In questo capitolo verranno presentati alcuni tra i principali tools necessari per implementare le metodologie DevOps.   
Sarà inoltre fornita una descrizione di altri programmi utilizzati per svolgere il lavoro che verrà presentato nel Capitolo 4 e perciò non necessariamente inerenti al contesto DevOps.

NB: in questo Capitolo non verrà menzionato Docker, strumento che a mio avviso merita una particolare attenzione e che sarà presentato a parte nel capitolo 5.

GIT

(figura Github)

Nel capitolo precedente ho più volte indicato quanto sia utile affidarsi a un version control system sin dalle prime fasi del progetto. E siccome siamo arrivati alla conclusione che non possiamo (e non ci conviene) più mettere sotto version control esclusivamente il codice, bensì praticamente tutto ciò che riguarda il nostro progetto e il suo contorno, tale pratica non si rivela soltanto utile, ma fondamentale.

Git è un sistema di controllo della versione distribuito. Scritto da Linus Torvalds nel 2005 per la comunità del kernel di Linux, Git è nato inizialmente come uno strumento da riga di comando, ma al giorno d’oggi sono principalmente utilizzati clienti o GUI (Graphic User Interface) che si integrino con Git. Il client più utilizzato è GitHub, servizio open-source che rappresenta un ambiente aperto in cui i programmatori possono condividere progetti e collaborare liberamente.

La differenza tra Git e molti dei precedenti sistemi di controllo di versione è che Git è un sistema distribuito, e non centralizzato. Ogni sviluppatore lavora sul proprio repository locale e l’intero team si appoggia al repository remoto del progetto, che può trovarsi su un server o su un altro computer di uno degli sviluppatori.   
Ogni membro del team interagisce con il repository remoto per mandare del nuovo codice (push) o per scaricare del nuovo materiale (pull).

A prescindere dal modo in cui si decide di utilizzare Git, tramite linea di comando o tramite client, ci sono tre comandi che è utile conoscere per avere le nozioni base di come si utilizza un version control system:

* COMMIT: con questo comando il lavoro viene “versionato” sul repository locale, in tal modo avremo sempre a disposizione questa versione del codice (anche dopo che si effettueranno successivi commit) e sarà quindi immediata recuperarla nel caso in cui ce ne fosse bisogno (ad esempio se lo sviluppatore si rendesse conto che una delle versioni precedenti del programma era più funzionante). È consigliato associare ad ogni commit un messaggio che spiega quali cambiamenti sono stati effettuati.
* PUSH: con questo comando viene mandata al repository remoto l’ultima versione del codice di cui si è fatta una commit sul repository locale.
* PULL: con questo comando si scarica il codice dal repository remoto per iniziare a usarlo in locale. Di conseguenza è logico aspettarsi delle modifiche nel codice rispetto a ciò che avevamo mandato al remoto con la Push perché altri membri del team potrebbero aver scaricato il materiale aggiungendo o modificando qualcosa.

(diagramma Git)

VIRTUALBOX

(figura Virtual Box)

VirtualBox è un software di virtualizzazione e come tale permette di astrarre le componenti hardware del PC o di un qualunque dispositivo in modo da renderle disponibili sotto forma di risorse virtuali.

Su una macchina ospitante (detta host), indipendentemente dal sistema operativo installato, è possibile caricare ed eseguire più sistemi operativi, anche molto diversi l'uno dall'altro.

Se sulla macchina "host" fosse installata, ad esempio, una certa versione di Windows, all'interno della macchina virtuale, grazie alla virtualizzazione si può installare una distribuzione Linux a cui verranno destinate parte delle risorse fisiche della macchina host.

Oltre al vantaggio di avere funzionalità appartenenti a sistemi operativi differenti su un’unica macchina, nel contesto dello sviluppo software la virtualizzazione assume un ruolo ancora più importante in quanto permette di avere potenzialmente più nodi della catena di deployment sulla stessa macchina.

VAGRANT

(figura Vagrant)

Vagrant è lo strumento di cui ci avvaliamo per integrare il concetto di virtualizzazione all’interno delle metodologie DevOps.

Vagrant è un software scritto in Ruby che assume la funzione di gestore di macchine virtuali permettendo di gestirne in maniera automatizzata la creazione, la configurazione e l’intera gestione.

Uno degli scenari più noiosi, infatti, è proprio la preparazione dell’ambiente di una macchina virtuale: dobbiamo avviare la macchina, farne il setup, rigenerare un pacchetto di diversi gigabyte e condividerlo con i membri del progetto.  
Se un giorno ci rendessimo conto di dover fare un upgrade di una componente: stesso giro e stesso scambio di gigabyte e la cosa peggiore è che questo processo non è versionabile e quindi molto difficile da reperire in un secondo momento. Tutto ciò, oltre a tenere impegnati per molto tempo i sistemisti in operazioni lente e ripetitive, è contro ai principi del DevOps.

Vagrant invece, consente di avere delle macchine virtuali completamente “scriptabili” sia in fase di configurazione (RAM, networking, spazio disco…) che di provisioning (installazione di software necessari). Ognuno di questi script potrà poi essere aggiunto al nostro version control system preferito e arrivare a versionare i vari scenari realizzando la cosiddetta Infrastructure as a Code, di cui si è già parlato nel Capitolo 2.

All’interno di un file, anch’esso scritto in Ruby, denominato Vagrantfile, sono contenute, le impostazioni e le configurazioni di tutte le macchine virtuali che si desidera avere.

Le componenti principali di Vagrant sono tre:

* Providers: per funzionare Vagrant deve appoggiarsi a un provider per gestire gli ambienti virtuali. VirtualBox, già presentato, è uno dei più utilizzati dagli sviluppatori.
* Box: Installare un sistema operativo all’interno di una macchina virtuale segue la stessa procedura lenta e macchinosa utilizzata per installarlo in una macchina reale. In tal modo il sistemista non trarrebbe alcun vantaggio, perciò Vagrant mette a disposizione dei “box” ovvero delle macchine pre-configurate con sistema operativo installato da cui partire. Inoltre c’è anche la possibilità di creare un box personalizzato secondo le necessità del nostro progetto (ad esempio box con LAMP stack: Linux, Apache, MySQL, PHP), metterlo sotto controllo di versione e riutilizzarlo quando e per tutte le macchine che vogliamo.  
  (comandi Vagrant)
* Provisioning: Il meccanismo di personalizzazione ed installazione di un software all’interno di una macchina vagrant è detto provisioning. All’interno del Vagrantfile, con il comando *config.vm.provision*, è possibile specificare le indicazioni su quali software devono essere installati sulla macchina.

(figura Vagrantfile)

Vagrant si utilizza da terminale, attraverso alcuni comandi che permettono di procedere alla creazione di macchine virtuali e alla loro gestione.   
Tali comandi, entrando nella sfera pratica dell’utilizzo del software, saranno presentati nel capitolo 4.

PUTTY

(figura Putty)

PuTTY è un SSH che permette di stabilire una sessione remota cifrata ad un altro computer emulando un terminale, effettuando il login con l’IP del computer a cui ci si vuole connettere.

Considerando un’architettura formata da macchine virtuali è utile avere uno strumento che permetta di collegarsi in maniera rapida e semplice, tramite la macchina host, a tutte le macchine virtuali guest per dialogare con loro e poterle controllare.

Per iniziare ad usarlo è importante impostare tutti i parametri base necessari per la connessione al server remoto (ad esempio l’IP, l’username e la password). Una volta fatto ciò sarà immediato collegarsi alla macchina e controllarla con il terminale PuTTY.

ANSIBLE

(figura Ansible)

Nel caso di architetture complesse, i temi dell’ Infrastructure as a Code e del Configuration Management diventano basilari e, se non implementati, si rischia di sprecare molte risorse, specie in campo Operations, per la gestione di tali aspetti.

Può quindi venir utile, oltre a quanto si può configurare tramite il Vagrantfile, affidarsi anche ad altri strumenti, specifici dell’Infrastructure as a Code e del Configuration Management.

Di seguito verrà presentato Ansible e nel farlo si menzioneranno anche Chef e Puppet, ma siccome nel Capitolo 4, per ragioni di complessità, tali strumenti non verranno utilizzati, se ne fornirà una spiegazione principalmente teorica in quanto gli aspetti pratici non vengono affrontati nel concreto in questo elaborato.

Ansible è un software libero che consente di automatizzare le procedure di configurazione e gestione sui sistemi unix-like utilizzando il protocollo SSH.  
Come per la maggior parte dei software di configurazione, Ansible sfrutta due tipi di server: nodi e macchine controllori. La macchina controllore è quella che orchestra il tutto eseguendo determinati comandi sui nodi connettendovisi attraverso una chiave di accesso SSH.

I software più popolari di questo tipo sono Chef e Puppet, ma entrambi, a differenza di Ansible, sono agent-based. Questo particolare è ciò che negli ultimi anni (Ansible nasce nel 2012) ha portato i leader nel campo DevOps a concentrarsi su un sistema agentless, quale è Ansible.  
In un'architettura agent-based, i nodi devono avere installato un demone che comunichi con la macchina controllore mentre con l'architettura agentless non serve che sui nodi sia installato alcun software specifico oltre al server SSH. Questo riduce di molto l'ipotetico traffico di rete necessario per utilizzare questo programma.

Prima di tutto non sempre è possibile installare un agente sul server di destinazione. Un agente è un piccolo software che rimane in esecuzione permanente su uno o più server ed ha lo scopo di attendere comandi e configurazioni impartite dal server master. Pensiamo ad esempio ad ambienti server che per ragioni di conformità o sicurezza non possono ospitare agenti o software estranei.

Un altro vantaggio di Ansible rispetto ai concorrenti è che non richiede la conoscenza di linguaggi di scripting complessi, ma necessita solo della presenza di python sulle macchine amministrate e la conoscenza della sintassi YAML.  
Una soluzione come Chef invece richiede la conoscenza approfondita di Ruby e non tutti gli amministratori di sistema, o più in generale i membri del team Operation, possono permettersi di studiare un linguaggio di programmazione che useranno solo limitatamente per gestire la configurazione di un sistema.

(figura architettura Ansible)

Le tre componenti principali di Ansible sono:

* Inventory: l’inventario è una lista di server sui quali Ansible applica le configurazioni di sistema e le istruzioni di automazione. Ad esempio, in una CD pipeline può essere interessante avere più file hosts, uno per stage, ciascuno a definire l'elenco delle macchine coinvolte dalle operazioni di automazione.
* Playbook: I Playbook sono dei "libri" dai quali Ansible "legge" le istruzioni da eseguire sui nodi destinatari. I Playbook, inseriti in Version Control System, permettono di gestire «as code» una infrastruttura, modellandola intorno ad uno scenario codificato.
* Modules: Ansible è composto da numerosi module. Si possono immaginare i module come "package" di azioni che estendono le capacità di Ansible, permettono di gestire risorse come file, package, servizi, e azioni su utenti e filesystem di ogni genere, Ad esempio esistono moduli per installare pacchetti software su server Linux, moduli per gestire file di configurazione e database, etc.

ANT

(figura Ant)

Sempre rimanendo nell’ambito dell’automazione, ma uscendo per un attimo dal contesto DevOps, è utile presentare Ant, strumento che, a prescindere dalle metodologie utilizzate, velocizza le operazioni di compilazione ed è quindi consigliabile nell’ambito della programmazione in ambiente Java.

Ant è una libreria JAVA sviluppata dalla Apache che permette di automatizzare il processo di sviluppo di applicazioni Java. Con Ant, infatti, è possibile creare un progetto che compila, genera la documentazione, realizza file jar, war ed ear ed effettua il deploy di un’applicazione web su un application server tipo Tomcat, tutto con il semplice lancio di un comando.

I comandi di Ant sono letti da un file XML, di solito chiamato build.xml. In questo file bisogna definire le operazioni, chiamate target, e per ciascuna operazione i comandi da eseguire, chiamati task.

Ant ha quindi il compito di leggere e compiere le istruzioni presenti in un file xml che verranno quindi eseguite in automatico lanciando semplicemente il comando di Ant. Nel Capitolo 4, all’interno dello sviluppo del framework, sarà messo in pratica tutto ciò.

Di seguito, con il fine di fornire una presentazione più completa dell’argomento, sono elencati alcuni tra i principali comandi che Ant mette a disposizione:

Analizziamo i comandi principali (task) che Ant mette a disposizione:

* <echo>: scrive un messaggio sulla console
* <property>: definisce una o più property. È possibile anche definire le property in un file esterno
* <jar>: crea un file jar, ovvero un archivio dati compresso contenente classi Java. È necessario definire il nome del file jar da creare e la directory contenente i file .class.
* <war>: permette di creare un file war, ovvero un archivio dati usato in Java per raggruppare file di diverso tipo (servlet, xml ecc) che compongono un’applicazione web in Java.
* <javac>: permette di compilare i file java. Ant scandisce ricorsivamente la directory specificata nel parametro srcdir alla ricerca di file java per i quali il relativo file .class non è presente o è meno recente del sorgente. Con questo controllo viene evitato di compilare i file già compilati che non hanno subito modifiche rispetto all’ultima compilazione.