Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Ansible:
utilizzi in sistemi distribuiti

Relatore:
Chiar.mo Prof.
COSIMO LANEVE

Presentata da:
DOMENICO CARAVAGGIO

Sessione II
Anno Accademico 2019/2020

Alla mia famiglia ...

Indice

Elenco delle figure

Sommario

Ad oggi lo scenario delle infrastrutture è dominato dalla virtualizzazione di macchine dei più disparati SO, che vengono utilizzati per creare applicazioni composte da piccoli nodi indipendenti e altamente disaccoppiati. Questo tipo di approccio fa un uso intensivo di macchine e di rete che vengono virtualizzate.

Il lavoro presentato in questa tesi è intitolato "*Ansible: utilizzi in sistemi distribuiti*", mirando ad una problematica che spesso si presenta in sistemi distribuiti, causata da macchine non eterogenee tra loro, e come utilizzare un software presente per configurazioni ed eseguire istruzioni su di esse. Ansible infatti è uno dei tanti software per gestire infrastrutture distribuite.

L'elaborato è diviso in cinque parti: nella prima parte si analizza la disciplina della virtualizzazione, esponendo le nozioni che hanno portato al suo utilizzo. Nei successivi paragrafi, vengono approfondite le differenze che emergono nei vari sottotipi di virtualizzazione analizzati e quali differenze caratterizzano gli stessi.

Nella seconda parte si definiscono i metodi di sviluppo e il ciclo di vita del software. In particolare il sottocapitolo 2.2 si occupa in modo dettagliato del ciclo di vita DevOps.

Nella terza parte si determinano cosa sono le configurazioni, esaminando i metodi per la loro distribuzione. In particolare nel sottocapitolo 3.2 si parla di Infrastruttura come codice.

Nella quarta parte, si analizza cos'è Ansible, le motivazioni per tale strumento, come utilizzarlo e le varie tecnologie che permettono di eseguirlo sulle

macchine.

Nella quinta parte, si esaminano e confrontano strumenti simili ad Ansible, individuando i punti di forza di uno o dell'altro strumento.

Infine sono stati realizzati alcuni test per sperimentare Ansible.

Capitolo 1

Virtualizzazione

Questo capitolo definisce il concetto di virtualizzazione e analizza le motivazioni che hanno portato alla sua introduzione. Vengono anche esposte le tecniche usate per la virtualizzazione.

1.1 Dominio della virtualizzazione

I principali motivi che hanno portato ad introdurre la virtualizzazione sono: richiesta di scalabilità delle risorse, costi di manutenzione hardware, maggior affidabilità dei sistemi operativi. Vediamoli in sintesi:

- **Affidabilità:** utilizzare macchine virtuali permette di incapsulare servizi che risultano più critici rispetto ad altri. In caso di problemi si può circoscrivere la problematicità e non compromettere tutto il sistema. Inoltre si garantisce una maggiore sicurezza a fronte di guasti, che possono essere prevenuti con backup di intere macchine virtuali, le quali impiegano meno risorse rispetto alla controparte fisica, con la conseguenza che un disaster recovery ha un basso impatto e può essere immediato.

- **Scalabilità:** a differenza delle macchine fisiche dove gestire le risorse significa mettere mano all'hardware, cioè spostare fisicamente le risorse che si vogliono dedicare, con tutti i rischi del caso, con l'uso di macchine virtuali, tramite dei semplici file di configurazione le risorse possono essere scalate a piacere, ovviamente nei limiti della macchina fisica sulla quale vengono virtualizzate.

- **Manutenibilità:** utilizzando meno macchine fisiche i costi di manutenzione calano vertiginosamente paragonati al numero di macchine fisiche che bisognerebbe impiegare: gli investimenti in nuovo hardware inferiore, ed un risparmio in energia elettrica e spazio.

- **Compatibilità:** uno degli aspettati sicuramente positivi che porta a questa scelta è la compatibilità più ampia, poiché avere la possibilità di virtualizzare qualsiasi sistema permette di avere una compatibilità con le versioni precedenti, cosa estremamente difficile quando si modificano dei software.

1.2 Metodi di Virtualizzazione

Per virtualizzazione, si intende un meccanismo che permette di eseguire più sistemi operativi sulla stessa macchina. Le tecniche più diffuse oggi sono due, quella a livello hardware e la virtualizzazione a livello del sistema operativo [2, 3]. Vedremo più approfonditamente queste due tecniche, che hanno differenze sostanziali.

1.2.1 Virtualizzazione Hardware

Consiste nell'emulare un certo hardware, quindi ha uno strato software chiamato Virtual Machine Manager (VMM) o Hypervisor, con il compito di

virtualizzare tutto l'hardware della macchina, occupandosi di allocare risorse e memoria per ogni macchina, così che ognuna abbia ha il proprio Kernel, file binari, librerie e applicazioni. Questo inserisce ovviamente un overhead, ma ci permette sicuramente di isolare le macchine. L'esempio in Figura 1.1 è solo uno dei due modi per utilizzare un Hypervisor il vantaggio di questa implementazione è quella di avere l'Hypervisor di dimensioni contenute ed a stretto contatto con l'Hardware, chiamati hypervisor bare metal.

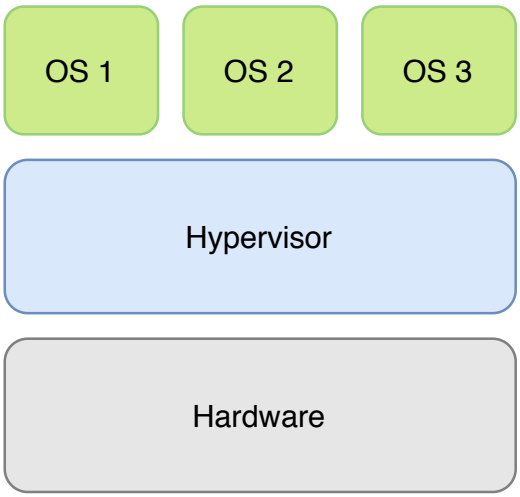

Figura 1.1: Esempio di Hypervisor

Il secondo tipo di Hypervisor utilizzato per la virtualizzazione Hardware è rappresentato in Figura 1.2. Questa implementazione ha il vantaggio di risiedere su un sistema operativo host, permettendo un controllo maggiore, ma ciò comporta che un guasto al sistema operativo host si ripercuota su tutti i sistemi guest.

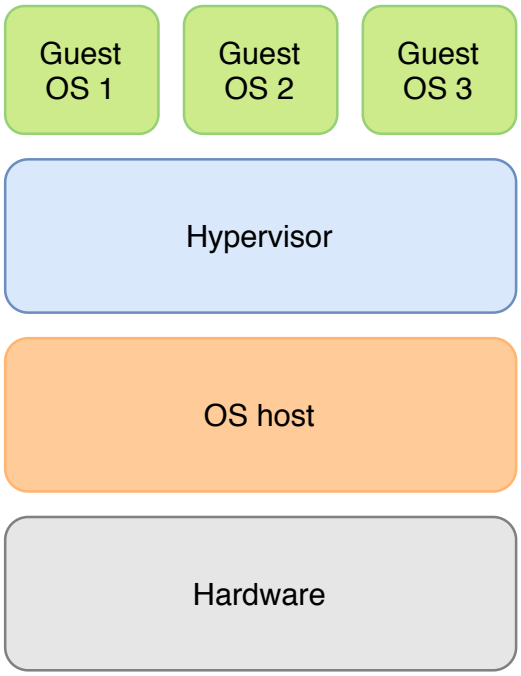

Figura 1.2: Altro esempio di Hypervisor

1.2.2 Virtualizzazione System-Level

Questo tipo di virtualizzazione si basa sul concetto di container, cioè un ambiente isolato che include tutte le risorse, librerie e configurazioni, necessari per l'esecuzione di un'applicazione. Questo implica che, a differenza dell'altra precedentemente citata, con la virtualizzazione di un container il kernel sia uno solo, cioè quello della macchina host, apportando il vantaggio di avere meno overhead e quindi notoriamente definita più leggera e performante, poiché tutte le System call on devono essere sottoposte a emulazione, ma vengono eseguite direttamente dalla macchina host.

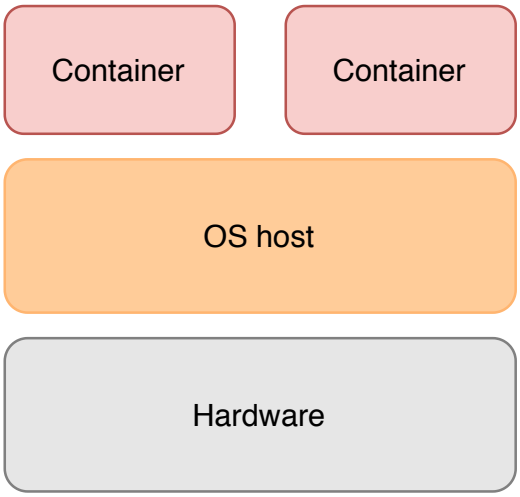

Figura 1.3: Esempio di Virtualizzazione System-Level

Questo livello di astrazione viene definito da un proprio file system permesso grazie ad alcune funzioni proprie del Kernel Linux come Namespace e Cgroups.

Capitolo 2

DevOps

Questo capitolo analizza da dove si è iniziato a sviluppare il software. Vengono anche esposte le tecniche usate e, in particolar modo, viene introdotto il modello DevOps.

Si definisce DevOps un insieme di pratiche che hanno lo scopo di ridurre il tempo tra il commit di un cambiamento a un sistema e l'effettiva messa in produzione del cambiamento, assicurando nel frattempo un alto livello di qualità [4]. DevOps è l'insieme di due parole: sviluppo di software (Development) e operazioni IT (Operations).

2.1 Modelli di sviluppo

Per modello di sviluppo si intende una divisione del ciclo di vita del software in più fasi, al fine di migliorarne lo sviluppo, il design e la manutenibilità. Ogni fase, dunque, produce un certo insieme di artefatti, che vengono spesso utilizzati come punto di partenza per le fasi successive. Introdurremo due modelli rilevanti al fine di introdurre le motivazioni dietro il pensiero DevOps: i modelli a cascata e i modelli agili.

2.1.1 Modello a cascata

Il ciclo di vita a cascata (Waterfall) è uno dei primi ad essere stato sviluppato nella storia dell'IT. Divide il ciclo di vita in cinque fasi o passi distinti in cascata, che producono degli artefatti provenienti dalle fasi successive, senza la possibilità di tornare indietro a fasi già concluse. Più nello specifico si individuano le seguenti fasi:

Figura 2.1: Rappresentazione del modello a cascata

- **Analisi dei requisiti:** si parla di analisi dei requisiti in ingegneria del software quando si interroga il cliente su quali requisiti devono essere soddisfatti dal software sviluppato. Come ovviamente si può capire è la prima frase di sviluppo software, la quale si conclude con la stesura dettagliata dei requisiti che descrive le funzionalità del nuovo software.

- **Design:** talvolta detta anche progetto, è la fase del ciclo di vita del software che parte dalla lettura degli artefatti precedentemente creati, infatti definisce quali requisiti saranno soddisfatti, entrando nel merito della struttura che dovrà essere data al sistema da realizzare,e in particolare la sua suddivisione in moduli e le relazioni fra di essi.

- **Sviluppo:** il software viene creato secondo lo schema architetturale e il linguaggio di programmazione stabilito.

- **Testing:** si riferisce ad una serie di operazioni messe in atto al fine di verificare il corretto funzionamento dal software, prima che venga destinato all'utilizzo, detto anche produzione. Infatti si eseguono test

di vario genere per assicurarsi che il prodotto sia privo di errori percettibili prima della consegna al cliente.

- **Deployment:** in questo passo il software viene messo in produzione ed è compito dell'azienda produttrice mantenerlo funzionante; comprende tutte le attività volte a migliorare, estendere e correggere il sistema nel tempo.

Come si è capito, il modello a casata presenta grosse problematiche date dalla sua rigidità. La possibilità che il cliente presenti nuovi requisiti ha l'effetto di far ricominciare da capo l'intero ciclo di sviluppo. Se dopo la consegna si scoprissero funzionalità implementate in maniera non corretta o che non soddisfano il cliente, ancora una volta si dovrebbero ripetere tutte le fasi dall'analisi. In genere si è visto che è molto difficoltoso rendere i requisiti stabili per tutta la durata dello sviluppo, se non altro sarebbe utile adottare un modello di sviluppo che favorisca maggiormente la comunicazione tra le parti e che sia più adatto ai cambiamenti continui dei requisiti.

2.1.2 Modello agile

Dal modello a cascata ci si è resi conto della necessità di un modello più flessibile, con consegne molto più veloci e con un feedback continuo da parte del cliente. Il modello agile nasce proprio con questi obbiettivi.

Seguendo le metodologie agili, gli sviluppatori periodicamente si confrontano con il cliente e presentano un prototipo, con il duplice scopo di rendere il cliente consapevole e di verificare la comprensione dei requisiti, venendo così a cerare degli incontri a breve distanza uno dall'altro (in genere 2-3 settimane). Questi cicli prendono il nome di sprint.

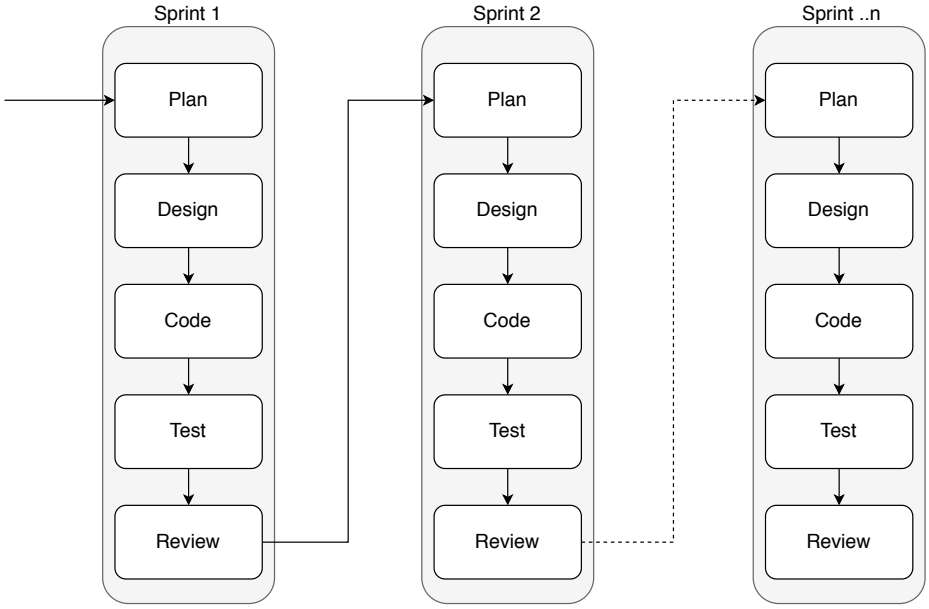

Figura 2.2: Rappresentazione del modello agile

Ogni sprint si divide approssimativamente nelle seguenti fasi:

- **Plain:** in questa fase si decide, insieme al cliente, quali sono le funzionalità che verranno implementate durante uno sprint.

- **Design:** come per l'approccio a cascata, in questa fase viene definita in maniera dettagliata la struttura.

- **Code:** si sviluppano le funzionalità, avvalendosi dei prototipi qualora le specifiche siano poco chiare.

- **Test:** nella fase di test, verranno eseguite una serie di azioni atte al verificare i requisiti.

- **Review:** verrà fatto intervenire il cliente che, qualora riterrà opportuno, indicherà le problematiche riscontrate e che potranno essere mi-

gliorate nel successivo sprint.

Le metodologie agili rappresentano un netto passo in avanti a confronto del metodo a cascata, ma hanno anch'esse delle problematiche ancora presenti. Infatti, anche gli approcci agili non fanno interagire a stretto contatto il team degli operatori, o sistemisti, che devono mantenere ed installare i software in produzione; questo presenta delle criticità poiché, non avendo una conoscenza spinta del codice, e nell'eventualità di librerie mancanti, software di terze parti non installati riscontreranno dei problemi che potrebbero essere risolti solo con una collaborazione stretta tra sviluppatori del codice e IT operations.

2.2 Il modello DevOps

Finora abbiamo analizzato le pratiche che sono utilizzate senza tener conto delle problematiche che verrebbero a crearsi in produzione, quindi senza una collaborazione stretta tra i team di developers e operations. Il modello DevOps viene presentato come una sinergia tra questi, che vengono uniti per creare un unico ciclo diviso in due stadi, specifici per area di competenza, il modello però prende a piene mani le politiche che si adoperano del modello agile. Come si vede in Figura 2.3 sono presenti otto stadi; i primi descrivono le attività svolte del team Dev, quindi riguardano pratiche già viste nel modello agile, in breve:

- **Plain:** pianificazione delle prossime feature da implementare.
- **Code:** sviluppo delle nuove feature, utilizzando tool di versionamento del codice.
- **Build:** compilazione delle singole componenti.
- **Test:** testing dei vari componenti.

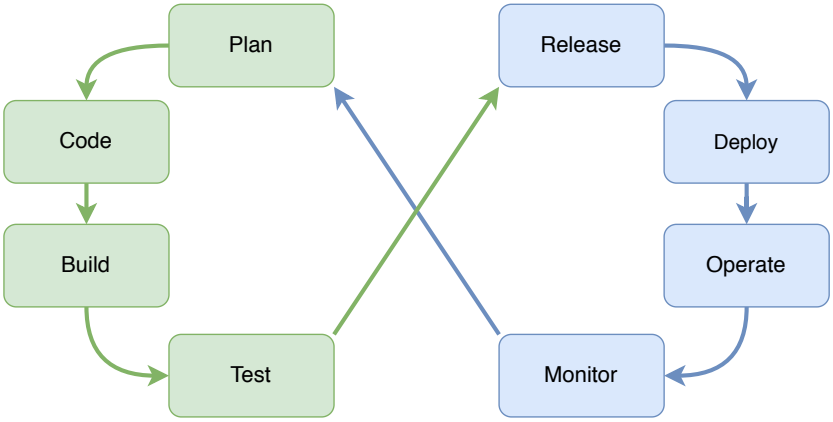

Figura 2.3: Rappresentazione del modello DevOps

Le restanti quattro sono fasi riferite al team Ops, che si riassumono in:

- **Release:** è quella fase dove vengono fatti interagire i moduli sviluppati da singoli team
- **Deploy:** è una delle fasi più delicate di tutto il processo e si riferisce all'installazione nell'ambiente di produzione.
- **Operate:** è il momento in cui vengono apportate le configurazioni necessarie dell'ambiente in produzione, per far in modo che il prodotto installato abbia tutte le dipendenze necessarie e quindi vada correttamente in esecuzione su una macchina.
- **Monitor:** fase del ciclo di vita che si impiega nel monitorare e verificare quale sia l'impatto sull'utente finale.

Facendo attenzione, si noterà che il modello agile è quasi del tutto simile a quello appena esposto. Le sostanziali differenze tra i due, infatti, sono le tempistiche dei rilasci, che nel modello agile sono sostanzialmente la durata dello sprint, mentre in questo appena esposto i rilasci posso essere eseguiti anche più volte al giorno. Questo da una parte ci mette di fronte ad alcuni lati positivi: il continuo rilascio di release rende i cambiamenti minimi, così da ridurre l'impatto sul cliente, la ricerca di eventuali errori e patch molto

più veloce visto che gli eventuali errori si possono individuare velocemente; ma dall'altra ci presenta alcune criticità dovute appunto alla collaborazione stretta dei due team.

2.3 Pratiche DevOps

Lo scopo delle pratiche DevOps è sicuramente rendere i processi di sviluppo più agevoli e gli sviluppatori più responsabili, creando codice che sia il più facilmente modificabile da eventuali errori, anche durante i processi di deployment e troubleshooting.

Tutte queste suddette problematiche ci fanno capire che con il modello DevOps i processi devono essere il più possibile automatizzati per premettere agli sviluppatori rilasci continui e al team Ops di non aggiungere un livello d'errore o di ritardo nelle richieste continue degli sviluppatori. Automatizzare tutte le procedure che sono soggette a errore umano è uno dei principi ponderanti di questa politica di rilasci. Standardizzando le procedure si avrà una ripetibilità che ci permette di eseguire in maniera strutturata azioni più o meno complesse senza l'intervento continuo dell'esperto o del team Ops.

Solitamente queste procedure si ottengono tramite degli script tratti come parti integranti dell'infrastruttura che circonda il progetto, quindi avente le stesse polite.

Capitolo 3

Distribuzione di configurazioni

Nel seguente capitolo si espone una panoramica dei principali modi utilizzati per la configurazione di macchine distribuite. Si esporrà anche un concetto molto importante in ambito DevOps e dell'infrastruttura come codice (IaC).

Dando un breve sguardo ai vari modi utilizzati per distribuire configurazioni e alle principali piattaforme software utilizzate per le configurazioni in ambienti distribuiti.

3.1 Principali modelli di configurazione in un'infrastruttura

Da quando vengono usate più macchine server per la creazione di servizi, uno dei più grandi problemi è la configurazione di esse.

In questi casi entrano in gioco gli strumenti di orchestrazione per la gestione delle configurazioni, solitamente configurazioni sulle macchine sia come host fisici o macchine virtuali, gestiti direttamente da console, spesso tramite connessione diretta in script ed ssh oppure a monte, con delle configurazioni direttamente nelle immagini dei os da virtualizzare, ma si sta affermando sempre di più un nuovo metodo per la gestione delle configurazioni. Vediamo in breve una descrizione dei casi appena citati.

- **Configurazione dell'immagine dell'OS:** sicuramente una delle tecniche più usate poiché permette di virtualizzare tutte le macchine di cui si ha bisogno ed avere un allineamento sia in termini di librerie del OS che di configurazioni come proxy e infrastruttura di rete. Uno dei punti a sfavore è legato all'aggiornamento delle librerie eseguibili su di esse, ad esempio una volta aggiornate le librerie sull'immagine si deve tornare su ogni macchina già in esecuzione e creare automatismi che permettano l'aggiornamento automatico.

- **Configurazione tramite Script ed SSH:** parliamo di una tecnica molto diffusa, poiché molto più flessibile, che si basa sul principio di depositare script in ogni macchina e dopodiché far eseguire in qualche modo da remoto, tramite ssh. Questo metodo ha il forte vantaggio di essere centralizzato su ogni macchina, quindi non si ha bisogno di un server orchestratore che gestisca tutti gli altri, a meno che non si voglia creare qualche forma di automatismo. Ha lo svantaggio di fornire una forte deframmentazione e quindi si rischia che i server non siano allineati.

- **Configurazione con software di configurazione:** per finire introduciamo questa pratica, cioè quella di usare software di configurazione per la gestione di configurazioni. Il concetto alla base è semplice, cioè quello di centralizzare le configurazioni da distribuire su tutte le macchine. Questa strategia ha due diverse famiglie di software, quelle basate su agenti e quelle basate su SSH e push based.

  Fatta una panoramica generale sull'utilizzo e su come vengono gestite configurazioni in più macchine, daremo uno sguardo su un modello chiamato IaC (Infrastructure as code).

3.2 Infrastruttura come codice

L'Infrastruttura come codice è il processo per gestire delle macchine senza configurare direttamente la macchina fisica o tramite strumenti iterativi. La progettazione dell'infrastruttura è la fase di vita del software dove vengono definite le esigenze di infrastruttura di un determinato software, come ad esempio il numero di VM necessarie, collegamenti ad altri servizi o macchine ecc [6].

Questo tipo di approccio è una parte integrante del modello DevOps, che utilizza le stesse politiche di versionamento del codice sorgente. Infatti, si basa sul principio che se lo stesso codice sorgente genera lo stesso file eseguibile, anche lo stesso modello IaC genera lo stesso ambiente. IaC è una delle pratiche chiave insieme alla consegna continua (o continuous delivery) nel modello DevOps.

Gli aspetti positivi dell'utilizzare tale tecnica sono molteplici: la possibilità di tracciamento avanti ed indietro, rimuovere del tutto il rischio associato all'errore umano, come un'errata configurazione di risorse che in altri casi è affidata alla configurazione manuale, la possibilità di creare ambienti del tutto uguali ad altri, in modo da creare ambienti isolati per fare test o riprodurre errori per un analisi, una chiara conseguenza è la diminuzione dei tempi di fermo dovuta appunto al tempo che si impiega nella configurazione manuale. Questo ci permette di creare ambienti stabili rapidamente e su larga scala in tempi brevi, prevenendo la deriva delle configurazioni in ambienti differenti.

Per utilizzare questo tipo di approccio è necessario qualsiasi tipo di framework o software che esegua configurazioni in modo dichiarativo, cioè che si concentra su quale dovrebbe essere la configurazione finale, in modo imperativo, che quindi definisce dei comandi che devono essere eseguiti nell'ordine appropriato per arrivare alla configurazione finale desiderata.

I metodi utilizzati da quasi tutti i framework sono due, il metodo push e quello pull. La differenza sta nel come vengono inviati i comandi ai server che devono essere configurati. Nel metodo push il server di controllo invia al sistema di destinazione le configurazioni da eseguire, nel metodo pull invece

è il server da configurare che interpella i server di controllo ed estrarrà le sue configurazioni.

3.2.1 Ambienti

Finora abbiamo parlato di ambienti senza darne alcuna definizione.

Per ambiente si intende tutto l'insieme di risorse computazionali sufficienti ad eseguire un determinato sistema software; questo include i dati, le comunicazioni esterne e interne, tutte le configurazioni e le entità esterne ed interne del sistema. Ciascun ambiente è teoricamente isolato dagli altri, quindi non può condividere risorse e tantomeno dati in scrittura, tranne che in alcuni casi in sola lettura sui dati. Infatti, se per esempio testiamo un'applicazione in un ambiente che condivide le risorse con quello di produzione, il test andrà a modificare i dati reali, compromettendo tutto il sistema.

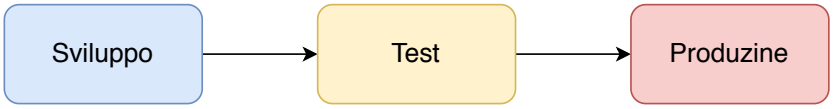

Figura 3.1: Rappresentazione della pipeline nei vari ambienti

L'Astrazione che hanno gli ambienti ci permette di utilizzarli in maniera diversa a seconda degli usi. Un esempio potrebbe presentarsi con un ambiente di test: in questo caso i logging applicativi saranno maggiori e più prolissi, cosa da minimizzare in ambienti più vicini alla produzione.

3.3 Introduzione sulle configurazioni

Abbiamo parlato abbondantemente di configurazioni, ma cosa sono e quali ambiti riguardano non è stato ancora approfondito. In questo paragrafo introdurremo cosa intendiamo per configurazioni e che ambiti riguardano. Prenderemo in oggetto tutti quei software, librerie e file che sono nel sistema

operativo e che sono da affiancare alle applicazioni che vengono eseguite sulla macchina. Parleremo dei più frequenti che possono sopraggiungere nella gestione [7].

3.3.1 Installazione di pacchetti

Installazione di pacchetti, solitamente di librerie o software necessari per la configurazione e l'esecuzione dei servizi che mette a disposizione una macchina, i pacchetti solitamente vengono gestiti da repository oppure compilati in modo nativo sulle proprie macchine dal codice sorgente, è un azione molto comune che solitamente viene eseguita per l'aggiornamento di bugfixes oppure quando si aggiungono funzionalità.

3.3.2 Autorizzazioni del file system

L'ambito di gestione delle autorizzazioni del file system, può riguardare vari aspetti, dallo spazio da dedicare ai log, che in generale occupano la parte più grande dei file system di un applicazione, a dove vengono collocati i dati, quali permessi sono attribuiti in modo da incapsulare e rendere più sicuro il sistema. Tutte queste problematiche possono essere racchiuse nelle autorizzazioni che l'amministratore di sistema propaga in tutte le macchine.

3.3.3 Distribuzione dei file di configurazione

La propagazione dei file di configurazione in più macchine è sicuramente una tra le azioni più comuni che possano essere fatte, che può riguardare sia livelli applicativi, come file inerenti ai singoli servizi, che file più specifici, come chiavi per alcune autenticazione; a livello applicativo o di sistema.

3.4 Peculiarità dei software di configurazione

In questo paragrafo introdurremo le principali caratteristiche dei software di configurazione, quali sono i più importanti, ed una panoramica generale

su di essi.

Inizierei introducendo gli strumenti di automazione più popolari, Ansible e Chef. Questi sono i principali software preposti ad automatizzare il provisioning del software, la gestione della configurazione e la distribuzione del software.

Si dividono in due macro famiglie, quella della distribuzione tramite agent e quella con connessione diretta sulle macchine.

3.4.1 Software di configurazione tramite agent

Si basano principalmente sul modello pull visto in precedenza. Quelli basati su agenti sono appunto gestiti da un agente ciò significa che quest'ultimo, installato sulla macchina, torna alla macchina di controllo per vedere quali modifiche devono avvenire; se ci sono queste verranno attuati anche sulle macchine che ospitano l'agente, altrimenti rimarranno silenti fin quando non verranno rilevate. Questo tipo di infrastruttura ha molteplici lati negativi, tra i quali la ridondanza dei dati, su ogni macchina verrà occupato spazio e bisognerà installare su ognuna l'agent, che ha bisogno di configurazioni network apposite.

Chef è principalmente basato su agent. Ciò non richiede SSH, ma richiede l'infrastruttura per eseguire il server di controllo.

3.4.2 Software di configurazione senza agent

Si basano principalmente sul modello push visto in precedenza. Questi tipi di software sono per lo più utilizzati su cluster di minor dimensione per la loro leggerezza di utilizzo, ma negli ultimi anni si sono sviluppati sempre di più i modelli di devops e sono tornati alla ribalta appunto per essere leggeri. I loro modelli si basano principalmente su connessioni push, quindi è il server di configurazione a comunicare con la macchina da configurare quali cambiamenti si devono portare.

Uno degli esponenti di questo tipo di software è sicuramente Ansible, lo vederemo più approfonditamente nel capitolo successivo.

3.4.3 Software di configurazione con Infrastruttura mutevole o immutabile

Gli strumenti di gestione della configurazione come Chef, Puppet, Ansible e Salt normalmente utilizzano un paradigma di infrastruttura mutabile. Ad esempio, configurando Ansible per installare una nuova versione di OpenSSL, verrà eseguito l'aggiornamento del software sui server esistenti e le modifiche verranno applicate sul posto. Nel tempo, man mano che si applicano sempre più aggiornamenti, ogni server crea una cronologia unica delle modifiche. Questo spesso porta a un fenomeno noto come deriva della configurazione, in cui ogni server diventa leggermente diverso da tutti gli altri, portando a sottili bug di configurazione difficili da diagnosticare e quasi impossibili da riprodurre.

Facendo lo stesso esempio con uno strumento di provisioning per distribuire una nuova versione di OpenSSL, dovresti creare una nuova immagine utilizzando Packer o Docker con la nuova versione di OpenSSL già installata, distribuire quell'immagine su una serie di server completamente nuovi, quindi annullare la distribuzione dei vecchi server. Questo approccio riduce la probabilità di bug di deriva della configurazione, rende più facile sapere esattamente quale software è in esecuzione su un server e consente di distribuire banalmente qualsiasi versione precedente del software in qualsiasi momento.

Capitolo 4

Ansible

Il seguente capitolo espone una panoramica su cos'è Ansible, del perché viene accostato al modello IaC. Esporremo come è formata la sua architettura in modo dettagliato, come interagisce con i server, parlando infine dei principali componenti che la popolano.

Abbiamo introdotto i concetti fondamentali che sono alla base di tutti gli automatismi che permettono la manutenzione e creazione di un infrastruttura eterogenea, dove altrimenti sarebbe difficoltoso anche un semplice aggiornamento. Parliamo del modello IaC, un approccio che garantisce una maggior portabilità delle applicazioni che man mano sono sempre più multi-piattaforma e multi-nodo.

La maggior parte dei software di gestione della configurazione è caratterizzata da un agent ed un server da cui iniziare l'orchestrazione. Ansible a differenza di questi, non lo richiede, quindi occupa spazio e risorse solo durante l'esecuzione dell'attività sul nodo controllato, ha la peculiarità di poter essere eseguito su qualsiasi macchina Linux, Unix-like e MacOS, può essere eseguita in Windows solo come nodo controllato.

Nelle sezioni successive si osserverà il funzionamento della piattaforma ed alcuni esempi di comandi per usarlo.

4.1 Elementi principali

In questa sezione analizzeremo i principali elementi di Ansible, quali componenti sono necessari per essere eseguito e come utilizzarlo. Prima daremo uno sguardo alle parole chiave che caratterizzano Ansible, come Inventario, Modulo, Playbook, Task.

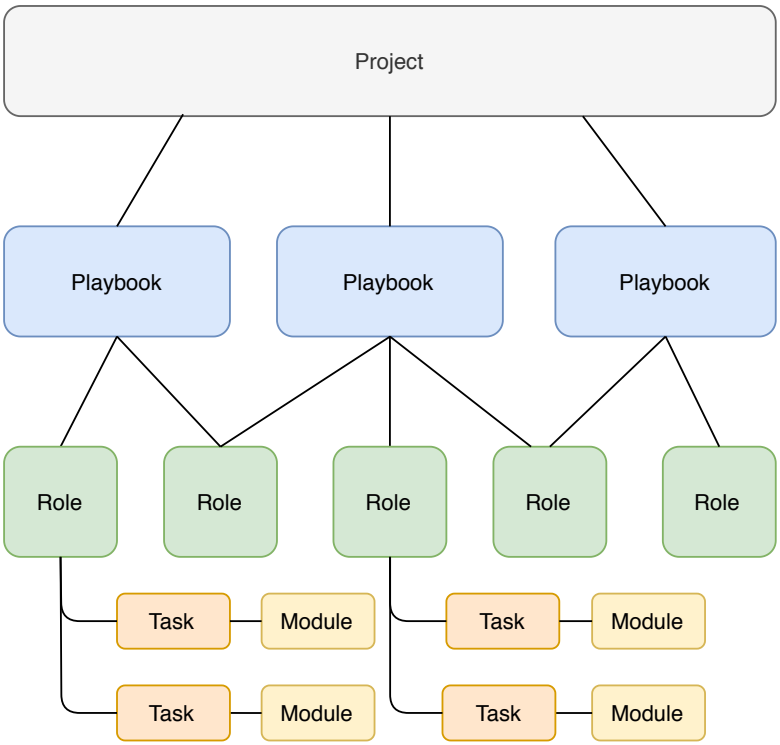

Figura 4.1: Struttura degli script [5]

Partiremo quindi in modo gerarchico nell'introduzione dei componenti che determinano Ansible.

4.1.1 Inventario

Quando si parla di Inventario si intende una lista di macchine dove Ansible può operare, raggruppate in gruppi più o meno grandi. L'inventario può essere scritto in vare estensioni, il file di default è nel percorso

/etc/ansible/hosts, ma niente ci vieta di creare altri Inventari e farli puntare in modo dinamico, passandoli da riga di comando.

```
#esempio di gruppo di macchine
10.25.1.56

[ansible]
10.25.1.57
db01.intranet.ansible.net
db-[99:101]-node.ansible.com
www[001:006].ansible.com
ansible1 ansible_host=192.168.0.0
```

Listing 4.1: Esempio di Inventario

Nel Listing 4.1 sono rappresentati alcuni esempi di come definire hosts in un inventario; la sintassi è molto semplice, i gruppi si creano utilizzando un nome tra parentesi quadre, quel nome etichetterà l'intero gruppo, le macchine possono essere elencate in vari modi tramite ip, nome macchina e persino indicandoli con un'etichetta a nostra scelta.

4.1.2 Moduli

I moduli consentono di controllare le risorse di sistema della macchina, come servizi, pacchetti, file, autorizzazioni, ecc. I moduli possono essere visti come delle singole istruzioni che vengono eseguite sulla macchina, è possibile richiamare un singolo modulo come un'attività singola o richiamare diversi moduli racchiusi in un playbook. Infatti per Ansible sono i comandi base da dove partire, possono essere sviluppati in Python, ma ne esistono molti messi a disposizione per essere utilizzati. I più interessanti per le nostra trattazione sono ovviamente quelli riguardanti l'installazione di vari pacchetti, sia tramite yum che apt.

```
root@server1:~# ansible all -m ping
[WARNING]: Platform linux on host 192.168.1.20 is using the
   discovered Python interpreter at /usr/bin/python, but
```

```
future installation of another Python interpreter could
    change this. See
https://docs.ansible.com/ansible/2.9/reference_appendices/
    interpreter_discovery.html for more information.
192.168.1.20 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false,
    "ping": "pong"
}
```

Listing 4.2: Esempio di Utilizzo di un Modulo

Ci sono una variegata schiera di moduli per la creazione di utenze, possibilità di distribuire file e altri moduli incentrati sui privilegi e permessi di accesso a cartelle o file. Tutti questi moduli sono già inclusi in Ansible, come si può vedere nella Figura 4.1 i moduli infatti sono la parte più in basso di un progetto Ansible e in generale di come è pensata la sua architettura. Vederemo questi aspetti in modo molto più approfondito successivamente.

4.1.3 Task

I task sono di fatto le operazioni eseguite in sequenza sulla macchina. Possiamo dire che i task sono una sequenza definita di moduli che vengono richiamati.

4.1.4 Ruoli

I ruoli sono componenti più complessi e raggruppano operazioni legate tra di loro con uno scopo specifico, esempio del riavvio di un servizio dopo aver modificato i file di configurazione. Sono composti da una directory con sottodirectory che contengono un file main.yml, che specifica la sequenza delle operazioni da eseguire. I ruoli infatti unificano una serie di operazioni coerenti tra loro per avere più riusabilità delle stesse. Ad esempio, l'installazione, la

configurazione e la gestione del servizi Apache HTTP Server possono essere raggruppati in un ruolo, avendo la possibilità di riutilizzare questo ruolo in diversi Playbook. In termini pratici, un ruolo è una struttura di directory contenente tutti i file, le variabili, i gestori, necessari all'esecuzione di un'attività. Quando viene creato un ruolo, la struttura di directory predefinita contiene quanto segue:

```
.travis.yml
README.md
defaults/
    main.yml
files/
handlers/
    main.yml
meta/
    main.yml
tasks/
    main.yml
templates/
tests/
    inventory
    test.yml
vars/
    main.yml
```

Listing 4.3: Esempio di alberatura file di Ruolo

4.1.5 Playbook

Il playbook è composto da una lista di task, delle relative variabili legati a determinati gruppi di macchine in inventario. Vengono utilizzate per avere la successione delle operazioni che Ansible dovrà eseguire sulle macchine in cui andrà ad operare. Sostanzialmente il Playbook è un file contenente i comandi da eseguire su quell'host, definisce inoltre il gruppo di host in cui devono essere eseguite le attività tramite i moduli per l'installazione automatica di applicazioni sulle macchine di destinazione.

```
#esempio copia di un file
- hosts : all
  tasks:
    - copy :
        src : /ansible/playbook/example.txt
        dest: /home/ansible
        owner: root
        group: root
        mode: 0644
```

Listing 4.4: Esempio di Playbook

4.2 Linguaggi utilizzati

Un modulo scritto per Ansible è semplicemente un file eseguibile che legge un JSON da stdin e stampa un JSON su stdout. Ciò significa che è possibile creare moduli in qualsiasi linguaggio, che sia Bash, Python, Node.js, Go o Ruby. Mentre per la sintassi dei playbook viene utilizzato un linguaggio chiamato YAML.

4.2.1 YAML

Vederemo in dettaglo il linguaggio YAML che viene utilizzato.

Il linguaggio preso in considerazione è nato con l'idea di essere utilizzato per la serializzazione di dati, quindi viene comunemente utilizzato per i file di configurazione nelle applicazioni in cui i dati vengono archiviati o trasmessi. Tra i suoi punti di forza ci sono sicuramente l'elevata leggibilità, YAML utilizza un indentazione simile a quella di Python e questo aiuta molto la leggibilità del codice. Un cheat sheet e le specifiche complete sono disponibili sul sito ufficiale [9]. Tra le peculiarità di questo linguaggio:

- **Spazi per la struttura:** il rientro degli spazi bianchi viene utilizzato per delineare la struttura; tuttavia, i caratteri di tabulazione non sono consentiti come parte del rientro.

- **I commenti:** i commenti iniziano con il segno # , possono iniziare ovunque su una riga e continuare fino alla fine della stessa.
- **Le stringhe:** le stringhe sono normalmente non quotate, ma possono essere racchiuse tra virgolette singole o virgolette doppie.
- **Elenchi:** i membri dello stesso elenco sono indicati da un trattino iniziale con un membro per riga, con lo stesso livello di rientro; un elenco può essere specificato anche racchiudendo il testo tra parentesi quadre con ciascuna voce separata da virgole.

```
1 ---  #esempio di elenco
2 -  Fedora
3 -  FreeBSD
4 ---  #esempio di elenco abbreviato
5 [ Debian ,  Fedora , FreeBSD ]
```

Listing 4.5: esempio di elenco

- **Dizionario:** un dizionario è rappresentato in una forma semplice, i due punti devono essere seguiti da uno spazio, sono possibili strutture di dati più complesse, come elenchi di dizionari, dizionari i cui valori sono elenchi o una combinazione di entrambi, ma come gli elenchi hanno anche loro una forma abbreviata.

```
1 ---  #esempio di dizionario
2 -  user:
3     name: Marta
4     job: Developer
5     skills:
6       - python
7       - YAML
8 ---  #esempio di dizionario abbreviato
9 user: {name: Marta, job: Developer, skill: [ python , YAML ]}
```

Listing 4.6: esempio di dizionario

Come si può notare tutti i componenti del linguaggio sono molto propensi ad una facile leggibilità del codice, questo aiuta e non poco lo sviluppo che risulta semplice ed intuitivo.

Capitolo 5

Ansible a confronto

Con questo capitolo si metterà a confronto Ansible con alcuni software simili, si vedrà prima dove tende la comunità Iac.

5.1 Andamento della comunità IaC

Cercando di comprendere anche altri software, daremo uno sguardo per grandezza di progetti aperti dalle comunità. Questo non è di certo un sintomo di qualità ma ci fa ben capire dove andranno nel futuro i software per distribuzione di configurazioni.

	Source	Cloud	Contributc	Start	Commit (1 month)	Issues(1 month)	Libraries	Stack Overflow	Jobs
Chef	Open	All	+18%	+31%	+139%	+48%	+26%	+43%	-22%
Puppet	Open	All	+19%	+27%	+19%	+42%	+38%	+36%	-19%
Ansible	Open	All	+195%	+97%	+49%	+66%	+157%	+223%	+125%
SaltStack	Open	All	+40%	+44%	+79%	+27%	+33%	+73%	+257%
Cloud Formation	Closed	AWS	?	?	?	?	+57%	+441%	+249%
Heat	Open	All	+28%	+23%	-85%	+1566%	0	+69%	+2957%
TerraForm	Open	All	+93%	+194%	-61%	-58%	+3555%	+1984%	+8288%

Figura 5.1: Come sono cambiate le comunità IaC tra settembre 2016 e maggio 2019 [13]

Analizzando questi dati che sono solamente una media di dati presi nel mese di settembre 2016 e nel mese di maggio 2019, si capisce dove si stanno affermando i vari software di configurazione.

Si vede da questi dati come l'interesse per strumenti senza un'installazione di software aggiuntivo, come potrebbe essere un master o un agent, è sempre molto apprezzata [14]. Infatti soluzioni a basso impatto sull'infrastruttura si sono affermate sempre di più.

Va fatta una precisazione: non tutti questi software elencati sono veri e propri software nati per la gestione di configurazione, infatti sono stati presi in considerazione anche software come CloudFormation e Terraform che sono strumenti di provisioning, il che significa che sono progettati per eseguire il provisioning dei server stessi, domandando ad altri strumenti la parte di configurazione. Questo però non va a discapito della bontà di questi dati, non essendo rigorosi e riconosciuti.

Nei prossimi paragrafi analizzeremo alcuni strumenti software per la gestione di configurazione,faremo un confronto con Ansible, quindi parleremo di Puppet e Salt.

5.2 Puppet

Puppet è uno strumento di gestione della configurazione, ispirato dal design di Chef, un altro strumento di gestione della configurazione molto popolare, potente e ampiamente utilizzato all'interno di infrastrutture di distribuzione molto complesse. Insieme ad Ansible sono strumenti di gestione della configurazione, il che significa che sono progettati per installare e gestire software su server esistenti.

Il progetto iniziale di Puppet era basato su un modello client-server, scritto in ruby, in cui nel client viene installato un agent su ogni nodo. Periodicamente il processo demone di Puppet si sincronizzerà con il server, per verificare eventuali modifiche alla configurazione che potrebbero essere necessarie per il push dal server, se il client trova qualcosa di diverso al di fuori dell'impostazione Puppet, verrà ripristinata allo stato precedente. Questo ci fa capire come Puppet sia sicuro e robusto [10].

5.2.1 Puppet - Ansible

Questi due strumenti per loro natura hanno parecchie caratteristiche in comune, ma analizzeremo cosa li distingue uno dall'altro in modo da differenziarli. Mettendoli a confronto nelle seguenti caratteristiche:

- **Il linguaggio:** Una delle differenze sostanziali tra software di configurazione diversi sono i linguaggi che sono alla base di essi. Come già esposto nel Paragrafo 4.2 Ansible usa un linguaggio chiamato YAML. Il linguaggio specifico di Puppet è radicato in Ruby, ma la sintassi è molto più vicina ai linguaggi imperativi. Questi due differenti approcci hanno lo scopo comune di essere facilmente appresi.

```
1 class foo {
2    user { 'user1':
3       ensure  => 'present',
4       comment => 'user1',
```

```
5       gid     => '1001',
6       groups  => ['sudo', 'staff'],
7       home    => '/home/user1',
8       shell   => '/bin/bash',
9       uid     => '1001',
10    }
11 }
```

Listing 5.1: Creazione di un utente con file in linguaggio Puppet

- **Facilità d'uso e curva di apprendimento:** La facilità d'uso dovrebbe far parte dei criteri di valutazione degli strumenti di gestione della configurazione. Dato che Ansible si concentra sulla semplicità d'uso esplicitamente come il suo più grande valore trainante, questa è un'area in cui funziona straordinariamente bene. Questo non vuol dire che Puppet sia difficile da usare, infatti racchiude molta di potenza e il suo approccio alla gestione della configurazione rende la maggior parte delle cose ovvie fintanto che si segua il modo di Puppet per automatizzare un'infrastruttura.

  Dal punto di vista della curva di apprendimento, entrambe le piattaforme sono facili da usare, ma Ansible ha un leggero vantaggio. Lo stile YAML dichiarativo è facile da imparare e il codice Ansible non diventa mai troppo complesso. Puppet è arrivato a riconoscere alcune delle sfide associate alla combinazione di dati e codici nei file sorgente. Ciò ha spinto l'ascesa di Puppet Hiera, una soluzione di archiviazione dati che utilizza il formato YAML per archiviare coppie chiave-valore dei dati di configurazione.

- **Gestione di errori e guasti:** Un argomento da non sottovalutare è sicuramente come questi strumenti gestiscono gli errori o guasti che possono esser causati sui nodi controllati. L'Agent Puppet ha delle politiche di sicurezza dove tutte le modifiche verranno annullate se differiscono da quanto descritto nei manifesti di Puppet. Questo tipo di

controllo in Ansible non esiste per cui è uno dei punti deboli di Ansible e può essere altamente distruttivo per le macchine gestite.

5.3 Salt

Salt è nato come un sistema di esecuzione remota distribuito, utilizzato per eseguire comandi e interrogare dati su nodi remoti, in seguito è stato esteso a un sistema di gestione della configurazione, in grado di mantenere i nodi remoti in stati definiti. Come già visto, anche Salt, come Puppet, utilizza un infrastruttura su un modello client-server, implementata anche in versione senza agent (salt-ssh).

5.3.1 Salt - Ansible

Mettendo a confronto i due software e l'infrastruttura, notiamo che sono molto simili, poiché si basano entrambi su python ed entrambi hanno la gestione dei loro moduli con il linguaggio YAML, ma differiscono per altre scelte degli sviluppatori , infatti Salt contrariamente ad Ansible è basato su agent. Daremo un breve sguardo alle caratteristiche che posso essere importanti:

- **Il linguaggio:** I punti che li accomunano sono la base in python che rende agevole lo sviluppo di nuovi moduli, la completezza della documentazione [11] che, essendo componenti opensource, agevolano molto il lavoro. Hanno in comune anche il linguaggio YAML per la rappresentazione dei file di stato.

```
1 user1:
2   user.present:
3     - fullname: user1
4     - shell: /bin/bash
5     - home: /home/user1
```

```
6     - uid: 1001
7     - gid: 1001
8     - groups:
9       - sudo
10      - staff
```

Listing 5.2: Creazione di un utente con il modulo salt.states.user

- **Facilità d'uso e curva di apprendimento:** La semplicità e la documentazione di facile consultazione di Ansible gli danno un vantaggio su Salt in questa categoria; infatti, è ampiamente considerata come la piattaforma di automazione più facile da usare. Salt d'altro canto fornisce anche un'ampia documentazione ben congegnata e completa.

- **Gestione di errori e guasti:** Quando si parla di sicurezza e gestione, Salt essendo basato su agent è molto più sicuro. Ogni esecuzione eseguita da Salt viene memorizzata per un numero predefinito di giorni nel master. Ciò semplifica il debug, ma consente anche di verificare se sia accaduto qualcosa di strano.

Conclusioni

In questo capitolo tirerò le somme su tutto ciò che abbiamo esposto.

Come abbiamo visto nei primi capitoli, si è iniziato dapprima con l'esporre come nascono i sistemi distribuiti, per poi passare ai modelli DevOps e approfondire le pratiche che caratterizzano questo modello di interazione. In seguito abbiamo esposto quali problematiche sono peculiari del team Ops, per poi passare a quali task sono più diffusi e in modo superficiale quali strumenti possono essere usati. Infine abbiamo analizzato uno strumento molto utile per i task esposti nei capitoli precedenti.

Ansible è uno strumento tanto potente, ma potenzialmente anche dannoso per la sua natura di cercare di lanciare qualsiasi comando come root di default. Per questo motivo Ansible si rivolge quasi esclusivamente agli amministratori di sistema e si tiene alla larga dal mondo degli utenti, dove può esser usato ma con parecchie limitazioni. Questa peculiarità non è passata inosservata durante la stesura di questo documento, ma si è voluto esporre appunto uno strumento per il team Ops che può implementare dei meccanismi per evitare questi lati negativi.

Siamo riusciti a vedere anche altri strumenti molto utili, paragonarli ad Ansible per complessità d'utilizzo e per il tipo di infrastruttura adottata; Ansible, infatti, è uno strumento che può essere adottato per compiti e per operazioni dirette sulla macchina, ma non gestisce tutti i casi possibili che vengono a crearsi, per questo va sicuramente affiancato ad altri strumenti che mantengano la robustezza dell'infrastruttura e che ne garantiscano l'integrità.

Infine, non manca il desiderio di riuscire ad implementare qualcosa di più

complesso con questo tipo di strumento.

Appendice A

Esempi di playbook Ansible

In questo appendice daremo un breve sguardo ad un utilizzo concreto di Ansible per alcuni playbook facili da implementare ma estremamente utili. Per prima cosa daremo un breve sguardo su come installare Ansible.

A.1 Dipendenze

- Python 2 (versione 2.7) o Python 3 (versioni 3.5 e successive).
- OpenSSH

A.2 Ansible Installazione

Installazione di Ansible sulle distribuzioni principalmente utilizzate [15].

A.2.1 Ansible Installazione : Debian e derivate

```
1 $ sudo echo "deb http://ppa.launchpad.net/ansible/ansible/
    ubuntu trusty main" | sudo tee -a /etc/apt/sources.list.d/
    ansible.list
2 $ sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-
    keys 93C4A3FD7BB9C367
3 $ sudo apt update
```

```
$ sudo apt install ansible
```

A.2.2 Ansible Installazione : Fedora e CentOS

```
$ sudo dnf install ansible
```

```
$ sudo yum install ansible
```

A.3 Esempi di playbooks

Creazione di semplici playbooks per alcuni tasks.

A.3.1 Distrubuzione di file

Questo playbook ci può essere d'aiuto nella distribuzione di file, su tutte le macchine presenti nel nostro file hosts.

```
- hosts : all
  tasks:
    - include_vars: file_list.yml
    - name: Copy file
      copy:
        src: "{{ item.src }}"
        dest: "{{ item.dest }}"
        owner: root
        group: root
        mode: "0644"
      with_items: "{{ files }}"
```

```
---
files :
  - { src: "./example.txt" , dest : "/example/" }
```

Listing A.1: Contenuto del file file_list.yml

A.3.2 Creazione di utenti

Con questo playbook vengono creati utenti e viene impostata la password solo per quelli appena creati, infatti è possibile aggiungere gli utenti già presenti ad altri gruppi [16].

```
- hosts: all
  tasks:
# task for create user
    - include_vars: users_list.yml
    - name: Create Users
      user:
        name: "{{ item.name }}"
        groups: "{{ item.groups }}"
        password: "{{passwd}}"
        comment: "{{ item.comment }}"
        update_password: on_create
      with_items: "{{ users }}"
```

```
---
passwd: "{{ 'password' | password_hash('sha512') }}"
users :
  - {name: user1 , groups: staff , comment: 'user1 example'}
  - {name: user2 , groups: staff , comment: 'user2 example'}
```

Listing A.2: Contenuto del file users_list.yml

A.3.3 Cancellazione di utenti

Con questo playbook vengono eliminati utenti, ma non i file e la directory home degli utenti eliminati.

```
- hosts: all
    - include_vars: users_list.yml
    - name: Delete Users and home
      user:
        name: "{{ item.name }}"
        state: absent
      with_items: "{{ users }}"
```

```
---
users :
  - {name: user1 }
  - {name: user2}
```

Listing A.3: Contenuto del file users_list.yml

Bibliografia

[1] Ansible Documentation, https://docs.ansible.com/

[2] Wikipedia, *Virtual machine*, 2020, https://en.wikipedia.org/wiki/Virtual_machine, 16 giugno 2020.

[3] Mendel Rosenblum, *The Reincarnation of Virtual Machines*, 2004, ACM Queue. Vol.2 no.5, https://queue.acm.org/detail.cfm?id=1017000, 31 agosto 2004.

[4] Len Bass, Ingo Weber, and Liming Zhu. *DevOps, A Software Architect's Perspective*, Pearson, 2015.

[5] P. Masek, M. Stusek, J. Krejci, K. Zeman, J. Pokorny and M. Kudlacek, *Unleashing Full Potential of Ansible Framework: University Labs Administration* , 2018 22nd Conference of Open Innovations Association (FRUCT), Jyvaskyla, 2018, pp. 144-150, doi: 10.23919/FRUCT.2018.8468270.0.

[6] M. Artac, T. Borovssak, E. Di Nitto, M. Guerriero and D. A. Tamburri, *DevOps: Introducing Infrastructure-as-Code*, 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), Buenos Aires, 2017, pp. 497-498, doi: 10.1109/ICSE-C.2017.162.

[7] Adam Hawkins, *Introduction to Ansible*, 2017, https://semaphoreci.com/community/tutorials/introduction-to-ansible, 26 Aprile 2017.

[8] Sameer S Paradkar, *A beginner's guide to everything DevOps*, https://opensource.com/article/20/2/devops-beginners, 27 Febbraio 2020.

[9] Oren Ben-Kiki, Clark Evans, Ingy döt Net, *YAML Ain't Markup Language (YAML^{TM}) Version 1.2 3rd Edition, Patched at 2009-10-01*, https://YAML.org/spec/1.2/spec.html, 1 Ottobre 2009.

[10] Puppet Documentation, https://puppet.com/docs/

[11] Salt Documentation, https://docs.saltstack.com/en/master/topics/ tutorials/walkthrough.html

[12] Anthony Shaw, *Ansible vs Salt (SaltStack) vs StackStorm*, https://medium.com/@anthonypjshaw/ansible-v-s-salt-saltstack-v-s-stackstorm-3d8f57149368

[13] Yevgeniy Brikman, *Why we use Terraform and not Chef, Puppet, Ansible, SaltStack, or CloudFormation*, https://blog.gruntwork.io/terraform-up-running-2nd-edition-early-release-is-now-available-b104fc29783f , The Gruntwork Blog 8 Luglio 2019.

[14] Flexera, *STATE OF THE CLOUD REPORT*, https://info.flexera.com/SLO-CM-REPORT-State-of-the-Cloud-2020

[15] Ansible Documentation, https://docs.ansible.com/ansible/latest/ installation_guide/intro_installation.html#managed-node-requirements

[16] Ansible Documentation, https://docs.ansible.com/ansible/latest/ modules /user_module.html

Ringraziamenti

Desidero innanzitutto ringraziare il relatore di questa tesi, Cosimo Laneve, per la sua disponibilità e la passione con cui svolge il proprio lavoro.

Un grazie di cuore alla mia famiglia, per il supporto che mi ha dato sotto ogni punto di vista e senza la quale non avrei mai potuto realizzare questo traguardo. Ringrazio in particolar modo mia sorella, Antonella Caravaggio, che si è sempre dimostrata una persona preziosa e leale.

Ringrazio anche tutti i miei amici, colleghi universitari e tutti coloro che ho incontrato in questo lungo cammino. Ringrazio particolarmente coloro che mi hanno aiutato alla stesura di questo documento, dandomi dei consigli e correggendomi dove la forma non sembrava adeguata. Ringrazio anche il mio amico Danilo Chiavelli, senza di lui non avrei mai scelto questa facoltà e di certo non avrei mai scelto questo ateneo.

Infine ringrazio la mia moka, ha deciso di rompersi proprio nei giorni dell'ultimo esame, senza di lei forse non sarei mai diventato un amante accanito di caffè.