opt/my_app/config`
- Per fare riferimento alla variabile possiamo usare la sintassi **Jinja2** ovvero usando le doppie graffe.
Es: `{{config_path}}/file.cfg`
- E' possibile anche "registrazione delle variabili", ovvero memorizzare il risultato di un task in una var ed utilizzarla in task successivi
- Per farlo dobbiamo utilizzare la parola chiave `register`
- Le variabili possono essere inserite in:
`Inventory`
`Playbook` (usando la parola chiave `vars`)
`Command Line` (usando l'opzione `-extra-vars`)
`File di variabili esterni` (usando la parola chiave `vars_files`)

Ansible utilizza le variabili per gestire le differenze tra i dispositivi. Con Ansible è possibile eseguire task e playbook su diversi dispositivi mediante un unico comando. Per rappresentare le varianti tra i dispositivi, è possibile creare variabili con la sintassi YAML standard, compresi gli elenchi e i dizionari. Queste variabili possono essere definite nei playbook, nell'inventory, in file o roles esterni o attraverso riga di comando. È anche possibile creare variabili durante l'esecuzione di una playbook, registrando il valore o i valori di ritorno di un'attività come una nuova variabile.

Dopo aver creato le variabili, definendole in un file, passandole alla riga di comando o registrando il valore o i valori di ritorno di un task come una nuova variabile, è possibile utilizzarle negli argomenti dei moduli, nelle dichiarazioni condizionali "when", nei modelli e nei cicli.

È possibile definire una variabile semplicemente utilizzando la sintassi YAML standard. Ad esempio:

```
remote_install_path: /opt/my_app_config
```

Dopo aver definito una variabile, si utilizza la sintassi di Jinja2 per farvi riferimento. Le variabili Jinja2 utilizzano doppie parentesi graffe. È possibile utilizzare la sintassi di Jinja2 nei playbook. Ad esempio:

```
ansible.builtin.template:  
  src: foo.cfg.j2  
  dest: '{{ remote_install_path }}/foo.cfg'
```

In questo esempio, la variabile definisce la posizione di un file, che può variare da un sistema all'altro.

È possibile definire variabili con più valori utilizzando elenchi YAML. Ad esempio:

```
region:  
  - northeast  
  - southeast  
  - midwest
```

Il primo elemento di un elenco è l'elemento 0, il secondo elemento è l'elemento 1. Ad esempio:

```
region: "{{ region[0] }}"
```

Il valore di questa espressione sarebbe "northeast".

Un dizionario memorizza i dati in coppie chiave-valore. Di solito, i dizionari vengono utilizzati per memorizzare dati correlati, come le informazioni contenute in un ID o in un profilo utente.

Ad esempio:

```
foo:  
  field1: one  
  field2: two
```

Per accedervi è possibile usare la dot notation o le parentesi quadre:

```
foo['field1']  
foo.field1
```

Entrambi gli esempi fanno riferimento allo stesso valore ("one"). La notazione a parentesi quadre funziona sempre, mentre la dot notation, delle volte, può causare problemi.

3.7.1 Registrare le variabili, [1]

È possibile creare variabili dall'output di un task di Ansible con la parola chiave **register**. Ad esempio:

```
- hosts: web_servers  
  
tasks:  
  
  - name: Run a shell command and register its output as a variable  
    ansible.builtin.shell: /usr/bin/foo  
    register: foo_result  
    ignore_errors: true  
  
  - name: Run a shell command using output of the previous task  
    ansible.builtin.shell: /usr/bin/bar  
    when: foo_result.rc == 5
```

3.7.2 Definizione delle variabili in runtime (attraverso la command line), [1]

È possibile definire le variabili durante l'esecuzione del playbook passando le variabili alla riga di comando con l'argomento **--extra-vars** (o **-e**). Quando si passano le variabili alla riga di comando, si deve usare una singola stringa quotata, contenente una o più variabili, in uno dei formati seguenti:

Formato chiave=valore

I valori passati con la sintassi key=value sono interpretati come stringhe. Bisogna utilizzare il formato JSON se si devono passare valori non stringhe, come booleani, interi, float, liste e così via.

```
ansible-playbook release.yml --extra-vars "version=1.23.45  
other_variable=foo"
```

Formato stringa JSON

```
ansible-playbook release.yml --extra-vars '{"version": "1.23.45", "other_variable": "foo"}'  
ansible-playbook arcade.yml --extra-vars '{"pacman": "mrs", "ghosts": ["inky", "pinky", "clyde", "sue"]}'
```

3.7.3 Input interattivo: prompts, [1]

Se si desidera il playbook richieda all'utente determinati input, basta aggiungere una sezione **"vars_prompt"**. La richiesta di variabili all'utente consente di evitare la registrazione di dati sensibili come le password. Oltre alla sicurezza, i prompt favoriscono la flessibilità. Ad esempio, se si usa una playbook per più versioni del software, si può richiedere la versione di una particolare release. Ad esempio:

```

---
- hosts: all
  vars_prompt:

    - name: username
      prompt: What is your username?
      private: no

    - name: password
      prompt: What is your password?

  tasks:

    - name: Print a message
      ansible.builtin.debug:
        msg: 'Logging in as {{ username }}'

```

Per impostazione predefinita l'input dell'utente risulta nascosto, ma può essere reso visibile impostando `private: no`.

Inoltre, se si dispone di una variabile che cambia di rado, è possibile fornire un valore predefinito che può essere sovrascritto.

`vars_prompt:`

```

- name: release_version
  prompt: Product release version
  default: "1.0"

```

3.8 Ansible Vault, [1]

- Permette di memorizzare dati sensibili come password all'interno di file crittografati
- La crittografia può essere fatta a livello di file o di variabile
- `ansible-vault create example.yml`
- La cifratura predefinita è `AES256`
- Altri comandi sono: `edit`, `rekey`, `encrypt`, `decrypt`, `view`
- `ansible-vault encrypt_string 'example' --name 'the_secret'` creerà una variabile criptata
- Utilizzando il tag `!vault` Ansible capirà che quella variabile dovrà essere decriptata
- `--ask-vault-pass` o `--vault-password-file` sono due modi per fornire la pwd al vault
- `$ANSIBLE_VAULT;1.1;AES256` sarà l'intestazione di un file criptato

Ansible Vault critta le variabili e i file in modo da poter proteggere i contenuti sensibili, come le password o le chiavi, anziché lasciarli visibili in chiaro nei playbook o nei roles. Per utilizzare Ansible Vault sono necessarie una o più password per criptare e decriptare il contenuto.

Utilizzate le password con lo strumento a riga di comando `ansible-vault` per creare e visualizzare variabili crittografate, creare file crittografati, crittografare file esistenti o modificare, ricodificare o decrittografare i file. È quindi possibile mettere il contenuto crittografato sotto controllo sorgente e condividerlo in modo più sicuro.

È possibile utilizzare variabili e file crittografati in comandi e playbook ad hoc, fornendo le password utilizzate per crittografarli. È possibile modificare il file ansible.cfg per specificare la posizione di un file di password o per richiedere sempre la password.

Ogni volta che si critta una variabile o un file con Ansible Vault, è necessario fornire una password. Quando si utilizza una variabile o un file crittografato in un comando o in un playbook, è necessario fornire la stessa password utilizzata per la crittografia.

3.8.1 Criptare le variabili, [1]

È possibile criptare singoli valori all'interno di un file YAML utilizzando il comando `ansible-vault encrypt_string`.

Vantaggi e svantaggi della crittografia delle variabili:

Con la crittografia a livello di variabile, i file sono ancora facilmente leggibili. È possibile mescolare variabili in chiaro e crittografate. Tuttavia, la rotazione delle password non è così semplice come nel caso della crittografia a livello di file. Non è possibile ricodificare le variabili crittografate. Inoltre, la crittografia a livello di variabile funziona solo sulle variabili. Se si desidera criptare attività o altri contenuti, è necessario criptare l'intero file.

Creare variabili criptate:

Il comando `ansible-vault encrypt_string` critta e formatta qualsiasi stringa digitata (o copiata o generata) in un formato che può essere incluso in un playbook, un ruolo o un file di variabili. Per creare una variabile crittografata di base, si passano tre opzioni al comando `ansible-vault encrypt_string`:

- un'origine per la password del vault (prompt, file o script, con o senza un ID del vault).
- la stringa da criptare.
- il nome della stringa (il nome della variabile).

Lo schema si presenta così:

```
ansible-vault encrypt_string <password_source> '<string_to_encrypt>' --name '<string_name_of_variable>'
```

Ad esempio, per crittografare la stringa "foobar" utilizzando l'unica password memorizzata in "a_password_file" e nominare la variabile "the_secret":

```
ansible-vault encrypt_string --vault-password-file a_password_file 'foobar' --name 'the_secret'
```

Il comando precedente crea questo contenuto:

```
the_secret: !vault |
    $ANSIBLE_VAULT;1.1;AES256
    62313365396662343061393464336163383764373764613633653634306231386433626436623361
    6134333665353966363534333632666535333761666131620a663537646436643839616531643561
    63396265333966386166373632626539326166353965363262633030333630313338646335303630
    3438626666666137650a353638643435666633633964366338633066623234616432373231333331
    6564
```

Per crittografare la stringa "fooodev", aggiungere l'etichetta ID del vault "dev" con la password del vault "dev" memorizzata in "a_password_file" e chiamare la variabile crittografata "the_dev_secret":

```
ansible-vault encrypt_string --vault-id dev@a_password_file 'fooodev' --name 'the_dev_secret'
```

Il comando precedente crea questo contenuto:

```
the_dev_secret: !vault |
$ANSIBLE_VAULT;1.2;AES256;dev
30613233633461343837653833666333643061636561303338373661313838333565653635353162
3263363434623733343538653462613064333634333464660a6636336239393439316636633863
61636237636537333938306331383339353265363239643939666639386530626330633337633833
6664656334373166630a363736393262666465663432613932613036303963343263623137386239
6330
```

3.8.2 Crittografare i file con Ansible Vault, [1]

3.9 I comandi ad hoc, [1]

- Utilizzando la riga di comando Ansible è possibile lanciare comandi ad-hoc per automatizzare un'attività su una singola macchina o su più nodi
- Sono comandi semplici e rapidi, ma non sono riutilizzabili. Sono ottimi per attività che ripetono raramente, senza la necessità di scrivere un playbook
- Un comando ad-hoc segue questa sintassi:
`ansible [pattern] -m [module] -a "[module options]"`
- Tra i moduli più utilizzati troviamo: il modulo `ping`, `setup`, `copy`, `yum`, `shell`, `service` ecc..
- Per effettuare la '*Privilege Escalation*' possiamo aggiungere l'opzione `-become`
- Un esempio reale utilizzando il modulo `shell`:
`ansible db -m shell -a "uptime"`

Un comando ad hoc di Ansible utilizza lo strumento a riga di comando /usr/bin/ansible per automatizzare una singola attività su uno o più nodi gestiti. I comandi ad hoc sono facili e veloci, ma non sono riutilizzabili. I comandi ad hoc dimostrano la semplicità e la potenza di Ansible.

Perché utilizzare comandi ad hoc?

I comandi ad hoc sono ottimi per i task che si ripetono raramente. Ad esempio, se si desidera spegnere tutte le macchine del laboratorio per le vacanze di Natale, si può eseguire un rapido comando ad hoc in Ansible senza scrivere un playbook. Un comando ad hoc si presenta così:

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

Per esempio, modulo `shell`:

```
$ ansible db -m shell -a 'uptime'
```

Con questa riga, Ansible esegue il comando `uptime` su tutte le macchine presenti in 'db'.

Altro esempio, module `user`:

```
$ ansible all -m user -a 'name=testUser' -become
```

Con questa riga, Ansible andrà a creare su tutte le macchine presenti nell'inventory (all) un nuovo utente denominato 'testUser'. Infine `-become` permette il privilege escalation.

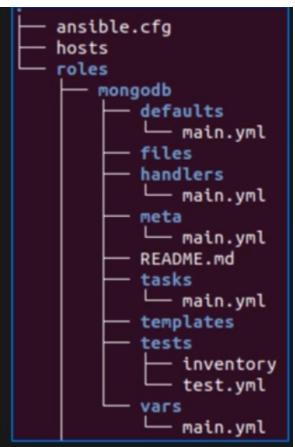
Altro esempio, modulo `copy`:

```
$ ansible all -m copy -a 'src=~/fileTest dest=~/'
```

Con questa riga, Ansible andrà a copiare su tutte le macchine presenti nell'inventory (all) il file situato nel percorso `src=~/fileTest`.

3.10 I ruoli, [1]

- I ruoli sono indipendenti l'uno dall'altro e quindi possono essere condivisi con altri utenti
- Permettono di creare playbook molto semplici e facili da leggere
 - **Tasks** : contiene l'elenco principale dei tasks che devono essere eseguite dal ruolo. Esso contiene il file **main.yml** per quel particolare ruolo.
 - **Handlers** : contiene handlers che possono essere utilizzati da questo ruolo o anche in qualsiasi altro luogo al di fuori di questo ruolo.
 - **Defaults** : contiene le variabili predefinite che verranno utilizzate da questo ruolo.
 - **Vars** : questa directory è composta da altre variabili che verranno utilizzate dal ruolo. Queste variabili possono essere definite nel tuo playbook.
 - **Files** : contiene i file che possono essere distribuiti da questo ruolo. Contiene file che devono essere inviati agli host durante la configurazione del ruolo.
 - **Meta** : definisce i metadati per questo ruolo. Fondamentalmente, contiene file che stabiliscono le dipendenze dei ruoli.
- Per creare un nuovo ruolo: **ansible-galaxy init test-role**



I ruoli consentono di caricare automaticamente le vars, i files, i task, i handler e gli altri oggetti di Ansible in base a una struttura di file nota. Dopo aver raggruppato i contenuti in ruoli, è possibile riutilizzarli facilmente e condividerli con altri utenti.

Un ruolo Ansible ha una struttura di directory definita con otto directory standard principali. È necessario includere almeno una di queste directory in ogni ruolo. È possibile omettere qualsiasi directory che il ruolo non utilizza. Ad esempio:

```
# playbooks
site.yml
webservers.yml
fooservers.yml
```

```
roles/
  common/
    tasks/          # this hierarchy represents a "role"
      main.yml     # <- tasks file can include smaller files if warranted
    handlers/
      main.yml     # <- handlers file
    templates/
      ntp.conf.j2  # <----- templates end in .j2
    files/
      bar.txt      # <- files for use with the copy resource
      foo.sh       # <- script files for use with the script resource
    vars/
      main.yml     # <- variables associated with this role
    defaults/
      main.yml     # <- default lower priority variables for this role
    meta/
      main.yml     # <- role dependencies
    library/
      module_utils/ # roles can also include custom modules
      lookup_plugins/ # roles can also include custom module_utils
                      # or other types of plugins, like lookup in this case

  webtier/          # same kind of structure as "common" was above, done for the webtier role
  monitoring/       # ""
  fooapp/           # ""
```

Per impostazione predefinita, Ansible cercherà in ogni directory all'interno di un ruolo un file **main.yml** per i contenuti rilevanti (oppure anche **main.yaml** e **main**):

- **tasks/main.yml** - l'elenco principale dei tasks che il ruolo esegue.
- **handlers/main.yml** - handlers, che possono essere usati all'interno o all'esterno di questo ruolo.

- `library/my_module.py` - moduli che possono essere utilizzati in questo ruolo.
- `defaults/main.yml` - variabili predefinite per il ruolo. Queste variabili hanno la priorità più bassa tra quelle disponibili e possono essere facilmente sovrascritte da qualsiasi altra variabile, comprese quelle dell'inventory
- `vars/main.yml` - altre variabili per il ruolo.
- `files/main.yml` - file che il ruolo distribuisce.
- `templates/main.yml` - modelli che il ruolo distribuisce.
- `meta/main.yml` - metadati per il ruolo, comprese le dipendenze del ruolo.

È possibile aggiungere altri file YAML in alcune directory. Ad esempio, è possibile inserire i task specifici della piattaforma in file separati e fare riferimento ad essi nel file `tasks/main.yml`:

```
# roles/example/tasks/main.yml
- name: Install the correct web server for RHEL
  import_tasks: redhat.yml
  when: ansible_facts['os_family']|lower == 'redhat'

- name: Install the correct web server for Debian
  import_tasks: debian.yml
  when: ansible_facts['os_family']|lower == 'debian'

# roles/example/tasks/redhat.yml
- name: Install web server
  ansible.builtin.yum:
    name: "httpd"
    state: present

# roles/example/tasks/debian.yml
- name: Install web server
  ansible.builtin.apt:
    name: "apache2"
    state: present
```

Il modo classico (originale) di usare i ruoli è con l'opzione `roles` per un certo play:

```
---
- hosts: webservers
  roles:
    - common
    - webservers
```

3.10.1 Ansible Galaxy, [1]

- E' un componente indipendente, che ci permette di riutilizzare dei passaggi di configurazione comuni. Ad esempio tutti gli step per installare un db mysql
- E' un repository per i ruoli Ansible disponibili per essere inseriti direttamente nei tuoi playbook
- Per installare un ruolo, possiamo utilizzare il comando:
`ansible-galaxy install username.rolename`
- Il ruolo verrà scaricato in `~/.ansible/roles/username.rolename` dell'utente corrente
- Altri comandi utili:
`ansible-galaxy list` → mostra la lista di tutti i ruoli installati
`ansible-galaxy remove [role]` → rimuove uno specifico ruolo
`ansible-galaxy init` → utilizzato per creare il tuo ruolo

Ansible Galaxy è un sito web gratuito per trovare, scaricare e condividere i ruoli sviluppati dalla community.

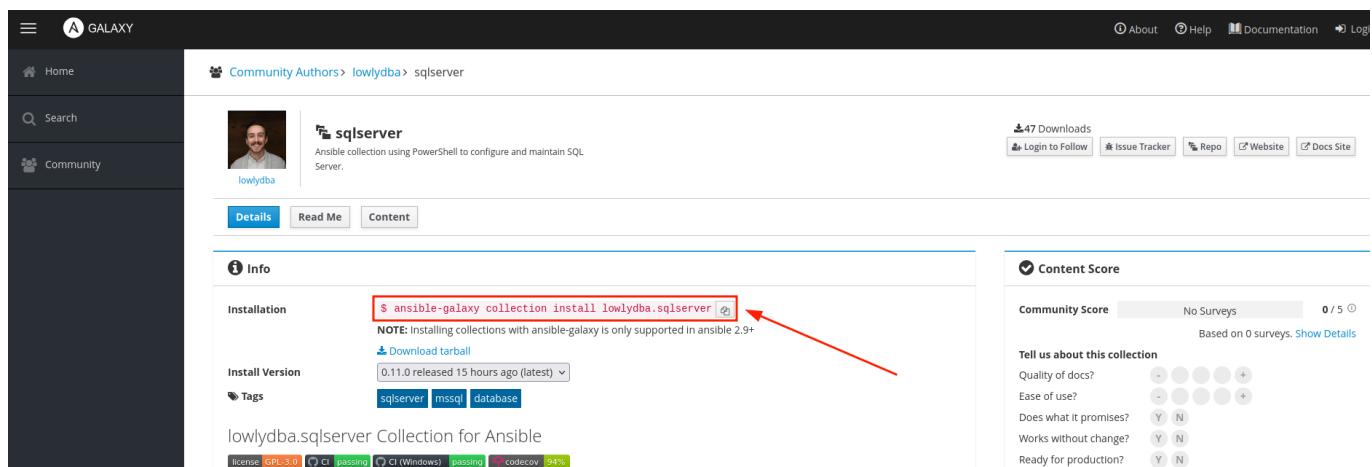
Galaxy fornisce unità di lavoro preconfezionate come i ruoli e le collezioni. È possibile trovare ruoli per il provisioning dell'infrastruttura, la distribuzione delle applicazioni e tutte le attività che si svolgono quotidianamente. Il formato di raccolta fornisce un pacchetto completo di automazione che può includere più playbook, ruoli, moduli e plugin.

Per installare una raccolta archiviata in Galaxy:

```
ansible-galaxy collection install my_namespace.my_collection
```

Per aggiornare:

```
ansible-galaxy collection install my_namespace.my_collection --upgrade
```



The screenshot shows the Ansible Galaxy interface. On the left is a sidebar with 'Home', 'Search', and 'Community' sections. The main area displays the 'sqlserver' collection by 'lowlydba'. The collection page includes a profile picture of the author, a brief description, download statistics (47 Downloads), and links to GitHub, issue tracker, website, and documentation. Below this, there are tabs for 'Details', 'Read Me', and 'Content'. The 'Details' tab is active, showing the 'Info' section which contains the installation command: '\$ ansible-galaxy collection install lowlydba.sqlserver'. A red arrow points to this command. Other details shown include the collection's purpose (Ansible collection using PowerShell to configure and maintain SQL Server), its version (0.11.0 released 15 hours ago), and tags (sqlserver, mysql, database). The right side of the page features a 'Content Score' section with a 'Community Score' of 0/5 based on 0 surveys, and a 'Tell us about this collection' section with five rating scales for various criteria like quality of docs, ease of use, and readiness for production.

4. Docker, [1], [play-with-docker], [katakoda]

4.0 Cheatsheet

Parte 1:

1. Containers

A lightweight virtual OS that run processes in full isolation.

1.1 Lifecycle

- `docker create` creates a container but does not start it.
- `docker rename` allows the container to be renamed.
- `docker run` creates and starts a container in one operation.
- `docker rm` deletes a container.
- `docker update` updates a container's resource limits.
 - `docker run --rm` : remove the container after it stops.
 - `docker run -v $HOSTDIR:$DOCKERDIR`: map the directory (\$HOSTDIR) on the host to a docker container (\$DOCKERDIR).
 - `docker rm -v`: remove the volumes associated with the container.
 - `docker run --log-driver=syslog` : run docker with a custom log driver.

1.2 Starting and Stopping

- `docker start` starts a container so it is running.
- `docker stop` stops a running container.
- `docker restart` stops and starts a container.
- `docker pause` pauses a running container, "freezing" it in place.
- `docker unpause` will unpause a running container.
- `docker wait` blocks until running container stops.
- `docker kill` sends a SIGKILL to a running container.
- `docker attach` will connect to a running container.

1.3 CPU Constraints

CPU can be limited either using a percentage over all CPUs, or by using specific cores.

- c or `cpu-shares`: 1024 means 100% of the CPU, so if we want the container to take 50% of all CPU cores, we should specify 512 for instance, `docker run -ti --c 512 ...cpuset-cpus`
- : use only some CPU cores, for instance, `docker run -ti --cpuset-cpus=0,4,6 ...`

1.4 Memory Constraints

Memory can be limited using `-m` flag, for instance, `docker run -it -m 300M ubuntu:14.04 /bin/bash`

1.5 Capabilities

- `cap-add` and `cap-drop`: Add or drop linux capabilities.
- Mount a FUSE based filesystem:
 - `docker run --rm -it --cap-add SYS_ADMIN --device /dev/fuse sshfs`
 - Give access to a single device:
 - `docker run -it --device=/dev/ttyUSB0 debian bash`

➢ Give access to all devices:

- `docker run -it --privileged -v /dev/bus/usb:/dev/bus/usb debian bash`

1.6 Info

- `docker ps` shows running containers.
- `docker logs` gets logs from container. (You can use a custom log driver, but logs is only available for json-file and journald in 1.10).
- `docker inspect` looks at all the info on a container (including IP address).
- `docker events` gets events from container.
- `docker port` shows public facing port of container.
- `docker top` shows running processes in container.
- `docker stats` shows containers' resource usage statistics.
- `docker diff` shows changed files in the container's FS.
- `docker ps -a` shows running and stopped containers

1.7 Import / Export

- `docker cp` copies files or folders between a container and the local filesystem.
- `docker export` turns container filesystem into tarball archive stream to STDOUT.

1.8 Executing Commands

`docker exec` to execute a command in container.

2. Images

A template or blueprint for docker containers.

2.1 Lifecycle

- `docker images` shows all images.
- `docker import` creates an image from a tarball.
- `docker build` creates image from Dockerfile.
- `docker commit` creates image from a container, pausing it temporarily if it is running.
- `docker rmi` removes an image.
- `docker load` loads an image from a tar archive as STDIN, including images and tags (as of 0.7).
- `docker save` saves an image to a tar archive stream to STDOUT with all parent layers, tags & versions (as of 0.7).

Parte 2:

2.2. Info

- docker history shows history of image.
- docker tag tags an image to a name (local or registry).

2.3. Cleaning up

- docker rmi remove specific images.
- docker-gc a tool to clean up images that are no longer used by any containers in a safe manner.

2.4. Load/Save image

- docker load < my_image.tar.gz load an image from file
- docker save my_image:my_tag | gzip > my_image.tar.gz save an existing image

2.5. Import/Export container

- cat my_container.tar.gz | docker import - my_image:my_tag import a container as an image from file
- docker export my_container | gzip > my_container.tar.gz export an existing container

3. Networks

A small def goes here

3.1. Lifecycle

- docker network create
- docker network rm

3.2. Info

- docker network ls
- docker network inspect

3.3. Connection

- docker network connect
- docker network disconnect

4. Registry & Repository

A repository is a hosted collection of tagged images that together create the file system for a container.

A registry is a host -- a server that stores repositories and provides an HTTP API for managing the uploading and downloading of repositories.

Docker.com hosts its own index to a central registry which contains a large number of repositories.

- docker login to login to a registry.
- docker logout to logout from a registry.
- docker search searches registry for image.
- docker pull pulls an image from registry to local machine.
- docker push pushes an image to the registry from local machine.

5. Volumes

Docker volumes are free-floating filesystems. They don't have to be connected to a particular container. You should use volumes mounted from data-only containers for portability.

5.1. Lifecycle

- docker volume create
- docker volume rm

5.2. Info

- docker volume ls
- docker volume inspect

6. Exposing ports

- docker run -p 127.0.0.1:\$HOSTPORT:\$CONTAINER-PORT --name CONTAINER -t docker_image mapping the container port to the host port using -p
- EXPOSE <CONTAINERPORT> expose port CONTAINERPORT at runtime (see dockerfile)
- docker port CONTAINER \$CONTAINERPORT check the mapped port

7. Tips

7.1. Get IP address

- > docker inspect some_docker_id | grep IPAddress | cut -d '"' -f 4
or install jq:
> docker inspect some_docker_id | jq -r '.[0].NetworkSettings.IPAddress'
- or using a go template:
> docker inspect -f '{{ .NetworkSettings.IPAddress }}' <container_name>

7.2. Get port mapping

- ```
docker inspect -f '${{range $p, $conf := .NetworkSettings.Ports}} {{$p}} -> ${{($conf | .HostPort)}} {{end}}' <containername>
```

### 7.3. Find containers by regular expression

- ```
for i in $(docker ps -a | grep "REGEXP_PATTERN" | cut -f1 -d" "); do echo $i; done
```

7.4. Get Environment Settings

- ```
docker run --rm ubuntu env
```

### 7.5. Kill running containers

- ```
docker kill $(docker ps -q)
```

7.6. Delete old containers

- ```
docker ps -a | grep 'weeks ago' | awk '{print $1}' | xargs docker rm
```

### 7.7. Delete stopped containers

- ```
docker rm -v $(docker ps -a -q -f status=exited)
```

7.8. Delete dangling images

- ```
docker rmi $(docker images -q -f dangling=true)
```

### 7.9. Delete all images

- ```
docker rmi $(docker images -q)
```

7.10. Delete dangling volumes

- ```
docker volume rm $(docker volume ls -q -f dangling=true)
```

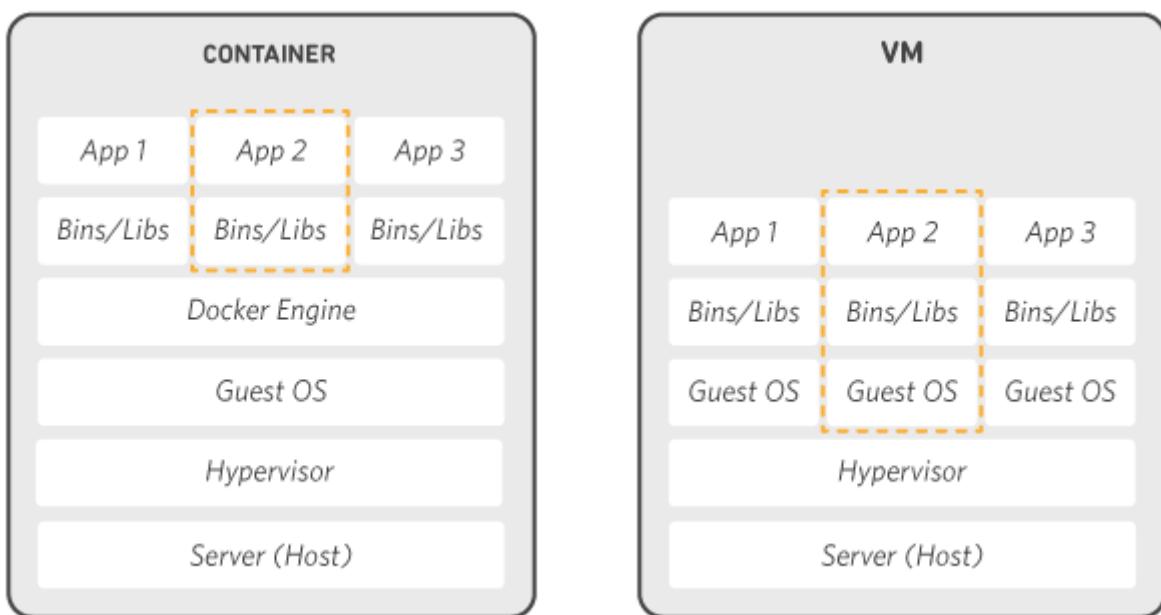
## 4.1 Introduzione

Docker è una piattaforma software che permette di creare, testare e distribuire applicazioni con la massima rapidità. Docker raccoglie il software in unità standardizzate chiamate container che offrono tutto il necessario per la loro corretta esecuzione, incluse librerie, strumenti di sistema, codice e runtime. Con Docker, è possibile distribuire e ricalibrare le risorse per un'applicazione in qualsiasi ambiente, tenendo sempre sotto controllo il codice eseguito.

L'esecuzione di Docker in AWS offre a sviluppatori e amministratori un modo altamente affidabile e poco costoso per creare, spedire ed eseguire applicazioni distribuite su qualsiasi scala.

### Confronto tra contenitori Docker e macchine virtuali:

- I **contenitori** includono l'applicazione e tutte le relative dipendenze. Condividono tuttavia il kernel del sistema operativo con altri contenitori, in esecuzione come processi isolati nello spazio utente nel sistema operativo host. Tranne che nei contenitori Hyper-V, in cui ogni contenitore viene eseguito all'interno di una macchina virtuale speciale per contenitore.
- Le **macchine virtuali** includono l'applicazione, le librerie o i file binari necessari e un sistema operativo guest completo. La virtualizzazione completa richiede più risorse rispetto alla creazione di contenitori.



Per le macchine virtuali, sono disponibili tre livelli di base nel server host, dal basso verso l'alto: infrastruttura, sistema operativo host e un hypervisor. Ogni macchina virtuale ha inoltre un proprio sistema operativo e tutte le librerie necessarie. Per Docker, il server host prevede solo l'infrastruttura e il sistema operativo, ma anche il motore del contenitore, che mantiene il contenitore isolato, ma che condivide i servizi di base del sistema operativo.

Poiché i contenitori richiedono un numero molto ridotto di risorse, ad esempio non necessitano di un sistema operativo completo, sono facili da distribuire e si avviano rapidamente. In questo modo è possibile ottenere un aumento della densità, vale a dire che è possibile eseguire più servizi nella stessa unità hardware, riducendo i costi.

Come effetto collaterale dell'esecuzione sullo stesso kernel, si ottiene un isolamento minore rispetto alle macchine virtuali.

L'obiettivo principale di un'immagine è rendere l'ambiente (dipendenze) uguale su distribuzioni diverse. Questo significa che è possibile eseguirne il debug nel computer corrente e quindi

distribuirla in un altro computer con lo stesso ambiente garantito.

Un'immagine del contenitore consente di creare un pacchetto di app o servizi e distribuirlo in modo affidabile e riproducibile. Si potrebbe dire che Docker non è solo una tecnologia, ma anche una filosofia e un processo.

Quando si usa Docker, non si sente dire agli sviluppatori: "Funziona nel computer, perché non in produzione?" Possono semplicemente dire: "Viene eseguito su Docker", perché l'applicazione Docker in pacchetto può essere eseguita in qualsiasi ambiente Docker supportato ed esegue il modo in cui è stato progettato per tutte le destinazioni di distribuzione , ad esempio Dev, QA, staging e produzione. [Una semplice analogia](#).

Un container è quindi un ambiente isolato, ed è proprio l'isolamento e la sicurezza che ne derivano che ci permettono di eseguire più container simultaneamente all'interno di un host.

I container sono leggeri in quanto non necessitano della presenza di un hypervisor come nel caso delle macchine virtuali. Si eseguono infatti direttamente all'interno del kernel della macchina host.

## 4.2 Concetti base

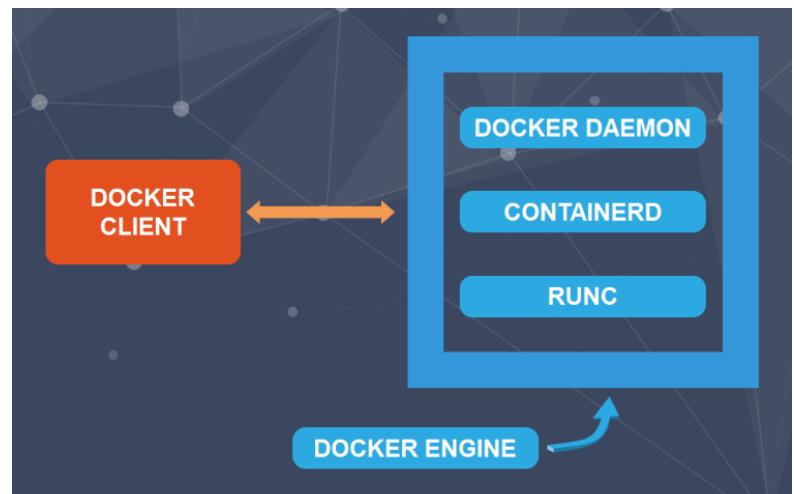
### Componenti principali:

Ad installazione completata saranno presenti i seguenti componenti:

- **Docker Client.**
- **Docker Daemon.**
- **Immagini.** Possiamo visualizzare le immagini Docker come delle entità contenenti il filesystem del SO ed una certa applicazione. Per fare un paragone, pensiamo ad un template di una VM oppure ad una classe Java.
- **Container.** Un container è un'istanza in esecuzione di una certa immagine Docker. Facendo riferimento al mondo della virtualizzazione è una VM che è stata avviata a partire da un template. A partire da una singola immagine possiamo avviare più container.
- **Dockerfile.** E' il file che descrive l'applicazione e fornisce a Docker le istruzioni su come l'applicazione deve essere definita all'interno dell'immagine.

### Docker Engine:

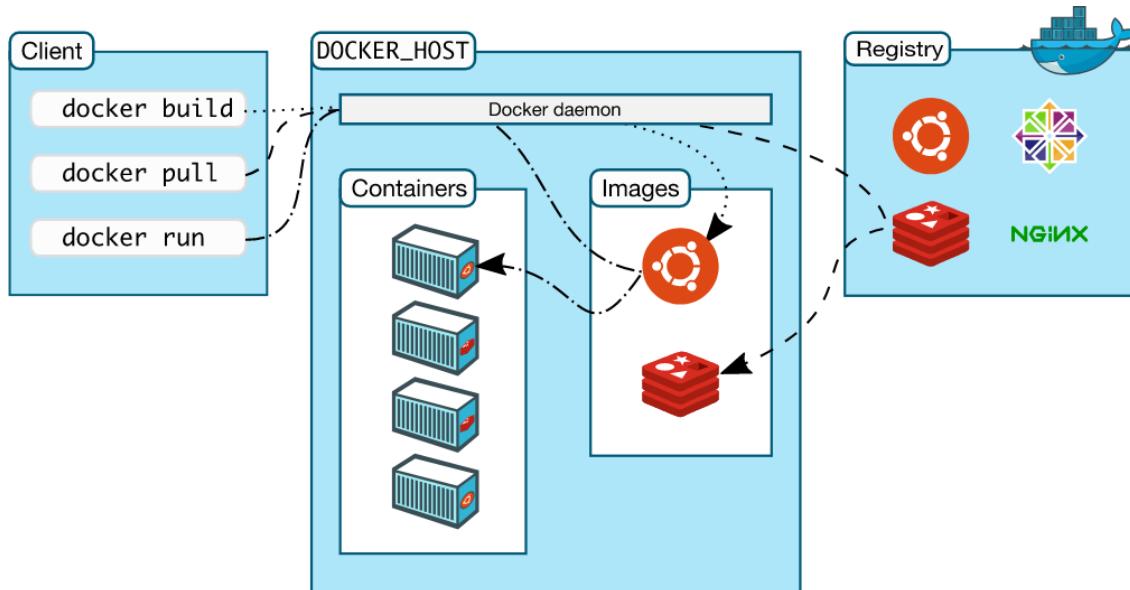
- Il docker engine è strutturato secondo una logica modulare basata sullo standard fornito dall'oci (open container initiative).
- I componenti principali del docker engine sono:
  1. Docker client.
  2. Docker daemon.
  3. Containerd.
  4. runc.
- Come interagiscono questi componenti?
  1. Dalla Docker CLI si esegue il comando di start del container.
  2. Il Docker Client converte le istruzioni in un formato adeguato per il Docker Daemon.



3. Quando il Docker Daemon riceve i comandi di creazione del container, viene a sua volta chiamato il componente “containerd”.
4. Containerd non può creare direttamente il container e utilizza il componente “runc”.
5. Runc si interfaccia con il kernel del SO e riunisce tutto ciò che è necessario per la creazione del container (namespace, cgroups, ecc).



#### 4.3 Architettura client-server di Docker



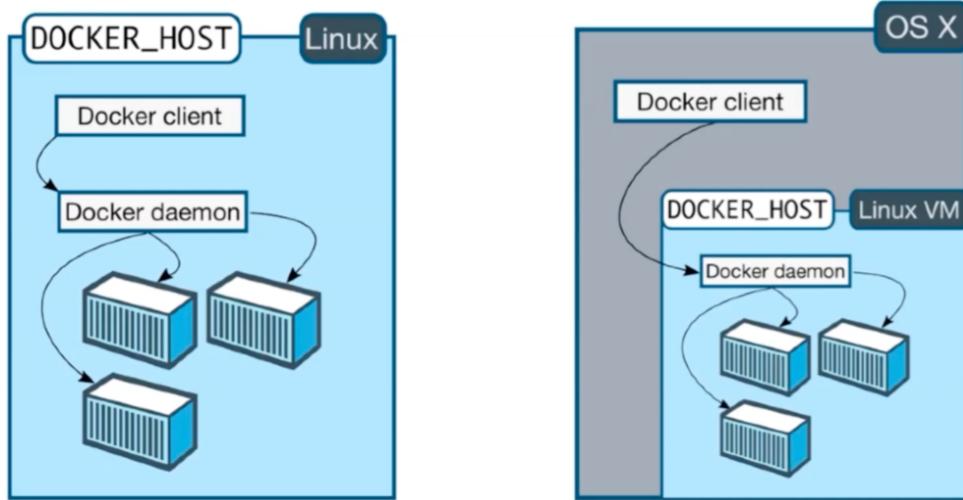
Docker ha una struttura formata da un client ed un server il quale ha lo scopo di gestire i container e le immagini che vengono montate sui container stessi. Gli utenti possono interagire con il server docker tramite il client in due modi:

- La command line (CLI).
- Kitematic, un'interfaccia grafica.

Il server docker viene chiamato anche **docker daemon** o **docker engine** o ancora **docker server**.

È possibile installare il docker client e il docker server sulla stessa macchina (anche se non conviene) purché docker venga eseguito nativamente su una macchina linux. Se non si dispone di una macchina linux bisogna eseguirlo su una virtual machine.

nb. scaricando il pacchetto per un OS diverso linux si installerà un ambiente linux.



#### 4.4 Installazione docker

##### Installazione di docker su Windows Server 2016:

1. Aprire PowerShell come Amministratore
- 2.

```
Install-Module -Name DockerMsftProvider -Repository PSGallery --Force
```

3. Ad una domanda digitare y
- 4.

```
Install-Package -Name docker -ProviderName DockerMsftProvider -verbose
```

5. Ad una domanda digitare y
6. Verrà richiesto di riavviare l'host.
7. Dopo il riavvio, riapriamo PowerShell come Amministratore.
8. Visualizziamo il servizio docker installato con:

```
get-service docker
```

9. Se non fosse in esecuzione bisognerebbe digitare:

```
start-service docker
```

10. Visualizziamo la versione di docker con:

```
docker version
```

Visualizzeremo una schermata simile a questa:

```
PS C:\Users\Administrator> docker version
Client:
 Version: 18.09.0
 API version: 1.39
 Go version: go1.10.3
 Git commit: 33a45cd0a2
 Built: unknown-buildtime
 OS/Arch: windows/amd64
 Experimental: false

Server:
 Engine:
 Version: 18.09.0
 API version: 1.39 (minimum version 1.)
 Go version: go1.10.3
 Git commit: 33a45cd0a2
 Built: 11/07/2018 00:24:12
 OS/Arch: windows/amd64
 Experimental: false
PS C:\Users\Administrator>
```

11. Verifichiamo la corretta installazione eseguendo un container di test, con il comando:

```
docker run hello-world
```

Questo comando scarica un'immagine di prova e la esegue in un contenitore. Quando il contenitore viene eseguito, stampa un messaggio ed esce.

**Ps.** Con questa installazione è possibile eseguire SOLO container nativi windows (vedere immagine sopra nella riga individuata da OS/Arch). Per riprova di quanto detto digitare il comando:

```
docker run ubuntu
```

Darà un errore simile.

```
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
C:\Program Files\Docker\docker.exe: no matching manifest for unknown in the manifest list entries.
See 'C:\Program Files\Docker\docker.exe run --help'.
```

### Installazione di docker su Windows 10:

L'installazione di docker su windows 10 farà installare una macchina virtuale linux, in maniera tale da emulare il kernel linux e riuscire ad avviare container che non siano nativi windows.

1. Recarsi a questo [LINK](#) e premere su "Docker Desktop for Windows".
2. .....

### Installazione di docker su Linux/Ubuntu:

1. Aggiornare i repository con il comando:

```
sudo apt update
```

- 2.

```
sudo apt install apt-transport-https ca-certificates curl
software-properties-common
```

3. Confermiamo con **y**

- 4.

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add
-
```

5. Otterremo una risposta: OK

6. Aggiungiamo il repository di docker alla nostra lista di repository:

```
sudo add-apt-repository "deb [arch=amd64]
https://download.docker.com/linux/ubuntu bionic stable"
```

7. Aggiorniamo nuovamente la lista dei repository con il comando:

```
sudo apt update
```

- 8.

```
apt-cache policy docker-ce
```

9. Possiamo procedere l'installazione effettiva del servizio docker con il comando:

```
sudo apt install docker-ce
```

10. Verifichiamo che il servizio sia effettivamente in esecuzione con il comando:

```
sudo systemctl status docker
```

```
(base) [REDACTED] -mint:~$ sudo systemctl status docker
[sudo] password di [REDACTED]:
● docker.service - Docker Application Container Engine
 Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: enabled)
 Active: active (running) since Thu 2022-05-12 18:21:47 CEST; 1h 2min ago
 TriggeredBy: ● docker.socket
 Docs: https://docs.docker.com
 Main PID: 39170 (dockerd)
 Tasks: 10
 Memory: 35.6M
 CGroupl: /system.slice/docker.service
 └─39170 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock
```

11. Testiamo la corretta installazione con il comando:

```
sudo docker version
```

```
(base) [REDACTED] -mint:~$ sudo docker version
Client: Docker Engine - Community
 Version: 20.10.16
 API version: 1.41
 Go version: go1.17.10
 Git commit: aa7e414
 Built: Thu May 12 09:17:28 2022
 OS/Arch: linux/amd64
 Context: default
 Experimental: true

Server: Docker Engine - Community
 Engine:
 Version: 20.10.16
 API version: 1.41 (minimum version 1.12)
 Go version: go1.17.10
 Git commit: f756502
 Built: Thu May 12 09:15:33 2022
 OS/Arch: linux/amd64
 Experimental: false
 containerd:
 Version: 1.6.4
 GitCommit: 212e8b6fa2f44b9c21b2798135fc6fb7c53efc16
 runc:
 Version: 1.1.1
 GitCommit: v1.1.1-0-g52de29d
 docker-init:
 Version: 0.19.0
 GitCommit: de40ad0
```

12. Verifichiamo infine se il container "Hello world" venga eseguito con successo, eseguiamo il comando:

```
sudo docker run hello-world
```

Questo comando scarica un'immagine di prova e la esegue in un contenitore. Quando il contenitore viene eseguito, stampa un messaggio ed esce.

```
(base) [REDACTED]-mint:~$ sudo docker run hello-world
```

```
Hello from Docker!
This message shows that your installation appears to be working correctly.
```

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.  
(amd64)
3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

To try something more ambitious, you can run an Ubuntu container with:

```
$ docker run -it ubuntu bash
```

Share images, automate workflows, and more with a free Docker ID:

```
https://hub.docker.com/
```

For more examples and ideas, visit:

```
https://docs.docker.com/get-started/
```

#### 4.4.1 Passaggi post installazione opzionali tuttavia consigliati, [1]

Dopo aver installato docker viene suggerito di:

- **Gestire docker come utente NON root:**

1. Creazione del gruppo docker, tramite il comando:

```
sudo groupadd docker
```

```
(base) [REDACTED]-mint:~$ sudo groupadd docker
groupadd: group 'docker' already exists
```

2. Aggiunta del nostro utente corrente al gruppo docker, tramite il comando:

```
sudo usermod -aG docker $USER
```

3. Riavviare l'host
4. Facciamo un test, tramite il comando:

```
docker run hello-world
```

- **Configurare docker per avviarsi all'avvio:**

La maggior parte delle attuali distribuzioni Linux (RHEL, CentOS, Fedora, Debian, Ubuntu 16.04 e versioni successive) utilizzano systemd per gestire quali servizi vengono avviati all'avvio del sistema. Su Debian e Ubuntu, il servizio Docker è configurato per l'avvio all'avvio per impostazione predefinita. Per avviare automaticamente Docker e Containerd all'avvio per altre distribuzioni, utilizzare i comandi seguenti:

```
sudo systemctl enable docker.service
sudo systemctl enable containerd.service
```

Per disabilitare questo comportamento, utilizzare invece:

```
sudo systemctl disable docker.service
sudo systemctl disable containerd.service
```

## 4.5 Immagini, container e registri docker

Possiamo identificare le immagini come dei container stoppati; come dei template o dei modelli da cui poter, successivamente, eseguire uno o più container.

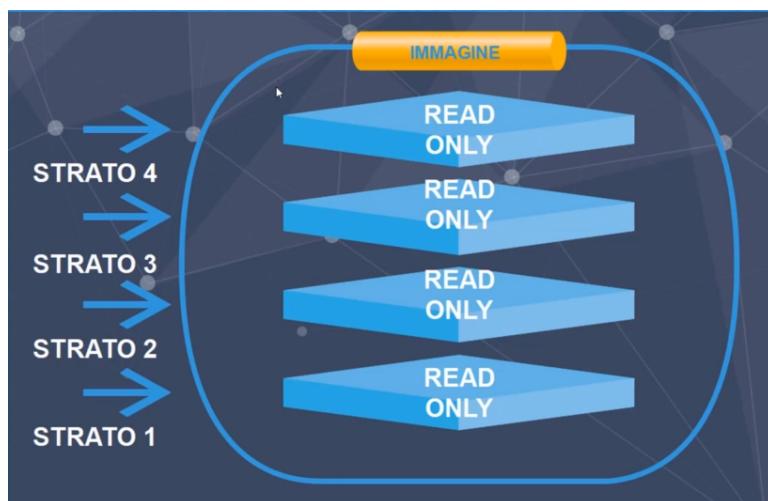
Le immagini possono essere scaricate da un *registro* in cui quest'ultime sono archiviate. Il registro più conosciuto è il [docker hub](#). Ma non è l'unico.

L'operazione di PULL di un'immagine effettua il download dell'immagine stessa da un registro sul nostro docker locale e ci permette quindi di eseguire il container derivato da tale immagine.

Possiamo considerare l'immagine composta da più strati (**layers**) impilati uno di seguito all'altro e che insieme compongono un unico oggetto. Ogni strato contribuisce a rendere l'immagine completa ed eseguibile.

Un'immagine è composta da più strati sovrapposti. Questi strati sono in modalità "solo lettura".

Sarà compito del docker engine unificare e considerare tutti questi strati come una singola entità.



### Gli strati di un'immagine

È possibile verificare gli strati che compongono un'immagine docker mediante due modi:

1. Mediante il comando

```
docker pull redis
```

```
$ docker pull redis
Using default tag: latest
latest: Pulling from library/redis
214ca5fb9032: Pull complete
9eeabf2ad250: Pull complete
b8eb79a9f3c4: Pull complete
0ba9bf1b547e: Pull complete
2d2e2b28e876: Pull complete
3e45fcdfb831: Pull complete
STRATI
Digest: sha256:ad0705f2e2344c4b642449e658ef4669753d6eb70228d46267685045bf932303
Status: Downloaded newer image for redis:latest
docker.io/library/redis:latest
```

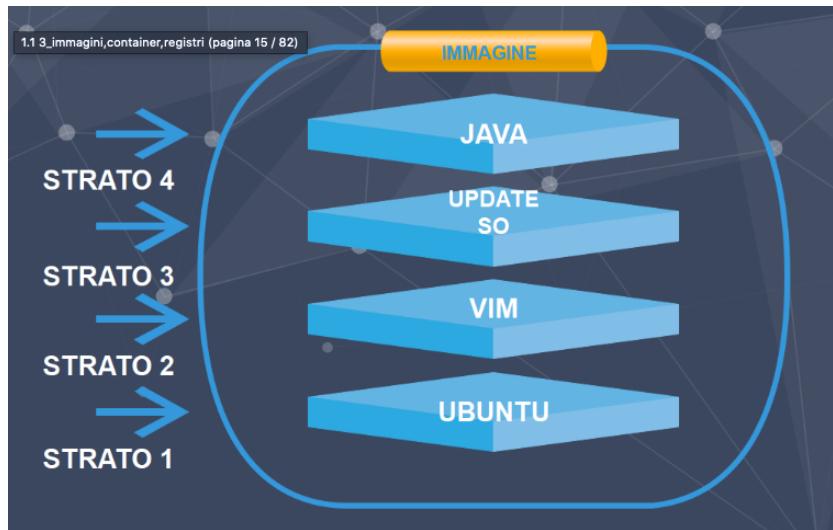
Come possiamo vedere lo scaricamento avviene per strati e alla fine sarà il compito del docker engine comporre il tutto affinché l'immagine sia pronta per essere utilizzata come container.

2. Un'altra modalità per osservare gli strati di un'immagine è tramite il comando:

```
docker inspect redis:latest
```

\* l'output non lo riporta in quanto molto lungo.

Notare che tutte le immagini Docker nascono con uno strato base. Ogni modifica effettuata all'immagine incrementerà il numero di strati. Successivamente quando analizzeremo il dockerfile vedremo come i comandi contenuti in quest'ultimo contribuiscono all'incremento degli strati che compongono l'immagine. Per esempio:



Ricordiamo che come best practice è necessario che l'immagine permetta l'esecuzione di uno solo servizio o applicazione, è quindi sconsigliato avere un container che esegua molti servizi o applicazioni. Per questo motivo solitamente le immagini docker sono leggere e fornite solo delle componenti essenziali.

### Build-time vs run-time

Possiamo definire le immagini come delle entità “build-time” ovvero che assumono significato solo nel momento della loro definizione a differenza dei container che sono entità “run-time” ovvero che assumono significato solo durante la loro esecuzione. Un'immagine sarà sempre un elemento statico, immobile da cui poter derivare molteplici istanze (container).



Tenere conto che si crea una dipendenza tra immagine e container. Se eseguiamo un container, quest'ultimo dipende dall'immagine da cui è derivato, infatti non è possibile eliminare un'immagine finché questa è in uso dal container.

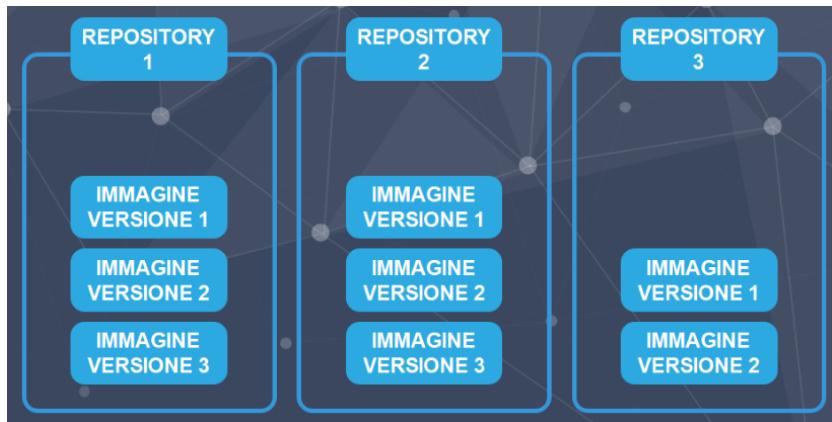
### Registri e repository

Il Docker Hub ci permette di selezionare l'immagine che più si adatta alle nostre esigenze e le immagini al suo interno sono migliaia. È fondamentale prendere confidenza con quest'ultimo in quanto la corretta scelta dell'immagine può far la differenza e migliorare l'esito di un progetto.

### Registri

Esistono anche altri registri, oltre al Docker Hub, alcuni di questi sono locati on-premise così da garantire un buon livello di sicurezza. I registri contengono uno o più REPOSITORY. Un repository contiene una o più IMMAGINI.

Un’ulteriore distinzione, presente anche sul Docker Hub, è tra repository ufficiale e non ufficiale. I repository ufficiali contengono spesso delle immagini qualitativamente migliori, più aggiornate e sicure. La maggior parte dei sistemi operativi e delle applicazioni hanno il loro repository ufficiale sul Docker Hub.



## "Pulling" di un'immagine

Come già anticipato, effettuare il “pull” di un’immagine significa scaricare l’immagine stessa sul docker installato localmente. E’ possibile specificare anche la versione ma lo vedremo più avanti. Docker verifica sempre se nella cache locale è già presente l’immagine o meno. Se non è presente procede con l’operazione di “pulling”. La seguente immagine descrive quanto detto:

```
[node1] (local) root@192.168.0.8 ~
$ sudo docker image ls
REPOSITORY TAG IMAGE ID CREATED SIZE
[node1] (local) root@192.168.0.8 ~
$ sudo docker pull alpine
Using default tag: latest
latest: Pulling from library/alpine
df9b9388f04a: Pull complete
Digest: sha256:4edbd2beb5f78b1014028f4fbb99f3237d9561100b6881aabbf5acce2c4f9454
Status: Downloaded newer image for alpine:latest
docker.io/library/alpine:latest
[node1] (local) root@192.168.0.8 ~
$ sudo docker image ls
REPOSITORY TAG IMAGE ID CREATED SIZE
alpine latest 0ac33e5f5afa 5 weeks ago 5.57MB
[node1] (local) root@192.168.0.8 ~
$ sudo docker pull alpine
Using default tag: latest
latest: Pulling from library/alpine
Digest: sha256:4edbd2beb5f78b1014028f4fbb99f3237d9561100b6881aabbf5acce2c4f9454
Status: Image is up to date for alpine:latest
docker.io/library/alpine:latest
```

In [1] possiamo verificare che l'immagine non è presente in locale, quindi il successivo pull [2] è costretto a scaricare da internet e salvarla in locale [3]. Rifacendo il pull della stessa immagine docker si accorge che è salvato in locale e quindi non la scarica di nuovo [4].

Il comando per effettuare il "pulling" di un'immagine dal repository ufficiale è:

```
docker image pull <nome repository>:<tag immagine>
```

**esempi:**

- con

```
docker image pull redis:5.0.1
```

verrà scaricata l'immagine docker relativa alla versione di redis 5.0.1.

- con

```
docker image pull redis:latest
```

verrà scaricata l'immagine docker relativa all'ultima versione di redis.

- con

```
docker image pull redis
```

se non specifichiamo nessun tag verrà scaricata l'ultima versione come specificando il tag "latest".

Il comando "docker run..." si utilizza per l'avvio di un container. Tuttavia possiamo utilizzarlo anche per effettuare il pull di un'immagine: prima verificherà se l'immagine è già presente localmente, poi provvederà allo scaricamento e all'esecuzione del container.

```
[node1] (local) root@192.168.0.8 ~
$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
2db29710123e: Pull complete
Digest: sha256:80f31dalac7b312ba29d65080fdddf797dd76acfb870e677f390d5acba9741b17
Status: Downloaded newer image for hello-world:latest
```

Hello from Docker!  
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.  
(amd64)
3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

To try something more ambitious, you can run an Ubuntu container with:

```
$ docker run -it ubuntu bash
```

Share images, automate workflows, and more with a free Docker ID:  
<https://hub.docker.com/>

For more examples and ideas, visit:  
<https://docs.docker.com/get-started/>

1

2

### Pulling di un'immagine dal docker hub

Dirigersi al sito <https://hub.docker.com>, premere Explore e successivamente scegliere un'immagine (nell'esempio sotto scegliamo Ubuntu).

The screenshot shows the Docker Hub search interface. A red box highlights the search bar with the query 'ubuntu'. A red arrow labeled '1' points to the search bar. Another red arrow labeled '2' points to the first search result, which is the 'ubuntu' official image card.

**Search results for 'ubuntu':**

- ubuntu** DOCKER OFFICIAL IMAGE
- Updated 15 days ago
- Description: Ubuntu is a Debian-based Linux operating system based on free software.
- Tags: Linux, 386, x86-64, ARM, ARM 64, PowerPC 64 LE, riscv64, IBM Z
- Downloads: 1B+ Stars: 10K+

**alpine** DOCKER OFFICIAL IMAGE

Updated a month ago

Description: A minimal Docker image based on Alpine Linux with a complete package index and only 5 MB in size!

Tags: Linux, x86-64, ARM, ARM 64, 386, PowerPC 64 LE, IBM Z, riscv64

Downloads: 1B+ Stars: 8.8K

Copiare quindi il relativo comando, vedi immagine sotto.

The screenshot shows the Docker Hub page for the 'ubuntu' image. A red arrow points to a button labeled 'Copy and paste to pull this image'.

**ubuntu** DOCKER OFFICIAL IMAGE

Ubuntu is a Debian-based Linux operating system based on free software.

Downloads: 1B+

Tags: Linux, ARM, ARM 64, PowerPC 64 LE, riscv64, IBM Z, 386, x86-64, Docker Official Image

**Copy and paste to pull this image**

**docker pull ubuntu**

[View Available Tags](#)

È possibile inoltre visualizzare i tag disponibili per questa immagine. Ad esempio per l'immagine Ubuntu ci sono i seguenti tag:

## Supported tags and respective Dockerfile links

- 18.04, bionic-20220427, bionic
- 20.04, focal-20220426, focal
- 21.10, impish-20220427, impish
- 22.04, jammy-20220428, jammy, latest, rolling
- 22.10, kinetic-20220428, kinetic, devel
- 14.04, trusty-20191217, trusty
- 16.04, xenial-20210804, xenial

```
[node1] (local) root@192.168.0.8 ~
$ docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
125a6e411906: Pull complete
Digest: sha256:26c68657ccce2cb0a31b330cb0be2b5e108d467f641c62e13ab40cbec258c68d
Status: Downloaded newer image for ubuntu:latest
docker.io/library/ubuntu:latest
[node1] (local) root@192.168.0.8 ~
$ sudo docker image ls
REPOSITORY TAG IMAGE ID CREATED SIZE
ubuntu latest d2e4e1f51132 2 weeks ago 77.8MB
alpine latest 0ac33e5f5afa 5 weeks ago 5.57MB
hello-world latest feb5d9fea6a5 7 months ago 13.3kB
```

### Pulling di un'immagine tramite il "Digest"

Solitamente il “pulling” di un’immagine viene effettuato utilizzando il meccanismo dei tag. Tuttavia per assicurarci che l’immagine scaricata sia effettivamente quella voluta e che i tag siano corretti, si può utilizzare un “hash” chiamato “**digest**”. La particolarità e il punto di forza del “digest” è che il suo valore è **immutabile**.

Ogni volta che effettuiamo il “pull” di un’immagine ci viene fornito in output anche il digest di tale immagine. E’ sufficiente copiarlo e conservarlo, nel caso di un nuovo “pull” potremmo confrontare i valori e se saranno diversi significherà che è stato apportato un cambiamento all’immagine.

*Vedi immagine sopra per identificare il digest.*

```
docker pull ubuntu@digest
```

Esempio:

```
[node2] (local) root@192.168.0.7 ~
$ docker pull ubuntu@sha256:26c68657ccce2cb0a31b330cb0be2b5e108d467f641c62e13ab40cbec258c68d
docker.io/library/ubuntu@sha256:26c68657ccce2cb0a31b330cb0be2b5e108d467f641c62e13ab40cbec258c68d: Pulling from library/ubuntu
125a6e411906: Pull complete
Digest: sha256:26c68657ccce2cb0a31b330cb0be2b5e108d467f641c62e13ab40cbec258c68d
Status: Downloaded newer image for ubuntu@sha256:26c68657ccce2cb0a31b330cb0be2b5e108d467f641c62e13ab40cbec258c68d
docker.io/library/ubuntu@sha256:26c68657ccce2cb0a31b330cb0be2b5e108d467f641c62e13ab40cbec258c68d
[node2] (local) root@192.168.0.7 ~
$ docker image ls
REPOSITORY TAG IMAGE ID CREATED SIZE
ubuntu <none> d2e4e1f51132 2 weeks ago 77.8MB
```

Questa modalità imposta il campo TAG su **<none>** e non su **latest**.

### Manifest List e immagini multi-architettura

Dal momento che la stessa immagine e la stessa versione, può essere adatta al “pulling” da parte di architetture differenti, si è introdotto il concetto di “manifest list” che contiene appunto la lista delle architetture disponibili relativamente ad una certa immagine e versione (tag).

Non dobbiamo specificare noi l’architettura, Docker lo farà in automatico.

Esempio di manifest list:

```
{
 "schemaVersion": 2,
 "mediaType": "application/vnd.docker.distribution.manifest.list.v2+json",
 "manifests": [
 {
 "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
 "size": 424,
 "digest": "sha256:f67dcc5fc786f04f0743abfe0ee5dae9bd8caf8efa6c8144f7f2a43889dc513b",
```

```

 "platform": {
 "architecture": "arm",
 "os": "linux"
 },
 {
 "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
 "size": 424,
 "digest": "sha256:b64ca0b60356a30971f098c92200b1271257f100a55b351e6bbe985638352f3a",
 "platform": {
 "architecture": "amd64",
 "os": "linux"
 }
 },
 {
 "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
 "size": 425,
 "digest": "sha256:df436846483aff62bad830b730a0d3b77731bcf98ba5e470a8bbb8e9e346e4e8",
 "platform": {
 "architecture": "ppc64le",
 "os": "linux"
 }
 },
 {
 "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
 "size": 425,
 "digest": "sha256:5bb8e50aa2edd408bdf3ddf61efb7338ff34a07b762992c9432f1c02fc0e5e62",
 "platform": {
 "architecture": "s390x",
 "os": "linux"
 }
 }
}
]
}

```

Per ispezionare il manifest di una qualsiasi immagine basta digitare il seguente comando:

```
sudo docker inspect <nome_immagine>
```

## Avvio e interazione di un container

Il comando “**docker container run**” ci permette di eseguire un container. Questo è il comando nella sua forma base, come vedremo abbiamo la possibilità di aggiungere delle opzioni e parametri.

Normalmente si utilizza un container allo scopo di utilizzare un’applicazione o un servizio; il comando assume quindi la seguente forma: “**docker container run <image> <app>**”.

Ad esempio: “**docker container run -it ubuntu /bin/bash**” ci permette di avviare una shell di tipo “bash” all’interno del container Ubuntu.

L’opzione “**-it**” collega il nostro terminale a quello del container così da poter effettuare le attività all’interno di quest’ultimo.

**Nota che un container resta in esecuzione finché l’applicazione o il servizio in esecuzione esiste.**

Il comando “**sleep**” ci permette di effettuare un test proprio in tal senso.

Inoltre, avviando un container ci rendiamo presto conto che alcuni comandi che normalmente

utilizziamo non sono disponibili: questo è normale. L'immagine contiene solo ciò che è strettamente necessario. Ovviamente abbiamo la possibilità di scaricare in seguito ciò che ci occorre.

Con il comando “**ps -elf**” possiamo verificare i processi in esecuzione all'interno del container. Ripetiamo ancora una volta: il container esiste solo in funzione dell'applicazione o servizio attualmente in esecuzione. Per cui se stoppiamo il processo associato a tale applicazione, anche il container cesserà la sua esecuzione.

Per uscire da un container in esecuzione, possiamo digitare la combinazione di tasti: “**CTRL+P** e **CTRL+Q**” così facendo il container resterà in esecuzione e noi torneremo sul terminal dell'host.

Per rientrare sul terminale del container digitiamo il comando: “**docker container exec -it ID\_CONTAINER bash**”.

```
[node2] (local) root@192.168.0.17 ~
$ docker run -it ubuntu /bin/bash
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
125a6e411906: Pull complete
Digest: sha256:26c68657ccce2cb0a31b330cb0be2b5e108d467f641c62e13ab40cbec258c68d
Status: Downloaded newer image for ubuntu:latest
root@f8a4f2ff2402:/# ls
bin dev home lib32 libx32 mnt proc run srv tmp var
boot etc lib lib64 media opt root sbin sys usr
root@f8a4f2ff2402:/# [node2] (local) root@192.168.0.17 ~
$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
f8a4f2ff2402 ubuntu "/bin/bash" 58 seconds ago Up 57 seconds wizardly_neumann
[node2] (local) root@192.168.0.17 ~
$ docker stop f8a4f2ff2402
f8a4f2ff2402
[node2] (local) root@192.168.0.17 ~
$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
```

## Comandi di gestione

- "docker ls" - Per verificare se il container è effettivamente in esecuzione.
- "docker ps" - Possiamo verificare i container in esecuzione.
- "docker ps -a" - Visualizziamo anche i container stoppati.
- "docker start <id\_container>" - Per effettuare lo start di un container stoppato.
- "docker stop <id\_container>" - Per stoppare un container.
- "docker rm <id\_container>" - Per rimuoverlo.
- "docker container prune" - Rimuove tutti i container che sono stati stoppati. È fortemente consigliato prima di rimuovere un container di effettuare lo stop di quest'ultimo. questo garantisce una corretta gestione dei processi e quindi una situazione più “pulita”.
- Per avviare un container in background si utilizza il flag "-d". Esempio: "docker run -d -it ubuntu /bin/bash".
- Il flag "--name" ci permette di dare un nome al container in esecuzione. Esempio: "docker run --name=esercizio1 -it ubuntu /bin/bash"
- Per accedere ad un container possiamo utilizzare anche il comando "docker attach <id\_container>"
- "docker top <id\_container>" - Per visualizzare i processi di un container in esecuzione.
- "docker stats <id\_container>" - Per visualizzare le statistiche di un container in esecuzione.
- "docker ps -q | xargs docker stats" - È la fusione del comando "docker ps" e "docker stats". Permette quindi di monitorare tutti i container in esecuzione.

- "docker logs <id\_container>" - Per visualizzare i log di un container.

## Persistenza dei dati di un container

Notare che le informazioni all'interno di un container **persistono finché quest'ultimo non viene eliminato definitivamente**. Lo stop di un container non porta alla perdita dei dati al suo interno. Verifichiamo:

```
[node1] (local) root@192.168.0.13 ~
$ docker run -d -it ubuntu /bin/bash
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
125a6e411906: Pull complete
Digest: sha256:26c68657ccce2cb0a31b330cb0be2b5e108d467f641c62e13ab40cbec258c68d
Status: Downloaded newer image for ubuntu:latest
cb838b82b8cc2f625926242cb8e731d077084d94cb7874f3be9b080913:f74b61
[node1] (local) root@192.168.0.13 ~
$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
cb838b82b8cc ubuntu "/bin/bash" 22 seconds ago Up 20 seconds xenodochial_driscoll
[node1] (local) root@192.168.0.13 ~
$ docker attach cb838b82b8cc
root@cb838b82b8cc:/# ls
bin dev home lib32 libx32 mnt proc run srv tmp var
boot etc lib lib64 media opt root sbin sys usr
root@cb838b82b8cc:/# touch prova.txt
root@cb838b82b8cc:/# ls
bin dev home lib32 libx32 mnt proc root sbin sys usr
boot etc lib lib64 media opt prova.txt run srv tmp var
```

In [1] creiamo il container in background ed eseguiamo come processo attivo una shell interattiva. Come possiamo vedere in [2] è attiva. A questo punto ci accediamo [3] e creiamo al suo interno un file denominato *prova.txt* [4]. Con */s* vediamo l'effettiva creazione [5].

Adesso si stoppa il container, usciamo dal container con CTRL+P e CTRL+Q.

Riavviamo il container e come possiamo osservare il file è ancora presente.

```
root@cb838b82b8cc:/# read escape sequence
[node1] (local) root@192.168.0.13 ~
$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
cb838b82b8cc ubuntu "/bin/bash" 8 minutes ago Up 8 minutes xenodochial_driscoll
[node1] (local) root@192.168.0.13 ~
$ docker stop cb838b82b8cc
cb838b82b8cc
[node1] (local) root@192.168.0.13 ~
$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
[node1] (local) root@192.168.0.13 ~
$ docker start cb838b82b8cc
cb838b82b8cc
[node1] (local) root@192.168.0.13 ~
$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
cb838b82b8cc ubuntu "/bin/bash" 9 minutes ago Up 9 seconds xenodochial_driscoll
[node1] (local) root@192.168.0.13 ~
$ docker attach cb838b82b8cc
root@cb838b82b8cc:/# ls
bin dev home lib32 libx32 mnt proc root sbin sys usr
boot etc lib lib64 media opt prova.txt run srv tmp var
root@cb838b82b8cc:/# []
```

## Policy di restart

È una best practice consigliata quella di creare una **restart policy** che permette il riavvio automatico del container al verificarsi di un qualche tipo di evento, imprevisto o meno che sia. Esistono tre tipologie di restart policy:

- **Always.** Questa tipologia riavvia sempre il container salvo se quest'ultimo è stato manualmente stoppato. Se viene riavviato il Docker Daemon tutti i container che sono stoppati e che contengono la policy "restart always" saranno avviati. Esempio:

```
docker run -d -it --restart always ubuntu /bin/bash
```

- **Unless-stopped.** Questa tipologia non riavvia il container anche se quest'ultimo è stato stoppato e il Docker Daemon si è riavviato.
- **On-failed.** Questa tipologia riavvia il container se quest'ultimo esce con un "non-zero exit code" e lo riavvia nel caso il Docker Daemon sia riavviato e lo stato sia "stopped".

## 4.6 Persistenza dei dati e Volumi

### Storage Drivers

I container necessitano di uno spazio disco locale per poter effettuare la "stratificazione" delle immagini e poter montare il filesystem.

La componente che si occupa della gestione di questo spazio disco è lo "**storage drivers**". È uno spazio disco non persistente e dura per l'intero ciclo di vita del container.

Linux supporta differenti tipologie di *Storage Drivers* e queste ultime hanno un forte impatto sulla stabilità del container. I più significativi storage drivers in Linux sono:

- Aufs.
- Zfs.
- Btrfs.
- Overlay2.
- devicemapper.

Docker in ambiente Windows supporta un solo storage driver: "**windowsfilter**".

**La scelta dello storage driver avviene sempre per host e mai per container.** In linux lo storage driver si può impostare sotto: /etc/docker/daemon.json

Con il comando "**docker system info**" o semplicemente "**docker info**" si può verificare la tipologia di storage driver attualmente in uso.

Per verificare quale sia la scelta migliore, dobbiamo sempre fare riferimento alla documentazione. Ad esempio, per la distribuzione Ubuntu:

- Se il kerner è uguale o superiore a 4.x allora è consigliato "overlay2".
- Se il kerner è inferiore a 4.x allora è consigliato "aufs".

### Volumi

In docker abbiamo la possibilità di memorizzare i dati in modo persistente e non persistente. La scelta dipende dall'utilizzo che dobbiamo fare dei dati.

Se la nostra esigenza è avere a disposizione i dati oltre la durata del container, allora dobbiamo introdurre il concetto di **volume**. Alcuni dei vantaggi sono:

- Separare i container dallo storage.
- Condividere un volume tra container differenti.
- Non perdere i dati ad eliminazione del container.

Il volume, quindi, ci garantisce che i dati continuino a persistere anche dopo l'eliminazione del container. In pratica:

1. Creiamo un volume.
2. Creiamo il container.
3. Montiamo il volume all'interno del container.
4. In fase di "mount" avremo specificato una cartella specifica all'interno del filesystem del container.
5. I dati saranno quindi scritti in questa directory.
6. Se il container sarà eliminato, i dati continueranno ad esistere.



I comandi per la gestione dei volumi sono (digitare "docker volume help" per la lista completa):

- "**docker volume create <nome\_volume>**" - Crea un nuovo volume. E' possibile creare un volume anche utilizzando il comando "docker run" e l'opzione "-v" non specificando il path assoluto ma il nome che dovrà avere il volume.  
E' possibile utilizzare anche un sistema di memorizzazione esterno tramite l'utilizzo di driver di terze parti. Ad esempio, facendo riferimento all'ecosistema Amazon AWS, possiamo utilizzare le seguenti tipologie di storage:
  - Object Storage → Amazon
  - File Storage → Amazon Elastic File
  - Block Storage → Amazon Elastic Block
- "**docker volume ls**" - Visualizza la lista dei volumi presenti.
- "**docker volume inspect**" - Visualizza in maniera dettagliata informazioni di uno o più volumi.
- "**docker volume prune**" - Rimuove tutti i volumi locali inutilizzati.
- "**docker volume rm <nome\_volume>**" - Rimuove uno o più volumi.

### Verifichiamo la persistenza dei volumi con un esempio pratico

1. Verifichiamo prima la presenza di immagini salvate nell'host con il comando:

```
[node2] (local) root@192.168.0.7 ~
$ docker ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
```

2. Non ce ne sono, scarichiamo quindi un'immagine con il comando:

```
[node2] (local) root@192.168.0.7 ~
$ docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
125a6e411906: Pull complete
Digest: sha256:26c68657ccce2cb0a31b330cb0be2b5e108d467f641c62e13ab40cbec258c68d
Status: Downloaded newer image for ubuntu:latest
docker.io/library/ubuntu:latest
[node2] (local) root@192.168.0.7 ~
$ docker image ls
REPOSITORY TAG IMAGE ID CREATED SIZE
ubuntu latest d2e4elf51132 2 weeks ago 77.8MB
```

3. Creiamo un volume da attaccare al container successivamente:

```
[node2] (local) root@192.168.0.7 ~
$ docker volume create volumeprimo
volumeprimo
[node2] (local) root@192.168.0.7 ~
$ docker volume ls
DRIVER VOLUME NAME
local volumeprimo
```

4. Eseguiamo il container in maniera interattiva (con -it), gli diamo un nome (con --name), e gli colleghiamo il volume "volumeprimo" precedentemente creato (con -v) specificando <nome\_volume>:<path\_destinazione>, infine scegliamo l'immagine ubuntu precedentemente scaricata passando come processo la shell bash.

```
[node2] (local) root@192.168.0.7 ~
$ docker run -it --name containerTest -v volumeprimo:/test ubuntu bash
root@49b08653cf4d:/# ls
bin dev home lib32 libx32 mnt proc run srv test usr
boot etc lib lib64 media opt root sbin sys tmp var
root@49b08653cf4d:/#
```

5. Spostiamoci nella directory del volume e creiamo un file di testo.

```
root@49b08653cf4d:/# cd test
root@49b08653cf4d:/test# touch testo.txt
root@49b08653cf4d:/test# ls
testo.txt
root@49b08653cf4d:/test#
```

6. Usciamo dal container con il comando exit. Notare che uscendo dal container automaticamente lo stoppiamo in quanto termina il processo bash.

```
root@49b08653cf4d:/test# exit
exit
[node2] (local) root@192.168.0.7 ~
$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
```

7. Cancelliamo il container stoppato e verifichiamo la presenza del volume.

```
[node1] (local) root@192.168.0.8 ~
$ docker ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
11904457c06f ubuntu "bash" 3 minutes ago Exited (0) 44 seconds ago
[node1] (local) root@192.168.0.8 ~
$ docker rm 11904457c06f
11904457c06f
[node1] (local) root@192.168.0.8 ~
$ docker ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
[node1] (local) root@192.168.0.8 ~
$ docker volume ls
DRIVER VOLUME NAME
local volumeprimo
```

8. Come possiamo notare la cancellazione di un container non provoca la cancellazione dei volumi ad esso collegati. Vediamo se vale lo stesso per i file memorizzati al suo interno:

```

[node1] (local) root@192.168.0.8 ~
$ docker volume inspect volumeprimo
[
 {
 "CreatedAt": "2022-05-17T17:03:51Z",
 "Driver": "local",
 "Labels": {},
 "Mountpoint": "/var/lib/docker/volumes/volumeprimo/_data",
 "Name": "volumeprimo",
 "Options": {},
 "Scope": "local"
 }
]
[node1] (local) root@192.168.0.8 ~
$ cd /var/lib/docker/volumes/volumeprimo/_data
[node1] (local) root@192.168.0.8 /var/lib/docker/volumes/volumeprimo/_data
$ ls
test.txt ←

```

9. Come possiamo vedere anche i file al suo interno rimangono preservati nonostante la rimozione del container.

Da notare che è possibile creare il volume anche in fase di caricamento del container:

```

[node2] (local) root@192.168.0.7 ~
$ docker volume ls
DRIVER VOLUME NAME
local volumeprimo
[node2] (local) root@192.168.0.7 ~
$ docker run -it -v volumesecondo:/test ubuntu bash
root@c9f66d9e4f55:/# exit
exit
[node2] (local) root@192.168.0.7 ~
$ docker volume ls
DRIVER VOLUME NAME
local volumeprimo
local volumesecondo ←
[node2] (local) root@192.168.0.7 ~

```

### Mapping verso una location fisica

Per la persistenza dei dati è possibile usare sia i volumi (descritti precedentemente) oppure mappare una location fisica del nostro host tramite sempre il comando "-v".

Esempio:

1. Per conoscere il path del punto di mount digitare:

```

[node1] (local) root@192.168.0.7 ~
$ pwd
/root

```

2. Creiamo una cartella per mappare l'host con il container.

```

[node1] (local) root@192.168.0.7 ~
$ mkdir cartellaTest
[node1] (local) root@192.168.0.7 ~
$ ls
cartellaTest
[node1] (local) root@192.168.0.7 ~
$ █

```

3. Eseguiamo il container mappando la cartella creata del nostro host su /test del container:

```
[node1] (local) root@192.168.0.7 ~
$ docker run -it --name container1 -v /root/cartellaTest:/test ubuntu bash
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
125a6e411906: Pull complete
Digest: sha256:26c68657ccce2cb0a31b330cb0be2b5e108d467f641c62e13ab40cbec258c68d
Status: Downloaded newer image for ubuntu:latest
root@bb2b6784e860:/#
```

4. Spostiamoci all'interno della cartella test e creiamo un file.

```
root@bb2b6784e860:/# ls
bin dev home lib32 libx32 mnt proc run srv test usr
boot etc lib lib64 media opt root sbin sys tmp var
root@bb2b6784e860:/# cd test
root@bb2b6784e860:/test# ls
root@bb2b6784e860:/test# touch testuno.txt
root@bb2b6784e860:/test# ls
testuno.txt
root@bb2b6784e860:/test#
```

5. Usciamo dal container senza stopparlo (CTRL+P e CTRL+Q) e rechiamoci nella directory dell'host condivisa con il container e verifichiamo la presenza del file.

```
root@bb2b6784e860:/test# [node1] (local) root@192.168.0.7 ~
$ ls
cartellaTest
[node1] (local) root@192.168.0.7 ~
$ cd cartellaTest
[node1] (local) root@192.168.0.7 ~/cartellaTest
$ ls
testuno.txt
[node1] (local) root@192.168.0.7 ~/cartellaTest
$
```

6. Il file è presente, proviamo adesso il viceversa. Creiamo un file nella cartella host da utente host.

```
[node1] (local) root@192.168.0.7 ~/cartellaTest
$ touch testdue.txt
[node1] (local) root@192.168.0.7 ~/cartellaTest
$ ls
testdue.txt testuno.txt
```

7. Rientriamo nel container e verifichiamo la presenza del file caricato dall'utente host:

```
[node1] (local) root@192.168.0.7 ~/cartellaTest
$ docker exec -it container1 bash
root@bb2b6784e860:/# ls
bin dev home lib32 libx32 mnt proc run srv test usr
boot etc lib lib64 media opt root sbin sys tmp var
root@bb2b6784e860:/# cd test
root@bb2b6784e860:/test# ls
testdue.txt testuno.txt
```

8. Da come possiamo vedere il file è presente, quindi possiamo fare il mapping tramite una location fisica e non solo tramite un volume. Infatti digitando il comando "docker volume ls" non vedremo il path condiviso:

```
[node1] (local) root@192.168.0.7 ~/cartellaTest
$ docker volume ls
DRIVER VOLUME NAME
```

#### 4.7 Dal servizio al container

I container eseguono delle applicazioni o dei servizi quindi nasce spontanea l'esigenza di convertire un'applicazione e renderla un container. Questo processo prende il nome di: "containerizing" oppure "dockerizing". In italiano non si riesce a tradurre correttamente. "Dockerizing" un'applicazione significa effettuare uno "snapshot" del filesystem e delle dipendenze dell'applicazione stessa. Questo snapshot non sarà altro che la nostra immagine docker.

Creare le proprie immagini grazie alla definizione di un dockerfile è un concetto fondamentale: ci permette di personalizzare le nostre immagini docker sulla base della nostra infrastruttura attuale.

Questo processo ad alto livello si può riassumere nelle seguenti fasi:

1. Creazione dell'applicazione e accesso al codice della stessa.
2. Creazione del "dockerfile" che contiene tutte le informazioni dell'applicazione, le dipendenze e tutto ciò che sarà necessario per eseguirla.
3. Creazione dell'immagine derivata dal dockerfile.
4. Creazione ed esecuzione del container e quindi esecuzione della nostra applicazione.

Tale processo si riassume nelle seguenti macro fasi: **BUILD → SHIP → RUN**.



#### Struttura di un dockerfile

Come già introdotto, il dockerfile ci permette di definire la nostra immagine che, successivamente, tramite il comando "docker image build" sarà effettivamente creata ed utilizzata per avviare i container (docker container run).

Il dockerfile è un semplice file di testo che contiene tutte le istruzioni affinché si possa creare l'immagine voluta.

Docker ci fornisce tutta una serie di comandi da utilizzare all'interno di questo file, tra cui: FROM, CMD, EXPOSE, ENV, COPY, ADD.

Di seguito un esempio di dockerfile

```

FROM golang:1.9.2-alpine3.6 AS build

Install tools required for project
Run `docker build --no-cache .` to update dependencies
RUN apk add --no-cache git
RUN go get github.com/golang/dep/cmd/dep

List project dependencies with Gopkg.toml and Gopkg.lock
These layers are only re-built when Gopkg files are updated
COPY Gopkg.lock Gopkg.toml /go/src/project/
WORKDIR /go/src/project/
Install library dependencies
RUN dep ensure -vendor-only

Copy the entire project and build it
This layer is rebuilt when a file changes in the project directory
COPY . /go/src/project/
RUN go build -o /bin/project

This results in a single layer image
FROM scratch
COPY --from=build /bin/project /bin/project
ENTRYPOINT ["/bin/project"]
CMD ["--help"]

```

o ancora..

```

Use the official image as a parent image
FROM python:latest
Set the working directory
WORKDIR /www
Copy the file or directory content from your host to WORKDIR
COPY www .
Run the command inside your image filesystem
RUN echo "append something" >> index.html
RUN echo "append something more" >> index.html
set env that persists at runtime; can be changed by docker run
ENV PORT 8080
which port the container is listening on at runtime
EXPOSE $PORT
Run the specified command within the container at runtime
CMD python -m http.server $PORT

```

Quando si definisce una nuova immagine, è possibile partire completamente da zero oppure partire da un'immagine già presente, ciò è possibile farlo tramite l'istruzione FROM all'interno del dockerfile.

Prendiamo adesso come esempio l'immagine ALPINE e procediamo ad aggiungere dei tool non presenti nell'immagine base:

```

FROM alpine:latest
RUN apk update
RUN apk add vim

```

Il comando RUN esegue i comandi Linux esplicati. APK è il package manager della distribuzione ALPINE.

Facciamo un esempio:

```
[node2] (local) root@192.168.0.12 ~
$ mkdir dockerfile
[node2] (local) root@192.168.0.12 ~
$ cd dockerfile/
[node2] (local) root@192.168.0.12 ~/dockerfile
$ touch Dockerfile.txt
[node2] (local) root@192.168.0.12 ~/dockerfile
$ ls
Dockerfile.txt
[node2] (local) root@192.168.0.12 ~/dockerfile
$ vi Dockerfile.txt
[node2] (local) root@192.168.0.12 ~/dockerfile
```

Inseriamo nel file Dockerfile.txt le righe:

```
FROM alpine
```

```
CMD ["echo","ciao mondo"]
```

premiamo ESC ed in fondo scrivere :wq e INVIO.

```
$ cat Dockerfile.txt
FROM alpine
CMD ["echo","ciao mondo"]

[node2] (local) root@192.168.0.12 ~/dockerfile
$ docker build -t container1 -f ./Dockerfile.txt .
Sending build context to Docker daemon 2.048kB
Step 1/2 : FROM alpine
latest: Pulling from library/alpine
df9b9388f04a: Pull complete
Digest: sha256:4edb2beb5f78b1014028f4fbb99f3237d9561100b6881aabbf5acce2c4f9454
Status: Downloaded newer image for alpine:latest
--> 0ac33e5f5afa
Step 2/2 : CMD ["echo","ciao mondo"]
--> Running in 48c9e7858a31
Removing intermediate container 48c9e7858a31
--> 38a48ed1119a
Successfully built 38a48ed1119a
Successfully tagged container1:latest
```

Adesso possiamo procedere a lanciare il container:

```
$ docker image ls
REPOSITORY TAG IMAGE ID CREATED SIZE
container1 latest 38a48ed1119a 4 minutes ago 5.57MB
alpine latest 0ac33e5f5afa 6 weeks ago 5.57MB
[node2] (local) root@192.168.0.12 ~/dockerfile
$ docker run --name container1 38a48ed1119a
ciao mondo
```

nb. nell'esempio sopra il file "dockerfile.txt" va ricreato come "dockerfile", così è possibile eseguirlo direttamente con "docker build ."

### Opzione Copy, [1]

Il comando copy permette di copiare un oggetto dalla source, ovvero l'host, verso un container.

```
$ docker cp [OPTIONS] CONTAINER:SRC_PATH DEST_PATH|-|
docker cp [OPTIONS] SRC_PATH|-> CONTAINER:DEST_PATH
```

Il comando `docker cp` copia il contenuto di **SRC\_PATH** nella destinazione **DEST\_PATH**. È possibile copiare dal file system del container all'host e viceversa. Il container può essere in esecuzione o stoppato e **SRC\_PATH** o **DEST\_PATH** possono essere file o cartelle.

## Examples

Copy a local file into container

```
$ docker cp ./some_file CONTAINER:/work
```

Copy files from container to local path

```
$ docker cp CONTAINER:/var/logs/ /tmp/app_logs
```

Copy a file from container to stdout. Please note `cp` command produces a tar stream

```
$ docker cp CONTAINER:/var/logs/app.log - | tar x -O | grep "ERROR"
```

## Opzione Run, [1]

Il comando `run` esegue ogni comando, costruendo nello specifico un nuovo layer sopra l'immagine corrente.

## Opzione CMD, [1]

L'opzione CMD ha tre forme possibili:

- `CMD ["executable","param1","param2"]` (*exec form, this is the preferred form*)
- `CMD ["param1","param2"]` (*as default parameters to ENTRYPPOINT*)
- `CMD command param1 param2` (*shell form*)

Per fare un test creare un'immagine a partire da questo docker file:

```
mkdir dockerdir
cd dockerdir
touch dockerfile
vi dockerfile

FROM ubuntu
RUN apt-get update
RUN apt-get install -y curl
RUN touch ip.txt
RUN curl ifconfig.me/ip -o "ip.txt"
CMD ["cat","ip.txt"]

docker build .
docker run "id container del passo precedente"
```

## Comando Entrypoint, [1]

Questo comando ci permette di specificare un comando e ci da la possibilità successivamente di appendere altre istruzioni a quel comando.

Possiede due forme:

The *exec* form, which is the preferred form:

```
ENTRYPOINT ["executable", "param1", "param2"]
```

The *shell* form:

```
ENTRYPOINT command param1 param2
```

Per fare un test creare un'immagine a partire da questo docker file:

```
mkdir dockerdir
cd dockerdir
touch dockerfile
vi dockerfile

FROM ubuntu
RUN apt-get update
RUN apt-get install -y iputils-ping
ENTRYPOINT ["ping", "-c", "5"]

docker build .
docker run "id container del passo precedente" www.google.com
```

Output:

```
$ docker run 3924273c833e www.google.com
PING www.google.com (142.250.185.132) 56(84) bytes of data.
64 bytes from fra16s50-in-f4.1e100.net (142.250.185.132): icmp_seq=1 ttl=117 time=5.07 ms
64 bytes from fra16s50-in-f4.1e100.net (142.250.185.132): icmp_seq=2 ttl=117 time=5.08 ms
64 bytes from fra16s50-in-f4.1e100.net (142.250.185.132): icmp_seq=3 ttl=117 time=5.09 ms
64 bytes from fra16s50-in-f4.1e100.net (142.250.185.132): icmp_seq=4 ttl=117 time=5.18 ms
64 bytes from fra16s50-in-f4.1e100.net (142.250.185.132): icmp_seq=5 ttl=117 time=5.14 ms

--- www.google.com ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4005ms
rtt min/avg/max/mdev = 5.072/5.115/5.181/0.088 ms
```

## 4.8 Networking di base, [1]

```
Commands:
 connect Connect a container to a network
 create Create a network
 disconnect Disconnect a container from a network
 inspect Display detailed information on one or more networks
 ls List networks
 prune Remove all unused networks
 rm Remove one or more networks
```

Uno dei motivi per cui i container e i servizi Docker sono così potenti è che è possibile collegarli tra loro o collegarli a carichi di lavoro non Docker. I container e i servizi Docker non devono nemmeno sapere che sono distribuiti su Docker o che i loro pari sono carichi di lavoro Docker o meno. Sia che gli host Docker eseguano Linux, Windows o un mix dei due, è possibile utilizzare Docker per gestirli in modo indipendente dalla piattaforma.

Questo argomento definisce alcuni concetti di base della rete Docker e vi prepara a progettare e distribuire le vostre applicazioni per trarre il massimo vantaggio da queste capacità.

Il modello di gestione del networking di docker è denominato: "container network model (cnm)". La sua implementazione è gestita dalla libreria "libnetwork". In questa sezione faremo riferimento alla gestione della rete "single-host". La gestione del networking "multi-host" la esamineremo successivamente.

### Driver di rete

Il sottosistema di rete di Docker è collegabile tramite driver. Diversi driver esistono per impostazione predefinita e forniscono le funzionalità di rete fondamentali:

- **bridge**: Il driver di rete predefinito. Permette la connettività con l'host e con gli altri container.
- **host**: Viene "scavalcato" il networking di Docker. Siamo connessi direttamente alle interfacce dell'host.
- **none**: non viene fornita nessuna rete al container. Il container non ha alcun indirizzo IP assegnato. Non è permesso quindi alcun tipo di comunicazione, sia esterno sia verso altri container.

E' possibile verificare le tipologie di reti disponibili tramite il comando "`docker network ls`":

```
$ docker network ls
NETWORK ID NAME DRIVER SCOPE
0f5316f5830a bridge bridge local
c331296620b2 host host local
9f221d08a361 none null local
```

Per una descrizione dettagliata di una di queste 3 reti digitare il comando: "`docker inspect network <network_id>`":

```
$ docker inspect network 0f5316f5830a
[{"Name": "bridge",
 "Id": "0f5316f5830a796b81610a69b5fd0d90e7f990786fc5af50f2971aadfe821c8",
 "Created": "2022-05-24T16:35:23.925848925Z",
 "Scope": "local",
 "Driver": "bridge",
 "EnableIPv6": false,
 "IPAM": {
 "Driver": "default",
 "Options": null,
 "Config": [
 {
 "Subnet": "172.17.0.0/16"
 }
]
 },
 "Internal": false,
 "Attachable": false,
 "Ingress": false,
 "ConfigFrom": {
 "Network": ""
 },
 "ConfigOnly": false,
 "Containers": {},
 "Options": {
 "com.docker.network.bridge.default_bridge": "true",
 "com.docker.network.bridge.enable_icc": "true",
 "com.docker.network.bridge.enable_ip_masquerade": "true",
 "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
 "com.docker.network.bridge.name": "docker0",
 "com.docker.network.driver.mtu": "1500"
 },
 "Labels": {}
}]
```

## Bridge

In termini di rete, un bridge di rete è un dispositivo di livello link che inoltra il traffico tra i segmenti di rete. Un bridge può essere un dispositivo hardware o un dispositivo software in esecuzione nel kernel di una macchina host.

In termini di Docker, una rete bridge utilizza un bridge software che consente ai container collegati alla stessa rete bridge di comunicare, fornendo al contempo l'isolamento dai container non collegati a quella rete bridge. Il driver bridge di Docker installa automaticamente delle regole nella macchina host in modo che i container su reti bridge diverse non possano comunicare direttamente tra loro.

Le reti bridge si applicano ai contenitori in esecuzione sullo stesso host del demone Docker. Per la comunicazione tra container in esecuzione su host di demoni Docker diversi, è possibile gestire il routing a livello di sistema operativo oppure utilizzare una rete overlay.

Quando si avvia Docker, viene creata automaticamente una rete ponte predefinita (chiamata anche bridge), alla quale si collegano i nuovi container, a meno che non sia specificato diversamente. È anche possibile creare reti bridge personalizzate definite dall'utente. Le reti bridge definite dall'utente sono superiori alla rete bridge predefinita.

La modalità bridge ci permette di interagire con l'host (ovvero il kernel Linux) tramite l'interfaccia Linux "docker0".

Per cui è presente una dipendenza tra il networking fornito da Docker (BRIDGE) e il networking del SO (interfaccia "docker0"). Verifichiamo installando il "brctl" tool con il comando: "apt-get install bridge-utils" e poi eseguendo: "brctl show".

Volendo essere più chiari possiamo affermare che la modalità "bridge" effettua principalmente due task fondamentali:

- Si comporta come una switch mettendo in comunicazione i container appartenenti alla stessa interfaccia bridge.
- Tramite il meccanismo di NAT fornito dall'host ci permette di comunicare verso il mondo esterno tramite il mapping delle porte già analizzato nelle precedenti sezioni.

### Comunicazione tra due container

```
docker run -it --name containerA ubuntu bash
apt-get update
apt-get install net-tools
ifconfig
```

```
root@792d26503e9b:/# ifconfig
eth0 Link encap:Ethernet HWaddr 02:42:ac:12:00:02
 inet addr:172.18.0.2 Bcast:172.18.0.255 Mask:255.255.255.0
 UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
 RX packets:3217 errors:0 dropped:2 overruns:0 frame:0
 TX packets:2007 errors:0 dropped:0 overruns:0 carrier:0
 collisions:0 txqueuelen:0
 RX bytes:30351103 (30.3 MB) TX bytes:137625 (137.6 KB)
```

Istanziamo un altro container:

```
docker run -it --name containerB ubuntu bash
apt-get update
apt-get install net-tools
ifconfig
```

```
root@fafe32a70414:/# ifconfig
eth0 Link encap:Ethernet HWaddr 02:42:ac:12:00:03
 inet addr:172.18.0.3 Bcast:172.18.0.255 Mask:255.255.255.0
 UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
 RX packets:4148 errors:0 dropped:0 overruns:0 frame:0
 TX packets:2652 errors:0 dropped:0 overruns:0 carrier:0
 collisions:0 txqueuelen:0
 RX bytes:30410716 (30.4 MB) TX bytes:211159 (211.1 KB)
```

La situazione è quindi la seguente:

| \$ docker ps | CONTAINER ID | IMAGE  | COMMAND | CREATED       | STATUS       | PORTS |
|--------------|--------------|--------|---------|---------------|--------------|-------|
|              | NAMES        |        |         |               |              |       |
|              | fafe32a70414 | ubuntu | "bash"  | 2 minutes ago | Up 2 minutes |       |
|              | containerB   |        |         |               |              |       |
|              | 5b3e7e249e4e | ubuntu | "bash"  | 3 minutes ago | Up 3 minutes |       |
|              | containerA   |        |         |               |              |       |

Per dimostrare che i due container si trovano sulla stessa sottorete di default (bridge) eseguire il comando "docker inspect network bridge":

```

"IPAM": {
 "Driver": "default",
 "Options": null,
 "Config": [
 {
 "Subnet": "172.18.0.1/24",
 "Gateway": "172.18.0.1"
 }
]
},
"Internal": false,
"Attachable": false,
"Ingress": false,
"ConfigFrom": {
 "Network": ""
},
"ConfigOnly": false,
"Containers": {
 "5b3e7e249e4ea1435e4362ea5b485a9946236deaac57580a816b6612d12dd5ff": {
 "Name": "containerA",
 "EndpointID": "fbfd87a3fa06957295888766f04d2e967b4f77759c6b7469721a155527954738",
 "MacAddress": "02:42:ac:12:00:02",
 "IPv4Address": "172.18.0.2/24",
 "IPv6Address": ""
 },
 "fafef32a70414ca8ff94379937c8b62031e0836203b848d6f867568515313d440": {
 "Name": "containerB",
 "EndpointID": "3ece8318fe621ac9b037481fceaa58488be075b9c70489159c666b00e0cd11573",
 "MacAddress": "02:42:ac:12:00:03",
 "IPv4Address": "172.18.0.3/24",
 "IPv6Address": ""
 }
}

```

Possiamo osservare la sottorete "172.18.0.1" e possiamo vedere inoltre i due container appartenenti alla sottorete citata prima. I due container possono quindi comunicare tra di loro. Per dimostrarlo colleghiamoci ad uno dei due container, per esempio "containerA", e digitiamo:

```

apt-get install iputils-ping //questo per installare l'utility per fare il ping
ifconfig //verifichiamo quale sia il nostro ip e pinghiamo l'altro container
ping 172.18.0.3

```

Il risultato sarà il seguente:

```

root@5b3e7e249e4e:/# ping 172.18.0.3
PING 172.18.0.3 (172.18.0.3) 56(84) bytes of data.
64 bytes from 172.18.0.3: icmp_seq=1 ttl=64 time=0.180 ms
64 bytes from 172.18.0.3: icmp_seq=2 ttl=64 time=0.114 ms
64 bytes from 172.18.0.3: icmp_seq=3 ttl=64 time=0.108 ms

```

## Comunicazione verso l'esterno (NAT)

Di default i container possono interagire con la rete globale (Internet) grazie ad un meccanismo di natting: l'ip privato del container viene mascherato in un ip pubblico da parte dell'host, così facendo dall'esterno si vedrà solo l'indirizzo ip pubblico dell'host e mai l'indirizzo ip privato del container.

## Esposizione|Mapping delle porte

Si vuole eseguire un container, mappare una porta del container con una dell'host e provare quindi a connetterci a quella porta specificata nell'host e verificare se il traffico viene ridirezionato nel container.

```

docker run -d --name webserver1 -p 8082:80 nginx

```

```
docker ps
```

```
$ docker run -d --name webserver1 -p 8082:80 nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
214ca5fb9032: Pull complete
66eec13bb714: Pull complete
17cb812420e3: Pull complete
56fbff79cae7a: Pull complete
c4547ad15a20: Pull complete
d31373136b98: Pull complete
Digest: sha256:2d17cc4981bf1e22a87ef3b3dd20fbb72c3868738e3f307662eb40e2630d4320
Status: Downloaded newer image for nginx:latest
688a023ebfe2f0920cc1d0dd14d13107a7572a885c284bb277861d7a78ed3d7c
$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS
S
688a023ebfe2 nginx "/docker-entrypoint..." 12 seconds ago Up 11 seconds 0.0.0.0:8082->80/tcp webs
erver1
```

La dicitura "0.0.0.0:8082→80/tcp" sta a significare che tutto il traffico proveniente dalla porta 8082 dall'host viene indirizzato nella porta 80 (protocollo tcp) del container.

### Creazione di una rete

Procediamo a creare una nuova rete, che sarà di tipo bridge (default) e avrà una sottorete assegnata e un gateway:

```
docker network create reteTest
```

verifichiamo l'effettiva creazione della rete con il comando:

```
docker network ls
```

```
$ docker network ls
NETWORK ID NAME DRIVER SCOPE
18313185a26c bridge bridge local
fa054a9af353 host host local
f50397115ef2 none null local
41518a23261d reteTest bridge local
$ ||
```

Vediamo la sottorete assegnata alla rete appena creata con il comando:

```
docker network inspect reteTest
```

```
$ docker network inspect reteTest
[
 {
 "Name": "reteTest",
 "Id": "41518a23261d7cab7f238750bc2046e0c9a24de25ff1b89159c9828da1b1e264",
 "Created": "2022-05-25T17:07:07.390903213Z",
 "Scope": "local",
 "Driver": "bridge",
 "EnableIPv6": false,
 "IPAM": {
 "Driver": "default",
 "Options": {},
 "Config": [
 {
 "Subnet": "172.17.0.0/16",
 "Gateway": "172.17.0.1"
 }
]
 },
 "Internal": false,
 "Attachable": false,
 "Ingress": false,
 "ConfigFrom": {
 "Network": ""
 },
 "ConfigOnly": false,
 "Containers": {},
 "Options": {},
 "Labels": {}
 }
]
```

Adesso creiamo un nuovo container e anziché assegnarlo alla sottorete di default (bridge) la assegnamo a reteTest.

```
docker run -it --name ubuntu1 --network reteTest ubuntu bash
```

Possiamo vedere che l'ip del container ubuntu1 è dentro la sottorete reteTest:

```
docker inspect ubuntu1
```

```
"Networks": {
 "reteTest": {
 "IPAMConfig": null,
 "Links": null,
 "Aliases": [
 "fe480ba88b18"
],
 "NetworkID": "41518a23261d7cab7f238750bc2046e0c9a24de25ff1b89159c9828da1b1e264",
 "EndpointID": "ac493418f663611f048abbcf6a16f5cae1c99dfa51e1de2d62ae367ca34a6a6e",
 "Gateway": "172.17.0.1",
 "IPAddress": "172.17.0.2", | ←
 "IPPrefixLen": 16,
 "IPv6Gateway": "",
 "GlobalIPv6Address": "",
 "GlobalIPv6PrefixLen": 0,
 "MacAddress": "02:42:ac:11:00:02",
 "DriverOpts": null
 }
}
```

Cosa importante è che tutti i container all'interno della sottorete reteTest potranno comunicare tra loro, ma non potranno comunicare con qualsiasi altro container presente in un'altra sottorete (a meno che non venga fatto esplicitamente).

Infatti se istanziamo un nuovo container senza esplicitare la sottorete (che quindi per default andrà in bridge):

```
docker run -it --name ubuntu2 ubuntu bash
```

Vediamo l'appartenenza alla sottorete:

```
docker inspect ubuntu2
```

```
"Networks": {
 "bridge": {
 "IPAMConfig": null,
 "Links": null,
 "Aliases": null,
 "NetworkID": "18313185a26cb405ff312d4673b5c2b30e1b813ce39d42ae13825f6af5988970",
 "EndpointID": "307807153a95086ed2c52c4f53438ca75fd51185bbcc90652aef799605a955aa",
 "Gateway": "172.18.0.1", |
 "IPAddress": "172.18.0.2", |
 "IPPrefixLen": 24,
 "IPv6Gateway": "",
 "GlobalIPv6Address": "",
 "GlobalIPv6PrefixLen": 0,
 "MacAddress": "02:42:ac:12:00:02",
 "DriverOpts": null
 }
}
```



L'ip è differente. Proviamo a vedere se riescono a comunicare facendo un ping dal container2 al container1 (ricordiamo: appartenenti a sottoreti differenti). Installiamo prima l'utilità per il ping:

```
apt-get update
apt-get install iputils-ping
ping 172.17.0.2
```

```
root@d9b051560103:/# ping 172.17.0.2
PING 172.17.0.2 (172.17.0.2) 56(84) bytes of data.
^C
--- 172.17.0.2 ping statistics ---
254 packets transmitted, 0 received, 100% packet loss, time 263129ms
```

Come è possibile notare il ping non riceve risposta. Questo poiché appartengono a sottoreti differenti e risultano quindi isolate.

**Attenzione:** la rete di default bridge non supporta la risoluzione dei nomi dns. Le reti custom di tipo "bridge" (come quella creata prima) invece supportano la risoluzione dei nomi tramite un dns server interno a docker.

## Rete None e Host

### 1. None

La rete *none* non viene inserita, installata e assegnato alcun indirizzo ip. Verifichiamo creando un container e assegnandolo alla sottorete *none*:

```
docker run -it --name ubuntuNone --network none ubuntu bash
```

Il comando:

```
apt-get update
```

non dovrebbe funzionare, infatti:

```

root@578c5a4ceafc:/# apt-get update
Err:1 http://security.ubuntu.com/ubuntu xenial-security InRelease
 Temporary failure resolving 'security.ubuntu.com'
Err:2 http://archive.ubuntu.com/ubuntu xenial InRelease
 Temporary failure resolving 'archive.ubuntu.com'
Err:3 http://archive.ubuntu.com/ubuntu xenial-updates InRelease
 Temporary failure resolving 'archive.ubuntu.com'
Err:4 http://archive.ubuntu.com/ubuntu xenial-backports InRelease
 Temporary failure resolving 'archive.ubuntu.com'
Reading package lists... Done
W: Failed to fetch http://archive.ubuntu.com/ubuntu/dists/xenial/InRelease Temporary fail
W: Failed to fetch http://archive.ubuntu.com/ubuntu/dists/xenial-updates/InRelease Tempor
W: Failed to fetch http://archive.ubuntu.com/ubuntu/dists/xenial-backports/InRelease Temp
W: Failed to fetch http://security.ubuntu.com/ubuntu/dists/xenial-security/InRelease Temp
W: Some index files failed to download. They have been ignored, or old ones used instead.

```

È anche possibile verificarlo anche mediante il comando:

```
docker inspect ubuntuNone
```

Come è possibile osservare la sezione ipaddress è vuota:

```

"Networks": {
 "none": {
 "IPAMConfig": null,
 "Links": null,
 "Aliases": null,
 "NetworkID": "f50397115ef29cd899b8b32ae5430ce59b40f5a049e88ed18c407e5fd1fc269b",
 "EndpointID": "b63e035d68119de470bab3f3697fcb86909d58e518f982253e62edf260f4e6ba",
 "Gateway": "",
 "IPAddress": "", [red box]
 "IPPrefixLen": 0,
 "IPv6Gateway": "",
 "GlobalIPv6Address": "",
 "GlobalIPv6PrefixLen": 0,
 "MacAddress": "",
 "DriverOpts": null
 }
}

```

Quindi a livello di networking è completamente disconnesso.

## 2. Host

Sfrutta direttamente le interfacce dell'host senza passare dall'intermediazione del bridge creato da docker.

```
docker run -it --name ubuntuHost --network host ubuntu bash
```

Installiamo l'utilità ifconfig per verificare se si sta condividendo le interfacce dell'host:

```
apt-get update
apt-get install net-tools
```

```

root@host01:/# ifconfig
docker0 Link encap:Ethernet HWaddr 02:42:df:59:57:7c
 inet addr:172.18.0.1 Bcast:172.18.0.255 Mask:255.255.255.0
 UP BROADCAST MULTICAST MTU:1500 Metric:1
 RX packets:0 errors:0 dropped:0 overruns:0 frame:0
 TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
 collisions:0 txqueuelen:0
 RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

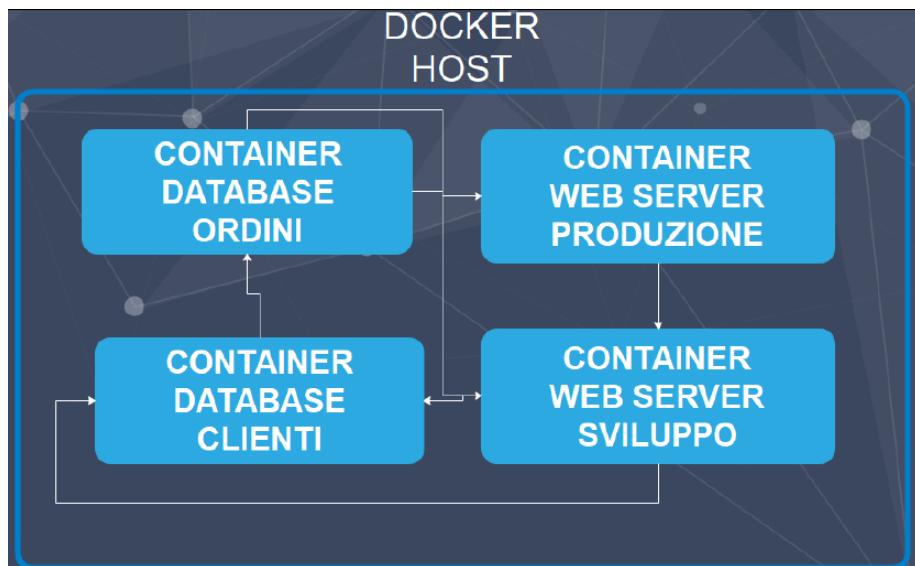
enp0s3 Link encap:Ethernet HWaddr 02:42:0a:00:00:16
 inet addr:10.0.0.22 Bcast:11.255.255.255 Mask:255.0.0.0
 inet6 addr: fe80::33a2:b175:4fa8:8f19/64 Scope:Link
 UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
 RX packets:4214 errors:0 dropped:0 overruns:0 frame:0
 TX packets:2501 errors:0 dropped:0 overruns:0 carrier:0
 collisions:0 txqueuelen:1000
 RX bytes:30430336 (30.4 MB) TX bytes:208467 (208.4 KB)

lo Link encap:Local Loopback
 inet addr:127.0.0.1 Mask:255.0.0.0
 inet6 addr: ::1/128 Scope:Host
 UP LOOPBACK RUNNING MTU:65536 Metric:1
 RX packets:0 errors:0 dropped:0 overruns:0 frame:0
 TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
 collisions:0 txqueuelen:1000
 RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

```

#### 4.9 Docker compose, [1]

La sezione "networking di base" ci ha mostrato come possono comunicare tra loro più container. Nelle **architetture a microservizi** avere più container che interagiscono tra loro è lo standard.



Ricordiamoci che la best practice è avere più container dove ognuno di questi esegue una singola applicazione o servizio.

Lo strumento "docker compose" ci permette di definire ed eseguire scenari applicativi complessi che richiedono l'utilizzo di più container.

Il principio di funzionamento di Docker Compose è il seguente:

1. Definizione del "docker compose" file in formato ".yml" che ci permette di definire e configurare i servizi relativamente alla nostra applicazione.

2. Esecuzione dei servizi (e quindi dei container) definiti all'interno del "docker compose" file.
3. Possibilità di effettuare lo "scaling" dei servizi in base al carico di lavoro e alle proprie necessità.
4. Arresto dei servizi (e quindi dei container) definiti all'interno del "docker compose" file.

Il componente "docker compose" è già compreso nell'installazione di docker nei sistemi windows e mac. Su linux è necessario installarlo in un secondo momento.

### Struttura del docker compose file

Il file "docker compose.yml" si definisce utilizzando il formato YAML. È necessario definire quali saranno i servizi (container) da eseguire e quali caratteristiche questi dovranno avere. Di seguito un esempio di docker compose file:

```
version: "3.9" # optional since v1.27.0
services:
 web:
 build: .
 ports:
 - "8000:5000"
 volumes:
 - ./code
 - logvolume01:/var/log
 links:
 - redis
 redis:
 image: redis
volumes:
 logvolume01: {}
```

```
Commands:
 build Build or rebuild services
 bundle Generate a Docker bundle from the Compose file
 config Validate and view the Compose file
 create Create services
 down Stop and remove containers, networks, images, and volumes
 events Receive real time events from containers
 exec Execute a command in a running container
 help Get help on a command
 images List images
 kill Kill containers
 logs View output from containers
 pause Pause services
 port Print the public port for a port binding
 ps List containers
 pull Pull service images
 push Push service images
 restart Restart services
 rm Remove stopped containers
 run Run a one-off command
 scale Set number of containers for a service
 start Start services
 stop Stop services
 top Display the running processes
 unpause Unpause services
 up Create and start containers
 version Show the Docker-Compose version information
```

## Creazione del Docker Compose File

Creiamo un dockerfile dove la parte database è svolta da un container con l'immagine nginx e la parte di web server dall'immagine redis:

```
mkdir composeFile
touch docker-compose.yml
vi docker-compose.yml
```

Scrivere:

```
version: "3"

services:
 web:
 image: nginx

 database:
 image: redis
```

Adesso validiamo il tutto digitando:

```
docker-compose config
```

Adesso procediamo con l'[esecuzione](#) di questi servizi utilizzando il comando:

```
docker-compose up -d
```

(-d indica che lo si avvia in modalità detach)

```
$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
0e83cf2e3dae nginx "/docker-entrypoint..." About a minute ago Up About a minute 80/tcp composefile_web_1
9828e1e8088b redis "docker-entrypoint.s..." About a minute ago Up About a minute 6379/tcp composefile_database_1
```

Adesso procediamo [stopando](#) questi servizi utilizzando il comando:

```
docker-compose stop
```

```
$ docker-compose stop
stopping composefile_web_1 ... done
Stopping composefile_database_1 ... done
```

## Opzione ports, [1]

Questo comando espone semplicemente le porte del container. Sintassi:

```
[HOST:]CONTAINER[/PROTOCOL] where:
```

- `HOST` is `[IP:]`(`port` | `range`)
- `CONTAINER` is `port` | `range`
- `PROTOCOL` to restrict port to specified protocol. `tcp` and `udp` values are defined by the specification, Compose implementations MAY offer support for platform-specific protocol names.

Esempio:

```
ports:
 - "3000"
 - "3000-3005"
 - "8000:8000"
 - "9090-9091:8080-8081"
 - "49100:22"
```

```
- "127.0.0.1:8001:8001"
- "127.0.0.1:5000-5010:5000-5010"
- "6060:6060/udp"
```

Ritornando all'esempio di prima modificando il dockerfile:

```
$ mkdir composeFile
$ touch docker-compose.yml
$ vi docker-compose.yml
$ cat docker-compose.yml
version: "3"
services:
 web:
 image: nginx
 ports:
 - 8090:80/tcp
 database:
 image: redis
```

```
docker-compose config
docker-compose up
```

Esempi di funzionamento:

```
$ wget localhost:8090
--2022-05-27 18:09:37-- http://localhost:8090/
Resolving localhost (localhost)... ::1, 127.0.0.1
Connecting to localhost (localhost) |::1|:8090... connected.
HTTP request sent, awaiting response... 200 OK
Length: 615 [text/html]
Saving to: 'index.html'

index.html 100%[=====] 615 --.-KB/s in 0s

2022-05-27 18:09:37 (60.8 MB/s) - 'index.html' saved [615/615]

$ curl localhost:8090
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
nginx.org.

Commercial support is available at
nginx.com.</p>

<p>Thank you for using nginx.</p>
</body>
</html>
$ []
```

## Opzione scale

Lo scaling consiste nell'esecuzione/eliminazione dinamica dei servizi in base al carico di lavoro necessario.

```
mkdir composeFile
touch docker-compose.yml
vi docker-compose.yml
```

Scrivere:

```
version: "3"
services:
 frontend:
 image: nginx
 deploy:
 mode: replicated
 replicas: 6
```

Infine eseguire il comando:

```
docker-compose --compatibility up
```

Per stopparli tutti in una volta sola basta il comando:

```
docker-compose down
```

#### 4.10 Docker Swarm, [1]

Docker swarm è un componente che ci permette la gestione dei cluster e l'orchestrazione dei container su scenari multi-host.

Uno "swarm" (in italiano "sciame") consiste in più host docker che si eseguono in modalità "swarm mode" e possono assumere i ruoli di "manager" o di "worker".

Nello specifico un host docker può assumere i seguenti ruoli:

- Manager
- Worker
- Entrambi.

Tutto parte dalla creazione di un servizio con specifiche caratteristiche. Docker swarm avrà il compito di mantenere lo stato desiderato.

Uno dei vantaggi legati all'utilizzo di docker swarm è la possibilità di modificare la configurazione di un servizio senza necessità di riavviare il servizio come nel caso dei container standalone.

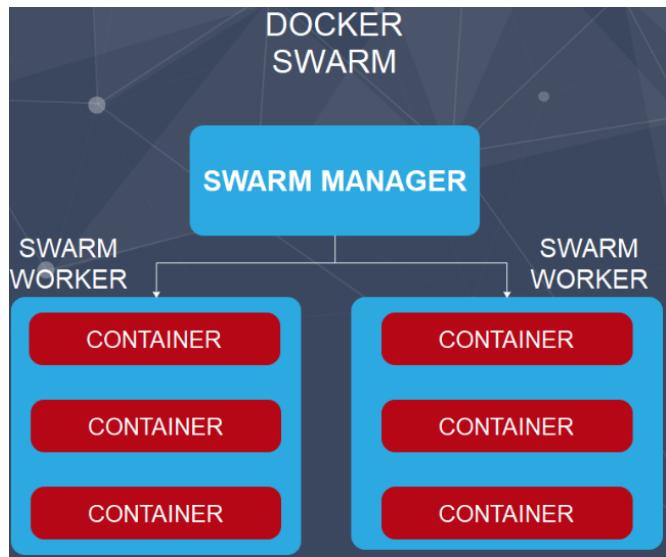
Un nodo è un'istanza di Docker che partecipa in uno "swarm". E' possibile eseguire uno o più nodi su un server fisico oppure su un server in cloud.

In pratica per effettuare il deploy di un'applicazione su uno "swarm" è necessario definire un servizio e assegnare la sua gestione ad un "nodo manager". Sarà poi il nodo manager ad assegnare i "task" ai nodi worker.

Il nodo manager di default è sia manager sia worker. E' possibile renderlo solo manager.

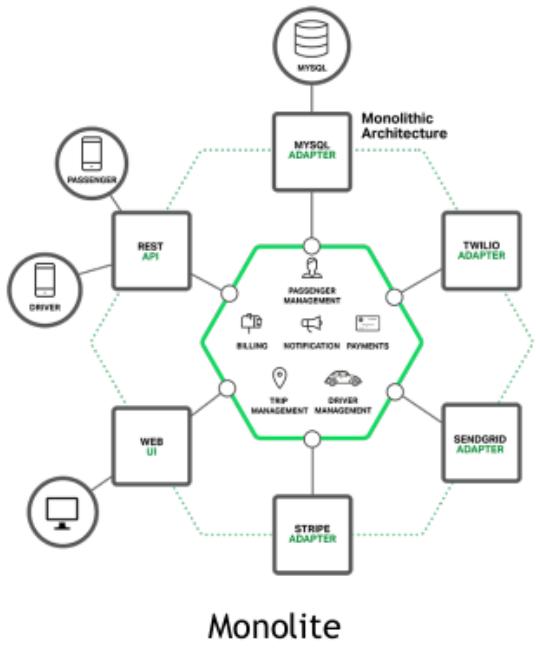
Un servizio è la definizione di un "task" eseguito dal nodo manager o dai nodi worker.

Quando si crea un servizio si definisce l'immagine del container e quali comandi devono essere eseguiti all'interno del container.

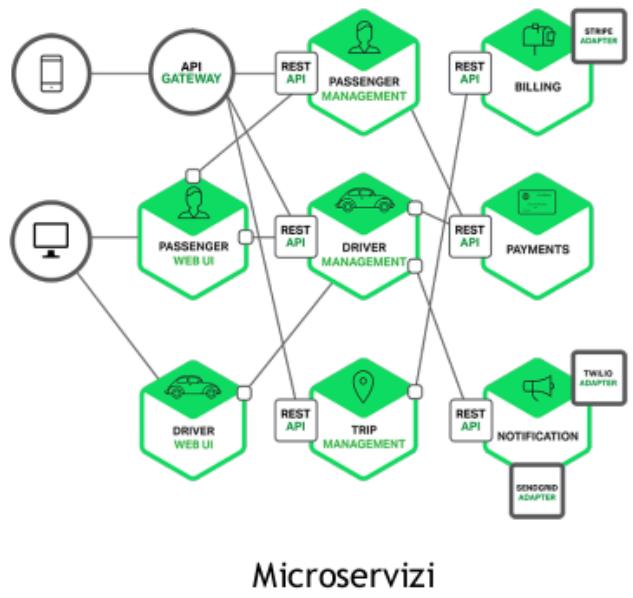


## 5. Microservizi, [1]

# Monolite vs Microservizi



Monolite



Microservizi

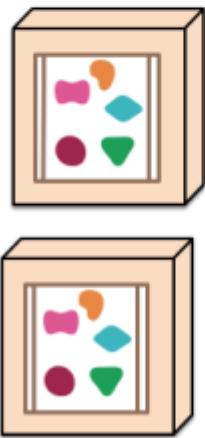
### Svantaggi Monolite:

- Al crescere della dimensione del sistema la produttività degli sviluppatori diminuisce (complessità);
- Difficile introdurre l'uso di nuove tecnologie;
- Se una parte del sistema va in errore, l'intero sistema ne risente (manca fault segregation);
- Se una parte del sistema viene aggiornata, bisogna fare il deploy dell'intera applicazione (EAR/WAR/JAR);
- Se una parte del sistema consuma troppe risorse, l'intera applicazione deve essere replicata (problemi di scalabilità).

# Scalabilità e ottimizzazione risorse

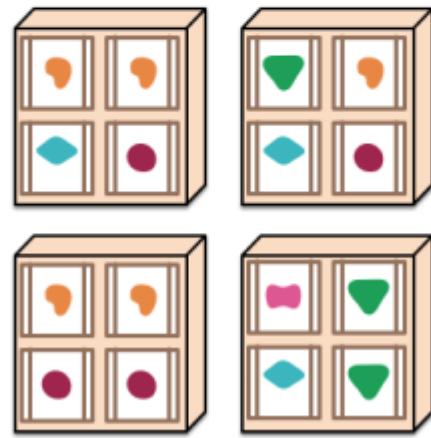
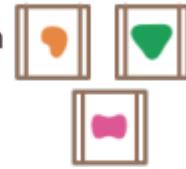
Un'applicazione monolitica esegue tutte le funzionalità in un singolo processo...

... e scala replicando il "monolite" su più server.



Un'architettura a microservizi esegue ogni funzionalità in un servizio separato...

... e scala distribuendo i servizi tra più server, replicandoli se necessario.



**Replico solo le parti che richiedono più risorse**

[vedi slide]