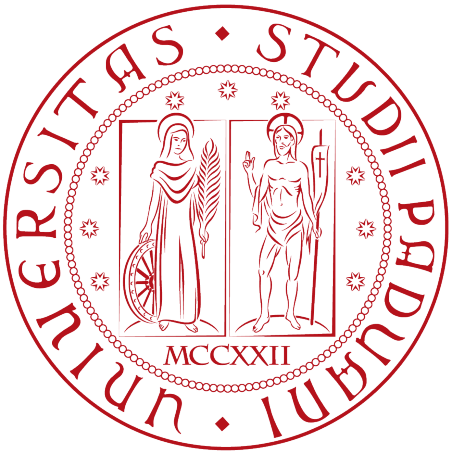
Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica



Containerizzazione ed orchestrazione di soluzioni software applicative .NET tramite utilizzo di Docker & Docker Compose

Tesi di laurea triennale

Relatore

Prof. Massimo Marchiori

Laureando

Edoardo Caregnato

Anno Accademico 2021-2022

Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

— Oscar Wilde

Dedicato a ...

Sommario

Il presente documento descrive il lavoro svolto durante il periodo di stage, della durata di circa trecento ore, dal laureando **Edoardo Caregnato** presso l'azienda **PAT - Infinte Solutions**. Gli obiettivi da raggiungere, concordati tra Azienda ed Università, al fine di completare con successo l'esperienza di stage erano questi di seguito esplicati.
In primo luogo è stato richiesto lo studio individuale relativo alle differenze architetturali tra Container e Virtual Machine, con relativa discussione ed esposizione di quanto elaborato al Tutor aziendale **Ruggero Maffei**.
In secondo luogo è stato richiesto uno studio individuale di Docker e Docker Compose e delle relative API di automation, il quale scopo finale era quello di predisporre un ambiente totalmente compatibile al fine di eseguire con successo i due applicativi di punta dell'Azienda, ovvero HDA e CX studio. Dopo un'attenta analisi sulla fattibilità e tempistiche del progetto, è stato concordato, assieme all'Azienda, di concentrare l'esperienza curricolare sulla containerizzazione dell'applicativo HDA con tutte le sue relative estensioni.
Il terzo obiettivo dello stage curricolare è stata la predisposizione dei relativi container atti all'esecuzione dell'applicativo HDA comprensivo di tutti gli strumenti di monitoraggio richiesti dall'Azienda.
Quarto ed ultimo obbiettivo è stato lo studio e la creazione di container atti all'aggiornamento della versione di HDA, con relativo studio sulla possibilità di automazione dell'intero processo.

"Life is really simple, but we insist on making it complicated"

— Confucius

Ringraziamenti

Innanzitutto, vorrei esprimere la mia gratitudine al Prof. Massimo Marchiori, relatore della mia tesi, per l'aiuto e il sostegno fornitomi durante la stesura del lavoro.

Desidero ringraziare con affetto i miei genitori per il sostegno, il grande aiuto e per essermi stati vicini in ogni momento durante gli anni di studio.

Ho desiderio di ringraziare poi i miei amici per tutti i bellissimi anni passati insieme e le mille avventure vissute.

Padova, Dicembre 2022 Edoardo Caregnato

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Il seguente capitolo intende introdurre il lettore al **contesto applicativo** dello stage effettuato presso **PAT Group** evidenziando, in ordine, le **metodologie di lavoro** dell'Azienda, l'idea generale alla base dello stage formativo e l'**organizzazione** del presente documento.

1.1 L'azienda

Pat Group è un'azienda innovativa che da oltre 25 anni opera nello sviluppo di applicativi software per aziende ed istituzioni pubbliche e private. Il principale *core business* dell'Azienda comprende lo sviluppo di applicativi di *IT Service Management*, *CRM*, *lead management*, *automazione dei processi di business* e *social collaboration*. Le principali energie che l'Azienda mette a disposizione sono destinate all'**evoluzione dei processi di business** dei clienti, fornendo supporto per l'innovazione digitale e l'automazione dei processi interni, per far raggiungere ad ogni cliente la *leadership* nel relativo settore in cui opera.
I principali valori che PAT desidera ottenere sono riassumibili nei seguenti punti:

* **Successo:** l'Azienda lavora costantemente al fine del raggiungimento, da parte dei suoi clienti, degli obiettivi di successo;
* **Innovazione:** PAT accompagna i propri clienti attraverso la difficile fase del cambiamento digitale, spingendo a innovare e costantemente aggiornare le tecnologie interne;
* **Qualità:** l'impegno di PAT, verso qualsiasi dei suoi clienti, è quello del raggiungimento di questo valore attraverso procedure di innovazione riconosciute e certificate a loro volta da standard qualitativi;
* **Talento:** PAT organizza per i suoi clienti dei percorsi formativi interni mirati alla ricerca di talento di ogni collaboratore;
* **Fiducia:** la realtà aziendale è fortemente concentrata sulla fiducia nella professionalità dei collaboratori dell'Azienda, per guidare al meglio nella crescita professionale ogni cliente;

- **Trasparenza:** al fine di raggiungere ogni obiettivo e valore l'Azienda adotta un metodo di comunicazione costante e continuo, atto a monitorare tutti i progressi effettuati ed interventire tempestivamente in casi di criticità.

PAT opera all'interno di un luogo innovativo chiamato **InfiniteArea**, ovvero uno spazio dedicato alla crescita di idee innovative nell'ottica di una futura implementazione nel *core business* dell'Azienda.
Da giugno 2013, PAT entra a far parte dell'universo **Zucchetti**, primo gruppo italiano nel panorama ICT, con **Patrizio Bof** in qualità di fondatore ed unico amministratore della Società **PAT Group**.
L'Azienda collabora e fornisce soluzioni software per una moltitudine di clienti, alcuni tra i più importanti risultano **Pirelli**, **Aruba**, **BPER**, **AirDolomiti**, **ARVAL** e **WeBank**.

1.2 Organizzazione dei capitoli

Il seguente elenco intende fornire al lettore, in maniera ordinata, una panoramica informativa riassuntiva circa il contenuto e la suddivisione in capitoli di questo documento:

capitolo 2: in questo capitolo viene esposta una **panoramica generale** relativa al progetto assegnato;

capitolo 3: questo capitolo tratta le principali differenze tra **architettura a macchine virtuali ed a container**, fornendo al lettore un approfondimento dettagliato su ognuna delle tecnologie trattate;

capitolo 4: questo capitolo espone l'**architettura generale** del progetto implementata durante tutto il percorso di stage;

capitolo 5: questo capitolo approfondisce la metodologia di **orchestrazione tra container** tramite utilizzo di Docker Compose;

capitolo 6: in questo capitolo viene trattato il tema di **Reverse Proxy**, fornendo al lettore tutti i passi eseguiti al fine dell'implementazione sul progetto assegnato;

capitolo 7: questo capitolo conclusivo espone una **panoramica sugli obiettivi raggiunti**, il grado di conoscenze acquisite e grado di soddisfacimento relativo allo stage curricolare da parte dello studente.

Capitolo 2

Descrizione dello stage

Breve introduzione al capitolo
In questo capitolo si esporrà una panoramica relativa al progetto assegnato dall'Azienda, con un approfondimento relativo ai requisiti obbligatori da soddisfare ed obiettivi raggiunti al termine dell'esperienza di stage.

2.1 Scopo dello stage curricolare

Lo scopo principale dello stage è la *containerizzazione* delle due soluzioni applicative maggiormente utilizzate dall'Azienda, ovvero HDA e CX studio in ambito Windows, con la possibilità di monitoraggio in *real-time* degli stessi tramite *containerizzazione* di ulteriori applicativi quali "**Telegraf**", "**Loki**" e "**Promtail**" precedentemente configurati da un altro stagista.
I tools appena citati non saranno oggetto di ulteriore approfondimento in questo documento, in quanto non raffiguravano nei requisiti concordati dello stage ed introdotti successivamente dall'Azienda al fine di collegarsi con il lavoro dell'ulteriore stagista **Francesco Pantaleoni**.
Tramite *containerizzazione* dei due applicativi sopracitati è stato richiesto, in aggiunta, la creazione di uno script di **reverse-proxy automatizzato** che, tramite Docker API, identificava le *sandbox* applicative, ovvero le **istanze applicative** di HDA, Loki e Promtail *containerizzate* attualmente in esecuzione ed aggiornava il file di configurazione di **NGINX** per permettere l'accesso alle utenze esterne ai container in esecuzione sull'host.
Per raggiungere questo obiettivo, lo stagista ha dovuto apprendere in maniera approfondita le tecnologie di *containerizzazione* e di *orchestrazione* di container, quali Docker e Docker Compose, unite ad elementi di *networking* generale e *scripting* in **Powershell** e **CMD**.
Verso la conclusione dello stage, lo stagista ha dovuto studiare la soluzione di **reverse proxy "NGINX"**, editandone, dapprima manualmente, tutti i file di configurazione e studiando approfonditamente la struttura di ogni file di configurazione al fine di automatizzare il tutto tramite script Powershell "**nginxREscript.ps1**".

2.2 Requisiti e obiettivi

Come già spiegato nell'introduzione di questo documento, causa mancanza di tempo e **sotto totale suggerimento** dell'Azienda, lo stagista si è dedicato alla creazione dell'immagine *containerizzata* del solo prodotto "**HDA**", **escludendo** quindi il software "**CX Studio**", creando, in aggiunta, **sei** diversi container aggiuntivi per permettere una soluzione di *reverse-proxy* e di monitoraggio in real-time dello stack applicativo containerizzato.
Causa questa modifica, i requisiti ed obiettivi dello stage curricolare sono così cambiati:

Requisito	Descrizione
Obbligatorio	Studio dell'architettura dello stack applicativo delle soluzioni software aziendali
Obbligatorio	Studio delle soluzioni software di containerizzazione ed orchestrazione, con focus sugli applicativi "Docker" e "Docker Compose"
Obbligatorio	Creazione del primo container contenente un'immagine di HDA funzionante ed usabile
Obbligatorio	Creazione di un container atto all'aggiornamento del database dell'applicativo HDA in caso di aggiornamento da una versione legacy di HDA ad una versione 11.x.x
Obbligatorio	Creazione, configurazione ed orchestrazione di un container dedicato all'applicativo "Loki"
Obbligatorio	Creazione, configurazione ed orchestrazione di un container dedicato all'applicativo "Promtail"
Obbligatorio	Aggiornamento del container contenente l'immagine di HDA con aggiunta e configurazione dell'applicativo "Telegraf"
Obbligatorio	Studio delle Docker API per riuscire ad avere la lista dei container attivi in un dato host
Obbligatorio	Creazione di uno script automatizzato che, tramite Docker API, ottiene le sandbox applicative di container attivi per la costruzione automatizzata del file di configurazione di NGINX (nginx.conf)

Tabella 2.1: Tabella dei requisiti.

Tutti i requisiti sopra-citati sono stati **completamente raggiunti e collaudati**, come verrà spiegato successivamente nel corso di questo documento.

2.3 Pianificazione

La pianificazione atta al soddisfacimento dei requisiti scritti è stata strettamente dettata dall'Azienda, ed è stata seguita in maniera pedissequa durante tutto il corso dell'esperienza di stage. Tutte le attività pianificate, con il rispettivo corrispettivo orario, possono essere riassunte nella seguente tabella:

Descrizione	**Quantitativo orario (h)**
Introduzione al tema Container vs Virtual machine: differenze tra le due tecnologie e illustrazione dei vantaggi derivanti dall'adozione dei Container	40
Docker ed estensioni (Docker, Compose e Swarm, Kubernetes) e relative API di automation: analisi delle componenti dell'ecosistema e delle opportunità di utilizzarle ai fini progettuali	40
Approfondimento sull'architettura su due casi studi da containerizzare: declinazione della soluzione tecnologica identificata ai punti precedenti su due applicazioni PAT	120
Creazione dei container ed automatizzazione del processo di building: realizzazione del processo di creazione delle immagini ed automazione dello stesso	80
Utilizzo di un container per verifica dell'esecuzione degli unit test: avvio delle immagini per effettuazione degli unit test in automatico	40
TOTALE ORARIO (h)	**320**

Tabella 2.2: Tabella relativa alla pianificazione dell'attività di stage.

Capitolo 3

Container VS Virtual Machine

Introduzione al capitolo
Nel presente capitolo si esporranno le principali differenze tra un'architettura basata su macchine virtuali ed un'altra basata invece su container, analizzando i pro ed i contro di entrambe le architetture e fornendo al lettore una panoramica sulla tecnologia di containerizzazione utilizzata in ambito aziendale.

3.1 Differenze architetturali tra Container e VM

La **virtualizzazione** è un insieme di software in grado di astrarre componenti hardware, permettendo l'esecuzione, anche simultanea, di più sistemi operativi su un singolo client. Verso la fine degli anni '90, la virtualizzazione ha cominciato ad essere sempre più utilizzata in ambienti *enterprise*, permettendo un aumento di scalabilità e flessibilità dell'infrastruttura informatica aziendale riducendone notevolmente i costi di gestione[1].
I **vantaggi** legati all'utilizzo della virtualizzazione, nello specifico, tramite l'uso di una o più macchine virtuali, comportano una separazione tra il sistema operativo host e guest, fornendo una serie di accessi logici utilizzati da utenti esterni agli applicativi eseguiti in ogni macchina virtuale.
Oltre ad una esecuzione parallela, dal punto di vista dell' *host*, un'architettura a VM è più facilmente **manutenibile**: una macchina virtuale infatti, può essere facilmente aggiornata, avviata o arrestata in base alle esigenze di carico (ex: *load-balancing*) o aziendali. La virtualizzazione, inoltre, aumenta l'**affidabilità** dell'intero sistema, in quanto garantisce l'**isolamento** di programmi e servizi i quali non andranno in conflitto tra di loro, contenendo, in aggiunta, il numero di server fisici presenti in datacenter nel caso in cui molteplici macchine virtuali vengano eseguite su un singolo *host*, con conseguente notevole riduzione dei costi legati all'hardware.
Un ulteriore vantaggio della virtualizzazione si rivela in caso di *disaster recovery*, dove l'intero sistema operativo *guest* può essere facilmente ripristinato su un altro server, indipendentemente dall'*hardware*, riducendo così notevolmente i tempi di indisponibilità di servizio (*downtime*) favorendo una maggior facile e rapida procedura di data *recovery*.
Esistono diversi tipi di virtualizzazione: ***native*** e ***hosted***. Una virtualizzazione di tipo **native** si appoggia direttamente all'*hardware host*, controllandolo direttamente per garantire tutte le funzionalità della virtualizzazione. Un esempio di applicativo

[1] **fonte: https://www.vmware.com/it/solutions/virtualization.html**.

dedicato alla virtualizzazione di tupo *hosted* può essere, ad esempio, **Hyper-V** della Microsoft[2] oppure l'applicativo **Xen** ampiamente utilizzato anche nell'ambiente Cloud di Amazon. La virtualizzazione di tipo **hosted** è invece in esecuzione sul sistema operativo *host* senza alcuna interfaccia diretta con l'hardware del computer. Questo tipo di virtualizzazione è molto diffusa, in quanto permette di accedere, in una maniera semplice ed immediata, al sistema operativo *host* e *guest* in maniera simultanea. Gli applicativi più usati in ambito **enterprise** che usano un tipo di virtualizzazione *hosted* sono, ad esempio, *VMware* o il gratuito *VirtualBox*.
Nella seguente figura è rappresentato il sistema operativo Windows 10 Pro virtualizzato con in esecuzione il browser web "Mozilla Firefox" tramite le due soluzioni software di virtualizzazione *hosted* appena descritte:

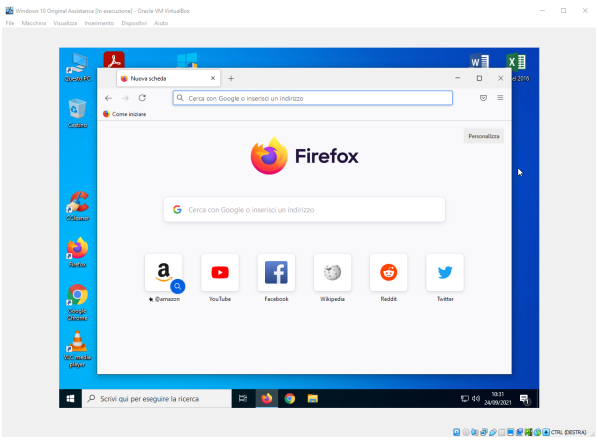


Figura 3.1: Browser "Firefox" virtualizzato tramite software di virtualizzazione "VirtualBox"

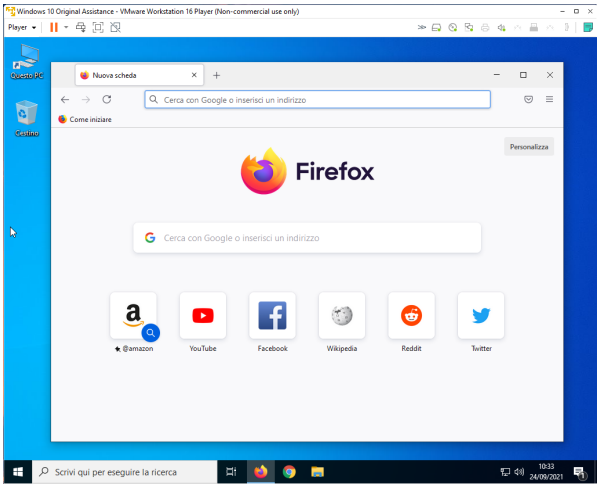


Figura 3.2: Browser "Firefox" virtualizzato tramite software di virtualizzazione "VMWare"

[2] **questa funzionalita' e' presente solamente nelle versioni Pro e Server di Windows 10.**

Al fine di permettere al sistema operativo *host* la virtualizzazione di uno o più sistemi operativi, è necessario installare un *hypervisor*[3], *native* o *hosted*, ovvero uno strato software che si interfacci e gestisca tutte le **istanze** di macchine virtuali in esecuzione sulla macchina locale.
La virtualizzazione non è priva di svantaggi. Il primo tra tutti, è appunto la necessità di dover *virtualizzare* un intero sistema operativo al fine di eseguire l'applicativo virtuale desiderato. Questo vincolo obbligatorio implica un consumo di memoria RAM e di storage non indifferente anche solo per eseguire il singolo sistema operativo virtualizzato **senza alcuna** applicazione virtuale in esecuzione. Ne consegue quindi, che un'architettura a macchine virtuali avrà bisogno di uno spazio di *storage* e di un quantitativo di memoria *RAM*[4] installata sul server non indifferente. Anche in termini di consumo *CPU*, la virtualizzazione di molteplici sistemi operativi con le relative applicazioni virtualizzate in esecuzione può comportare **grossi carichi prestazionali** al server fisico, in quanto la CPU dell'host dovrà servire ed eseguire ogni sistema operativo di **ogni istanza** di virtualizzazione.
Dal punto di vista della sicurezza, quando si virtualizza un sistema operativo, sia nella virtualizzazione *native* che *hosted*, alcuni registri CPU sono direttamente esposti alla macchina virtuale come, ad esempio, i registri **VT-x** e **VT-d** del processore[5][6]. Questi registri permettono al processore di non rendere accessibile la totalità dei suoi registri all'**hypervisor** e di controllare le chiamate dirette al *DMA* da parte delle soluzioni software virtualizzate.
Relativamente alla condivisione della rete tra macchine virtuali e host fisico, nel caso in cui si fosse installato un commutatore di rete virtuale di tipo **NAT**, la scheda di rete dell'host e il relativo traffico sarebbe **esposta** a tutto il set applicativo virtualizzato e viceversa, con conseguente mancanza di isolamento tra macchine virtuali stesse ed *host* fisico. Ne conseguirebbe quindi, che eventuali condivisioni di rete, o connessioni applicative, sarebbero disponibili a **tutto il set** di macchine virtuali.
Una possibile soluzione a questo problema potrebbe essere il passaggio da commutatore virtuale di tipo *NAT* ad un commutatore virtuale che riesca ad isolare le singole macchine virtuali tra di esse e l'*host* fisico, anche, nel caso più estremo, assegnando ad ogni macchina virtuale una **propria scheda di rete** ed una **propria** *VLAN* di rete dedicata[7].
Virtualizzare un intero sistema operativo implica, come abbiamo appena analizzato, un **elevato consumo di risorse fisiche**, specialmente nel caso in cui, per esigenze lavorative, si debba ricorrere ad una multipla virtualizzazione di sistemi operativi dove, in ognuno di essi, viene eseguita una specifica applicazione che deve essere accessibile ad altri *client*.
Uno dei principali aspetti positivi di un'architettura a container sta proprio nel poter virtualizzare (o *containerizzare* nel caso appunto di container) una singola e specifica applicazione **senza la necessità di inglobare un intero sistema operativo** nell'immagine virtuale. L'esecuzione dell'applicativo, nonostante appunto la mancanza di un sistema operativo, sarà comunque possibile grazie a chiamate di sistema al kernel del sistema operativo *host*. Ne consegue quindi, che il container applicativo risultante di un'applicazione *containerizzata* è di gran lunga di **dimensione inferiore** rispetto all'

[3] **installabile solamente se il processore supporta la virtualizzazione e se quest'ultima e' abilitata da BIOS**.
[4] **il tipo di RAM "ECC" risulta preferibile ma non obbligatorio**.
[5] **e' necessario abilitare le estensioni di virtualizzazione da BIOS della scheda madre.**.
[6] **nel caso di architettura avente processori Intel; IOMMU per architetture basate su processori AMD.**.
[7] **per creare o impostare una VLAN; fare riferimento al router/firewall o allo switch di rete**.

immagine[8] della stessa applicazione *virtualizzata*, causa appunto, in primis, mancanza di sistema operativo.
Un container è quindi una **singola unità atomica** contenente l'applicativo (il programma *containerizzato*) con i relativi file atti alla sua corretta esecuzione senza l'immagine di un sistema operativo completo. Al momento dell'esecuzione del container, l'applicazione *containerizzata* verrà eseguita immediatamente sopra lo stato del sistema operativo *host*, attraverso l'aiuto del **Docker Engine**, senza alcun *hypervisor* come, ad esempio, nel caso dell'architettura a macchine virtuali.
Una rappresentazione grafica del concetto appena descritto è data dalla seguente immagine:

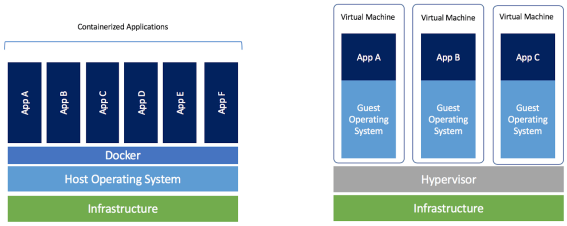


Figura 3.3: Rappresentazione grafica dell'architettura a container ed a macchine virtuali

Un'architettura a **container** infatti, a differenza dell'architettura a macchine virtuali, garantisce un'esecuzione **separata** e **protetta** di **ogni singolo applicativo** compatibile con il sistema operativo host, indipendentemente dal numero di container presenti nel sistema o dal tipo di interfaccia di rete.
E' possibile, inoltre, far coesistere multipli container di uno stesso applicativo in esecuzione nello stesso momento (anche sfruttando il *load-balancing*, come si accennerà nel corso di questa tesi) assegnandoci, esattamente come con le macchine virtuali, eventuali **indirizzi IP** statici, **CPU limit** e **disk quota**.
Essendo un container una **sandbox** applicativa indipendente dal sistema operativo, i dati generati dalla sua esecuzione sono destinati a scomparire nell'eventualità in cui il container venisse distrutto. Per ovviare a questo problema, si può ricorrere ad una tecnica di **volume-mapping**, ovvero una tecnica che permette di esporre il **filesystem** interno al container permettendone quindi la **lettura e scrittura** direttamente da parte dell'*host*. La tecnica appena accennata sarà trattata in maniera più approfondita nel corso della lettura di questa tesi.
Godendo i container di un approccio standardizzato per la loro costruzione ed esecuzione, è quindi di facile intuizione la facilità di *portabilità* di questi. Un container, infatti, può essere distribuito su una nuova piattaforma in maniera estremamente veloce e senza alcuna modifica allo stesso. Ne consegue, che un eventuale **rilascio e distribuzione** di un applicativo interno ad un container può essere molto velocizzato rispetto alla stessa operazione svolta invece su un'architettura a macchine virtuali. Relativamente sempre allo sviluppo e distribuzione di un applicativo, tramite container è possibile un controllo di versione dell'applicativo stesso più flessibile: è possibile infatti gestire le versioni del codice con le relative dipendenze inglobando il tutto in un unico container, formando quindi un' unità atomica di più facile manutenzione e distribuzione. Un eventuale ripristino o **rollback** delle modifiche potrà essere facilmente eseguito tramite

[8] **inteso come dimensione in Gb del virtual disk image (*.vdi) dell'immagine virtualizzata.**

ripristino in esecuzione del container contenente le modifiche precedentemente effettuate dal team.
Un altro dei vantaggi decisivi di un'architettura a container è la sua facilità di gestione. L'avvio, rimozione o la duplica dei container è un'operazione decisamente **meno onerosa** rispetto alla controparte nelle macchine virtuali (basti solo pensare al tempo di *boot* del sistema operativo), e può essere **facilmente automatizzata e gestita** dal Docker Engine. Ne consegue quindi, che la **scalabilità**, ovvero la facilità di modifica dell'infrastruttura per far fronte alle variazioni di mole di informazioni trattate o carichi di lavoro, risulta di **gestione più semplice** anche per la figura sistemistica interna all'azienda.
La presente immagine ritrae il browser web "Mozilla Firefox" in esecuzione all'interno di un container esposto all'*host* nella porta 5800:

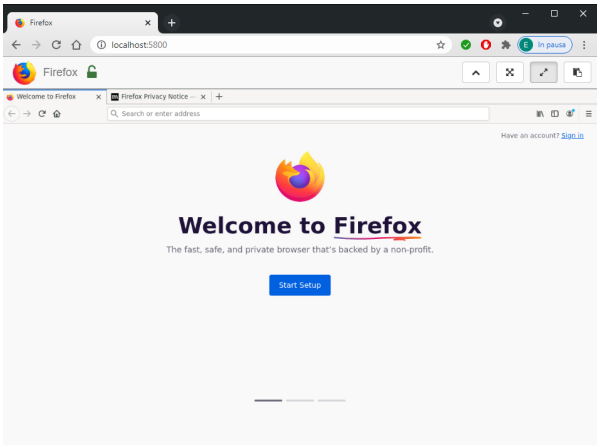


Figura 3.4: Browser web "Firefox" in esecuzione su un container con porta 5800 esposta all'host

3.2 Analisi di Docker: architettura e funzionalità

Una delle piattaforme di riferimento più utilizzate e supportate in ambito container è appunto **Docker**. Nato dall'azienda dotCloud nel 2003 e pubblicato successivamente come progetto **open-source**, è una piattaforma atta alla gestione dei container. Tramite **Docker Desktop**[9], è possibile monitorare, avviare, arrestare e gestire eventuali container creati.
Come già anticipato nel corso di questa tesi, Docker lavora a stretto contatto con il suo motore, ovvero Docker Engine, che si occupa appunto della **creazione** e **manutenzione** dei container nel sistema. Il *Docker Daemon* è il processo principale di Docker, cui compito è appunto la gestione e l'orchestrazione dei container nel sistema operativo dove è installato.
Al momento della creazione di un nuovo container, questo processo interroga i *Docker Registries* per verificare la presenza di una immagine già pronta all'uso del programma che si vuole *containerizzare*. I Docker Registries sono un insieme di immagini di container **pronte all'uso** e **scaricabili gratuitamente** da qualunque client Docker. Queste immagini sono create e manutenzionate dalla **Docker Community**, e permettono allo sviluppatore di avere delle immagini (o *layer*) già pronti, testati e funzionanti al fine di costruire con successo la propria immagine di applicativo virtuale. Il Docker Registries più comune ed utilizzato è "**DockerHub**".
Come già menzionato nel corso di questo documento, tramite Docker è possibile costruire delle immagini di un applicativo virtuale. Un' immagine è un **set di comandi** cui scopo è quello di creare, una volta eseguito, un container. Un' immagine può basarsi, a sua volta, su altre immagini, ed il set di comandi che espandono l'immagine sorgente compongono, a loro volta, la nuova immagine applicativa. I comandi atti alla corretta costruzione della stessa vengono scritti su un apposito file, chiamato appunto "**dockerfile**", ed ogni comando (riga del file) si traduce in un nuovo *layer* della nuova immagine. Un **dockerfile** è un documento di testo, senza alcuna estensione, contenente una serie di passi ed istruzioni atti alla **corretta creazione** di un' immagine di un applicativo. Il Docker Daemon, per costruire correttamente un' immagine, leggerà ed eseguirà in maniera **sequenziale** ogni comando trascritto all'interno del *dockerfile*, ed al termine della lettura dello stesso, si avrà un'immagine eseguibile di un applicativo.
Un **container** è un'**istanza di esecuzione** di un'immagine a cui è attribuito un nome, arbitrario o definito dal programmatore, attraverso il Docker Daemon.

3.3 Creazione di container vs creazione di VM

La creazione di una VM è possibile tramite apposito *hypervisor*. Tuttavia, al momento della creazione, l'utente deve essere a conoscenza della quantità massima di risorse *hardware* da destinare alla stessa. Questa stima, oltre ad includere il costo[10] richiesto dal programma da *virtualizzare* che si andrà ad installare all'interno della VM, deve includere anche il costo, almeno soddisfacente i requisiti minimi, relativo al sistema operativo. Terminata la fase di **analisi dei costi**, sarà necessario installare **manualmente** il sistema operativo all'interno della macchina virtuale appena creata. Per fare ciò, bisognerà avviare la macchina virtuale da un supporto di **boot**, come ad esempio una immagine ISO avviabile di un sistema operativo. Successivamente alla fase di avvio di installazione della VM, si passerà alla fase vera e propria di installazione

[9] **presente solamente per le versioni Windows**.
[10] **inteso come quantitativo di risorse fisiche da allocare**.

del sistema operativo.Una volta ottenuta un'installazione funzionante ed avviabile, per ottimizzarne al meglio le prestazioni, tramite ad esempio installazione dei relativi driver video virtuali, ed espanderne le funzionalità, come la possibilità di accedere a *volume-mapping* condivisi con il sistema operativo *host*, è necessario installare le relative **guest additions**, proprie di ciascun programma di virtualizzazione. Una volta fatti i passi sopra-descritti, si avrà un sistema operativo virtualizzato completamente funzionante, pronto per l'installazione di tutto il software applicativo desiderato.
Come precedentemente affermato, un container è un'unità atomica costituita principalmente da una o più applicazioni e dalle librerie di sistema operativo atte al loro corretto funzionamento. Al fine di poter generare ed utilizzare prima un'immagine applicativa e successivamente un container, è necessario prima costruirne il relativo **dockerfile**. Ogni *dockerfile* di ogni immagine applicativa contiene un set di istruzioni e comandi atti alla corretta installazione e configurazione dell'applicativo virtuale. Ogni comando scritto nel *dockerfile* comporrà un nuovo *layer* del container. La creazione di multipli container cloni, ovvero container generati da una singola immagine, è un'operazione di semplice e veloce esecuzione, in quanto il Docker Engine, durante la costruzione di ogni container, adotta un meccanismo di *caching* dei vari *layer* di cui un container è composto. Questo meccanismo permette quindi una costruzione, o aggiornamento, di un container **molto più veloce** rispetto ad una sua completa ri-creazione, in quanto si useranno, se compatibili, i layer in cache già **precedentemente costruiti** al fine di costruire il nuovo container contenente l'applicativo, nuovo o aggiornato, desiderato.
Durante tutta la durata dello stage si è utilizzata la piattaforma "Docker" per la creazione delle varie immagini e container per gli applicativi specificati nel sommario e nell'introduzione.

3.4 Installazione di Docker in ambiente Windows

Al fine di poter creare ed orchestrare dei container, è necessaria l'installazione del software "**Docker Desktop**" nel proprio computer. Docker Desktop è compatibile con tutte le versioni di Windows a partire da Windows 10 Home (build 19041), ed i requisiti minimi sono i seguenti:

* Processore a **64 bit**;
* 4Gb di memoria **RAM**;
* Virtualizzazione hardware abilitata da **BIOS**;
* Funzionalità **Hyper-V** abilitata da Windows[11].

Una volta scaricato l'installer di Docker Desktop, l'installazione è una procedura relativamente semplice, in quanto è del tipo "**one click-install**".
Durante tutta l'installazione è necessaria una connessione ad internet attiva.
Terminata l'installazione, per avviare Docker Desktop basterà aprire il menu **Start** di Windows e cliccare sulla relativa icona di Docker Desktop come nell'immagine:

[11] **Per abilitare il supporto di virtualizzazione Hyper-V su Windows bastera' entrare nell'apposito menu "Abilita o**

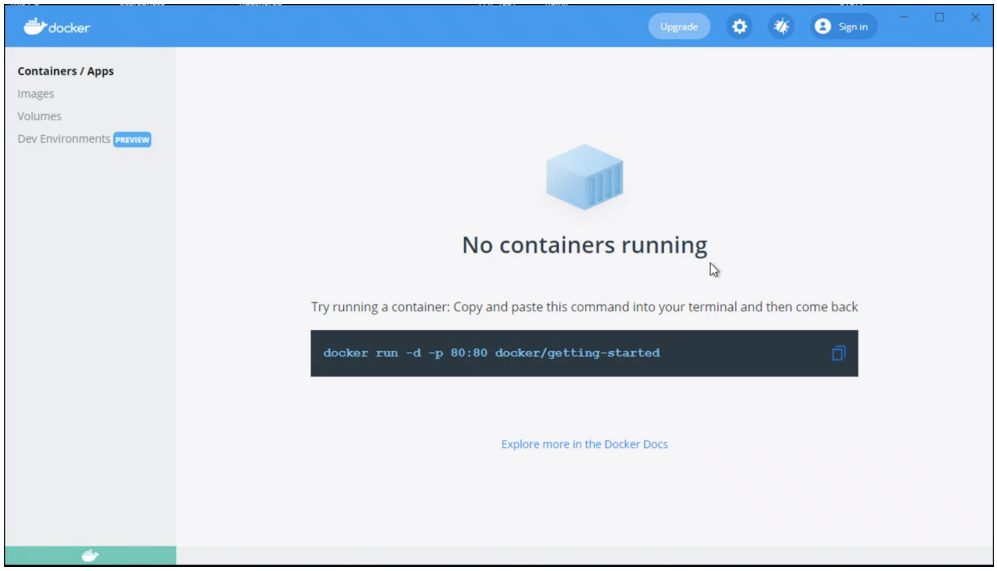


Figura 3.5: Schermata principale di "Docker Desktop" con nessun container in esecuzione

Capitolo 4

Architettura generale ed analisi container

Breve introduzione al capitolo
In questo capitolo si esporrà l'architettura generale del progetto implementata in Azienda, fornendo un'analisi dettagliata sulla composizione di ogni Dockerfile relativo ad ogni container. Infine, verrà esposto il meccanismo di costruzione automatizzato di una *sandbox* applicativa.

4.1 Architettura generale

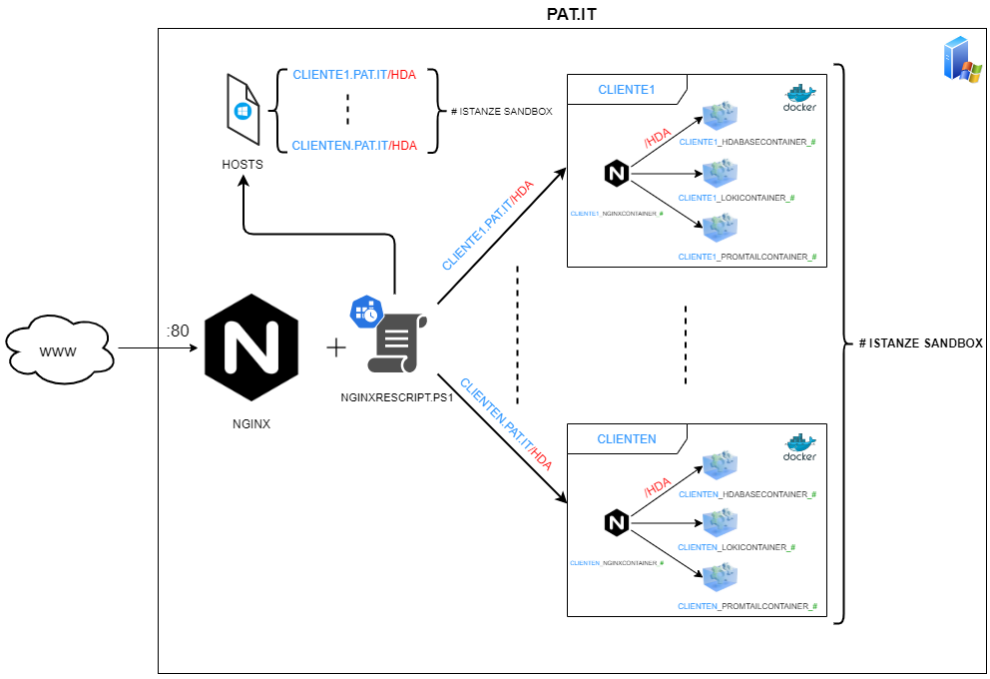


Figura 4.1: Architettura generale del progetto

La figura soprastante rappresenta l'architettura finale del prodotto interamente sviluppata ed installata con successo su un *host*. Una *sandbox* applicativa è un **insieme di container** in esecuzione, simultaneamente, per un singolo cliente. All'interno di un singolo host possono coesistere **multiple *sandbox*** applicative, lanciate manualmente dal tecnico installatore.
Una *sandbox* applicativa è composta dal seguente insieme di container:

* **nginxcontainer**;
* **hdabasecontainer**;
* **promtailcontainer**;
* **lokicontainer**;

ed è rappresentata nella figura 4.1 tramite il quadrato contenente appunto i container sopraelencati. Ad ogni *sandbox* è associato un **nome**, tipicamente proprio del cliente (nell'immagine di esempio ,"cliente1" o "clienten"), ed i container in essa contenuti acquisiscono, nel **prefisso**, anch'essi il nome della *sandbox*. Ogni cliente sarà associato univocamente ad **una sola *sandbox* applicativa**, quindi ogni cliente si collegherà solamente alla relativa *sandbox* contenente la versione di HDA (sita nel container "hdabasecontainer") appositamente *customizzata* per lo stesso. Il corretto instradamento della richiesta dell'utente esterno ad una determinata *sandbox* è gestita dall'istanza di **NGINX esterna** ad ogni singola *sandbox*, ma installata nel medesimo *host* (nell'immagine PAT.IT) d'esecuzione delle stesse.
All'arrivo di una richiesta di connessione **http/https** all'istanza di NGINX esterna attraverso la porta "80" o "443" nel caso di https, questo controllerà il proprio **file di configurazione** (nginx.conf) cercando una entry corrispondente al **sotto-dominio di terzo livello** contenuto nel **CNAME** della propria istanza di HDA (ex: **cliente1**.pat.it) alla quale l'utente vuole collegarsi. Se questa entry (cliente1) è presente all'interno del file di configurazione "nginx.conf", si **instraderà la richiesta** all'indirizzo IP della relativa *sandbox* in esecuzione sull'*host* "PAT.IT", nell'esempio, la *sandbox* denominata "**cliente1**.pat.it". Sarà poi compito dell'istanza di NGINX interna alla *sandbox* (ex: **cliente1_nginxcontainer_**1) instradare la richiesta, precedentemente arrivata dall'istanza NGINX esterna, al relativo container interno alla *sandbox* a seconda dei seguenti suffissi:

* /**HDAPortal** instraderà la chiamata al container "hdabasecontainer" e, quindi, all'istanza dell'applicativo HDA contenuto in esso;
* /**loki** instraderà la chiamata al container loki, per la visualizzazione via browser delle metriche di HDA;
* /**promtail** instraderà la chiamata al container "promtailcontainer" contenente un' istanza dell'applicativo "promtail" e si visualizzerà a video una serie di metriche relative all'applicativo HDA.

Un eventuale suffisso vuoto "/" instraderà, di default, la chiamata al container "**hdabasecontainer**". Il *discovery* di eventuali nuove *sandbox* applicative, o la rimozione di quelle non più attive dal file "host" del sistema operativo, è interamente gestita dallo **script Powershell "nginxREscript.ps1"**. Una panoramica dettagliata relativa alla costruzione e funzionamento di questo script è disponibile al capitolo 6.3 di questo documento.

4.2 Composizione container

Come già precedentemente spiegato in questo documento, un container è un'**unità atomica** contenente un **applicativo** con le relative **dipendenze** atte al suo corretto funzionamento. Ogni container, ad esclusione del container "nginxcontainer" si basa su un'immagine del sistema operativo "**Windows ServerCore IIS**".
La composizione di un container può essere più facilmente espressa tramite la seguente immagine:

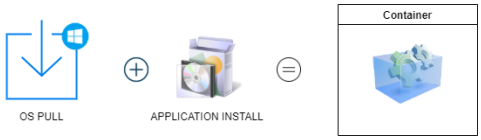


Figura 4.2: Rappresentazione grafica relativa alla composizione generale di un container

Per necessità progettuali che saranno descritte nel corso di questa relazione è stata necessaria la costruzione di **sei** diversi container. Di seguito è fornita al lettore una panoramica dettagliata sulla composizione di ognuno dei singoli:

4.2.1 Container "hdaprepcontainer":

S.O./immagine di base: Windows Servercore IIS

Immagine in output: hdaprepimg

Descrizione: Lo scopo del seguente container è quello di **installare un'istanza di HDA** all'interno di esso, popolando quindi il *volume-mapping* condiviso con l'*host* con tutti i file di HDA con i **relativi permessi** dell'utenza di **IIS** impostati in automatico dal processo di installazione di HDA lanciato dall'eseguibile "**update.exe**". Una volta terminata l'installazione, il container terminerà automaticamente la sua esecuzione. Per lanciare una istanza di HDA containerizzata bisognerà quindi eseguire il successivo container "hdabasecontainer" spiegato immediatamente.

4.2.2 Container "hdabasecontainer"

S.O./immagine di base: hdaprepimg

Immagine in output: hdabaseimg

Descrizione: Lo scopo del seguente container è quello di **avviare**, ed eventualmente **re-installare**, una istanza di HDA con i **relativi servizi**. Una volta fatto quanto specificato, a differenza del container "hdaprepcontainer", questo non interromperà la sua esecuzione, ma **rimarrà attivo** per permettere ad utenti esterni di usufruire dell'applicativo HDA **collegandosi alla web-interface** propria dell'istanza di HDA in esecuzione. In aggiunta a questa istanza, è presente una installazione dell'applicativo "Telegraf", ovvero un *agent* atto alla

raccolta di tutte le metriche prestazionali del container quali CPU usage, RAM usage e network usage.
Per favorire uno scambio di dati tra host e container, questo container si interfaccia direttamente con due *volume-mapping* condivisi con l'*host*:

* **hdashared**: è il *volume-mapping* che espone la cartella "**App_Data**" del programma HDA. Questo *volume-mapping* permette all'installatore di esportare od importare degli *overrides* applicativi per delle *customizzazioni* specifiche di ogni cliente **create ad-hoc** dal team di sviluppo di HDA sulla base delle esigenze del cliente stesso.
* **lokishared**: è il *volume-mapping* che espone tutti i log applicativi dell'istanza di HDA, come ad esempio *hda_log.txt*, *error_log.txt* e *wsc4_log.txt* al container "lokicontainer" atto al monitoraggio dell'istanza di HDA presente in questo container.

Questo container dipende (eredita) il *filesystem* ed i relativi permessi **precedentemente configurati** dal container "hdaprepcontainer".

4.2.3 Container "hdadbupdatercontainer"

S.O./immagine di base: hdabaseimg

Immagine in output: hdabasedbimg

Descrizione: Lo scopo del seguente container è quello di eseguire un aggiornamento del database di HDA **manualmente creato** dall'utente installatore all'interno di uno specifico server **MSSQL**. I parametri di connessione al database pre-esistente di HDA sono nel template-file "**istance.json**" presente nel pacchetto di installazione di HDA fornito. Il presente container non ha associato**alcun indirizzo IP** o scheda di rete virtuale, in quanto l'utente non ha la necessità di interfacciarcisi durante la sua esecuzione e può, inoltre, essere eseguito **indipendentemente** da qualsiasi altro container in esecuzione sullo stesso *host*. La versione di HDA presente nel container "hdaprepcontainer", e di conseguenza nel container "hdabasecontainer", deve essere compatibile con la versione del database installato tramite questo container. Il controllo di versione **non è automatizzato**, e lo dovrà quindi fare il tecnico installatore manualmente. La tabella informativa relativamente alla compatibilità tra versioni di HDA e database è presente all'interno del file OneNote aziendale.

4.2.4 Container "lokicontainer"

S.O./immagine di base: Windows Servercore IIS

Immagine in output: lokiimg

Descrizione: Lo scopo del seguente container è quello di eseguire un'istanza del programma "**Loki**" atto al monitoraggio dei **log** di HDA. Il presente container preleva i dati popolati dall'istanza di HDA in esecuzione nel container "hdabase-container" dal *volume-mapping* "lokishared" ad esso collegato, per permettere all'applicativo "Grafana", in esecuzione su un altro *host*, di mostrare graficamente le statistiche ed eventuali errori legati all'istanza di HDA in esecuzione sul container "hdabasecontainer".

4.2.5 Container "promtailcontainer"

S.O./immagine di base: Windows Servercore IIS

Immagine in output: promtailimg

Capitolo 5

Orchestrazione container

Breve introduzione al capitolo
In questo capitolo verrà esplicata la metodologia di orchestrazione tra container adottata in Azienda, fornendo al lettore, in primis, una panoramica su Docker Compose e, successivamente, l'integrazione dello stesso per la corretta costruzione di una *sandbox* applicativa di HDA.

5.1 Breve panoramica su Docker Compose

Docker Compose è uno strumento per la gestione ed orchestrazione di applicazioni Docker multi-container. In un ambiente costituito da multipli container in esecuzione, può essere relativamente difficile e gravoso per l'utente manutenzionare, avviare o fermare multipli container. Per aiutare appunto nell'orchestrazione, intesa come monitoraggio, avvio, ri-avvio in caso di *failure* e arresto di multipli container, l'utente può installare Docker Compose sulla propria workstation, per centralizzare così la gestione di tutti i container da un'unica interfaccia.
Tramite definizione di un file **YAML** (ex: *docker-compose.yml*), si può istruire Docker Compose relativamente a quali container dovrà avviare specificando, inoltre, eventuali parametri aggiuntivi, quali volume-mapping o interfacce di rete aggiuntive.
I passi che Docker Compose esegue per creare quindi la *sandbox* applicativa di HDA sono i seguenti:

* **lettura** del file docker-compose.yml;
* lettura, ricorsiva, dei **dockerfile** relativi ai **container** definiti nel docker-compose.yml;
* **costruzione**, secondo l'ordine ed eventuali parametri definiti nel docker-compose.yml, delle **immagini** dei container dichiarati nel Compose-file (docker-compose.yml);

Una volta costruite tutte le immagini dei container propri della sandbox applicativa di HDA, è possibile lanciare la sandbox applicativa tramite il semplice comando:

docker-compose -f docker-compose.yml -p nomecliente up

sostituendo "nomecliente" con il nome effettivo del cliente proprietario della *sandbox* di HDA. I parametri **-f** e **-p** definiscono, rispettivamente, la possibilità di

dichiarare un Compose-file alternativo e la possibilità di assegnare un nome alternativo all'istanza, la quale, di *default*, acquisirebbe il nome della directory corrente.

5.2 Docker compose per la costruzione di una sandbox applicativa

Come visto nel paragrafo precedente, tramite Docker Compose è possibile orchestrare l'avvio e arresto di una intera *sandbox* applicativa di HDA tramite un singolo comando. Tramite l'uso di Docker Compose, l'esecuzione **simultanea** di più *sandbox* applicative di HDA risulta di semplice esecuzione, in quanto basterà lanciare nuovamente il comando descritto nel paragrafo precedente per creare una nuova *sanbdbox* applicativa destinata ad un altro cliente. Ogni *sanbdbox* ha un nome che deve necessariamente essere **univoco**. La seguente immagine mostra una panoramica, fornita da Docker Compose, di due sandbox applicative, con i relativi container di cui composte, in esecuzione simultanea sullo stesso host, chiamate rispettivamente *nomecliente* e *cliente2*:

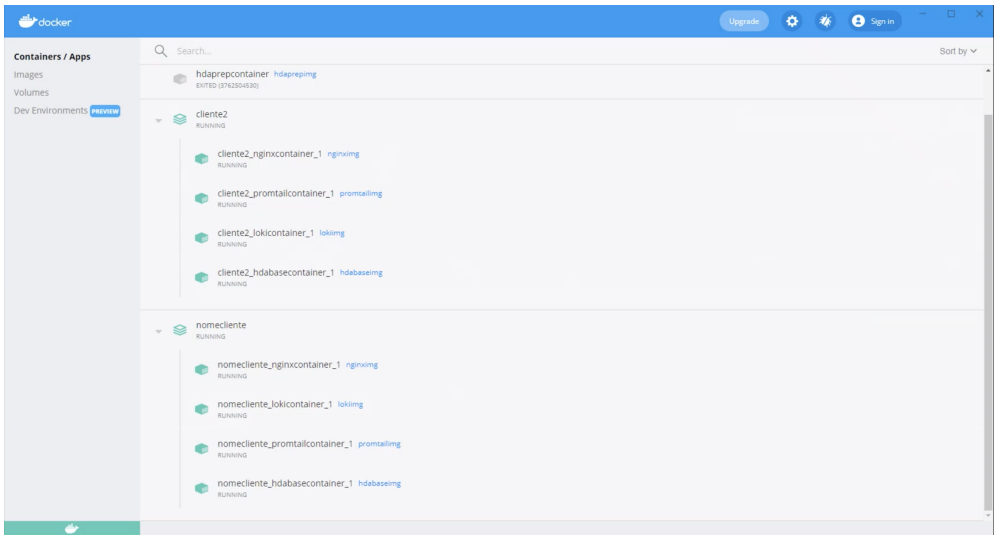


Figura 5.1: Interfaccia di Docker Compose con in esecuzione due *sandbox* applicative

5.3 Integrazione di HDA Sandbox Builder con Docker Compose

Come visto all'interno di questo capitolo, tramite Docker Compose è possibile la costruzione delle immagini relative ai container che comporranno, successivamente, una volta lanciati in esecuzione, la *sandbox* applicativa di HDA. Per arrivare ad una corretta esecuzione di una *sandbox* applicativa però, la costruzione tramite

Docker Compose dei container stessi che la compongono non è sufficiente. E' il caso, ad esempio, del container atto alla preparazione del filesystem di HDA ("hdaprepcontainer"), senza il quale l'istanza di HDA non riucirà ad avviarsi a causa di un settaggio errato dei permessi nella cartella /App_Data dello stesso. Questo container non è inserito all'interno all'interno del Compose-file, in quanto deve essere manualmente lanciato e supervisionato dall'utente installatore, e l'esecuzione del container dovrà terminare **prima** di poter lanciare una qualsiasi *sandbox* applicativa di HDA. Anche il container "hdadbupdatercontainer" è escluso dal Compose-file, in quanto, anch'esso, dovrà essere eseguito manualmente dall'utente installatore e continuamente supervisionato durante la sua completa esecuzione.
E' quindi facilmente intuibile, che l'esecuzione di Docker Compose dovrà essere necessariamente **subordinata** all'esecuzione del *batch-file* **HDA_sandbox_builder.bat**.
Riassumendo, i passi atti ad una corretta esecuzione di una *sandbox* applicativa di HDA sono i seguenti:

* **Passo 1:** lancio del *batch-file* HDA_sandbox_builder.bat, con costruzione ed esecuzione **obbligatoria** del container "hdaprepcontainer";
* **Passo 2:** creazione di una directory avente nome del cliente stesso che conterrà il filesystem della nuova *sandbox* applicativa di HDA generato dal passo precedente;
* **Passo 3:** da prompt dei comandi con privilegi di amministratore, **xcopy** dei file situati nella cartella App_Data del container "hdaprepcontainer" all'interno della directory creata al punto precedente;
* **Passo 4:** esecuzione del comando di Docker Compose "**docker-compose-f docker-compose.yml -p nomecliente up**" sostituendo al posto della stringa "nomecliente" il nome del cliente scelto nel passo 2.

Il corretto instradamento degli accessi da parte degli utenti alle istanze di HDA contenuta nel relativo container ("hdabasecontainer") all'interno delle *sandbox* applicative sarà trattato nel prossimo capitolo.

Capitolo 6

Reverse Proxy tramite NGINX sulle sandbox di HDA

Breve introduzione al capitolo
In questo capitolo verrà esplicata la metodologia di **reverse-proxy** tra *sandbox* applicative di HDA per premettere a diverse utenze esterne un accesso privato ad una singola istanza di HDA in esecuzione su un determinato host.

6.1 Introduzione ad NGINX

NGINX è un applicativo web-server multipiattaforma ad alte prestazioni, comunemente usato come reverse-proxy o load-balancer. Nato nel 2004 e progettato per garantire un basso consumo di memoria, utilizza un approccio asincrono, basato su *event*, dove le richieste vengono gestite su un singolo thread. Ogni *thread* worker, ovvero un processo che esegue un'elaborazione effettiva, è gestito da un *thread* master, il quale lo coordina e lo controlla.
All'interno di Docker-Hub è presente un repository ufficiale, gestito dalla Nginx Foundations, contenente un'immagine ufficiale di NGINX basata sul sistema operativo Alpine Linux, ovvero una distribuzione Linux che ha fatto della leggerezza un requisito fondamentale. Come precedentemente spiegato in questo documento, il container atto al reverse-proxy automatizzato contenente un'immagine di NGINX è chiamato "nginxcontainer", e fa parte anch'esso della *sandbox* applicativa di HDA.

6.2 Logica di reverse-proxy su sandbox di HDA

Affinchè un utente, interno od esterno alla *intranet* aziendale, possa accedere al relativo container contenente l'istanza, personalizzata o meno, di HDA, è necessario che conosca il relativo **FQDN** della *sandbox* applicativa di HDA alla quale si vuole accedere.
Un FQDN, acronimo di *Fully Qualified Domain Name*, è un nome di dominio che specifica la posizione assoluta di un nodo (in questo caso, la nostra

sandbox applicativa di HDA) all'interno della gerarchia DNS. Un FQDN si distingue da un generico nome di dominio per l'aggiunta del nome dell'host nel prefisso della stringa di dominio, in modo tale da renderla univoca. Un esempio di FQDN di un container è facilmente visibile nell'immagine esplicante l'architettura del progetto nel capitolo 4.1. Nonostante l'utente **non acceda in maniera diretta** al container contenente l'istanza di HDA, l'attributo FQDN di ogni container nelle relative *sandbox* è assegnato in maniera automatica da Docker Compose secondo la seguente logica decisa assieme all'Azienda:

nomecliente_nomecontainer_numeroistanza

dove:

* **nomecliente** identifica il nome della *sandbox* applicativa a cui il container appartiene. Generalmente, il nome della *sandbox* è il nome del cliente stesso;
* **nomecontainer** identifica il nome effettivo del container (ex: "hdabasecontainer") interno alla *sandbox* applicativa avente il nome del cliente;
* **numeroistanza** nell'immagine 4.1 identificato con il simbolo "#", identifica la quantità di istanze simili di un determinato container sono in esecuzione contemporaneamente sulla *sandbox* applicativa.

Una nomenclatura di container fissa e *standardizzata* secondo quanto deciso con l'Azienda è di fondamentale importanza, in quanto permette di instradare le richieste provenienti dall'esterno ad una qualsiasi *sandbox* applicativa di HDA. Per capire al meglio come questo processo funziona, è doveroso fornire in primis al lettore una panoramica grafica circa il funzionamento, con relativa spiegazione immediatamente sottostante:

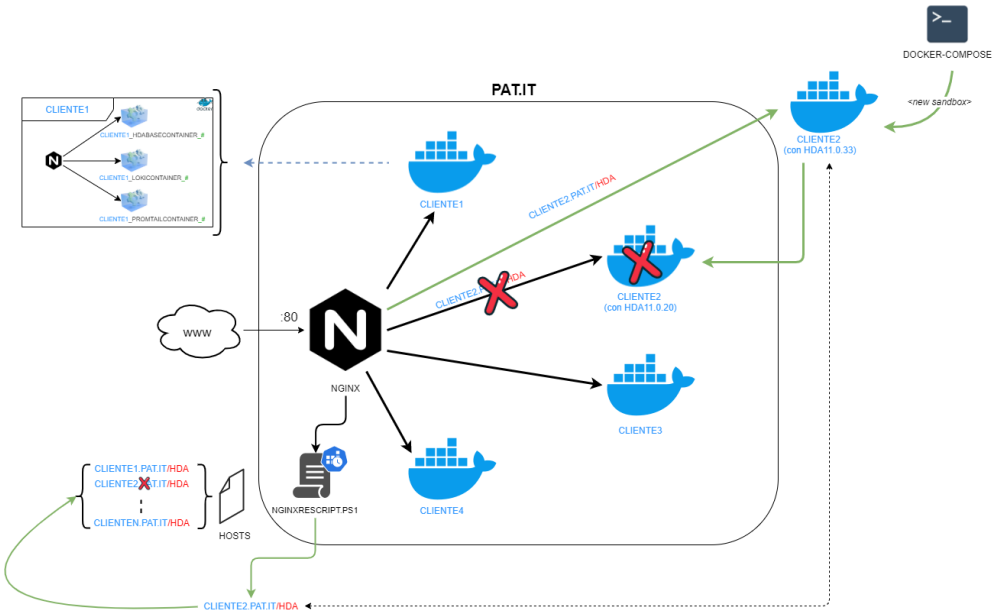


Figura 6.1: Rappresentazione grafica circa la gestione runtime delle *sandbox* applicative di HDA

Quanto sopra rappresentato, può essere facilmente riassunto nei seguenti passi:

1. Arrivo della richiesta HTTP esterna (ex: cliente1.pat.it) all'NGINX dell'**host *PAT.IT***;
2. Il sistema operativo dell'host *PAT.IT* controlla se l'FQDN della richiesta è presente all'interno del suo **file host**;
3. Il webserver NGINX controlla se l'entry "cliente1.pat.it" è presente nel suo **config file** (nginx.conf) e, se presente, provvede ad **inoltrare la richiesta** alla relativa *sandbox*;
4. La richiesta HTTP **arriva al webserver NGINX interno** alla relativa *sandbox* avente il nome uguale al prefisso della richiesta HTTP (ex: cliente1);
5. Il webserver NGINX interno alla *sandbox* applicativa **inoltra la richiesta** HTTP al container "**hdabasecontainer**", permettendo quindi agli utenti dell'azienda "cliente1" il **totale accesso all'istanza di HDA**;
6. Eventuali programmi esterni di monitoring in real-time (quali, ad esempio, Grafana) possono ora essere collegati alla *sandbox* applicativa semplicemente digitando nel relativo config-file (ex in Grafana: grafana.conf) l'**FQDN della relativa *sandbox* da monitorare** (ex: cliente1.pat.it). Sarà compito del relativo NGINX interno alla *sandbox* gestire e dirottare (tramite proprio file di configurazione nginx.conf) ai relativi container le richieste del programma di monitoring.

6.2.1 Aggiornamento di HDA nella sandbox applicativa

Come precedentemente accennato in questo documento, e come rappresentato sempre nell'immagine 6.1, risulta facilmente intuibile la fattibilità di un eventuale aggiornamento di una relativa *sandbox* applicativa. Per effettuare infatti un aggiornamento alla *sandbox* applicativa di HDA, sarà sufficiente eseguire i seguenti passi:

1. **Eliminazione** della *sandbox* applicativa contenente la versione legacy di HDA;
2. **Creazione** delle nuove immagini dei nuovi container della futura nuova *sandbox* applicativa di HDA tramite batch-file **HDA_sandbox_builder.bat**;
3. Effettuare la **migrazione manuale** di eventuali *overrides* o *extensions* contenuti nella cartella App_Data nel relativo volume-mapping nell'host ("hdashared");
4. **Avviare**, tramite comando Docker Compose, la nuova *sandbox* di HDA con il **nome uguale** alla *sandbox* appena sostituita.

Facendo questo, in maniera del tutto automatica, gli utenti accederanno così alla nuova istanza di HDA aggiornata.

Grazie al risolutore DNS interno a Docker ed al suo load-balancing automatico, nel caso di necessità di aggiornamento di **molteplici container** all'interno della *sandbox* applicativa di HDA, si potrà aggiornare relativamente la versione di HDA in ogni container **senza generare alcun downtime** al cliente, semplicemente aggiornando **un container alla volta**, per permettere quindi al load-balancer di

Docker di poter inoltrare tutte le richieste HTTP ad almeno un container attivo e funzionante di HDA. Il meccanismo di aggiornamento di multipli container all'interno di una singola *sandbox* applicativa non era nei requisiti di questo stage né è stato in alcun modo testato dallo stagista durante tutto il periodo di stage. Questa possibilità di aggiornamento e di load-balancing dei container di HDA è stata sviluppata a livello completamente teorico dal sottoscritto e dal proprio tutor Ruggero Maffei con il preziosissimo aiuto del Sig. Adriano Trevisan.

6.3 Analisi NginxREScript.ps1

Nel caso in cui il lancio o rimozione di *sandbox* applicative possa essere molto frequente, per evitare all'utente installatore la continua modifica dell'host file e del file di configurazione di NGINX, è stato predisposto uno script automatizzato che modifica, ad ogni sua esecuzione, il config file dell'**istanza di NGINX esterna** alle *sandbox* applicative ed il **file host del server** dove è installata l'architettura (ex: PAT.IT).
Questo script Powershell, tramite interrogazione delle Docker API, è in grado di rilevare ogni *sandbox* applicativa di HDA in esecuzione nell'host generando, in maniera dinamica, il file di configurazione dell'istanza di NGINX (nginx.conf) esterna alle *sandbox* per permettere il corretto puntamento, e quindi una corretta connessione da parte degli utenti, alle nuove *sandbox* applicative appena create, o dismettere i vecchi puntamenti alle *sandbox* obsolete od eliminate. Un flow-chart riassuntivo, per motivi di spazio, relativo allo script Powershell "**nginxREscript.ps1**" è il seguente:

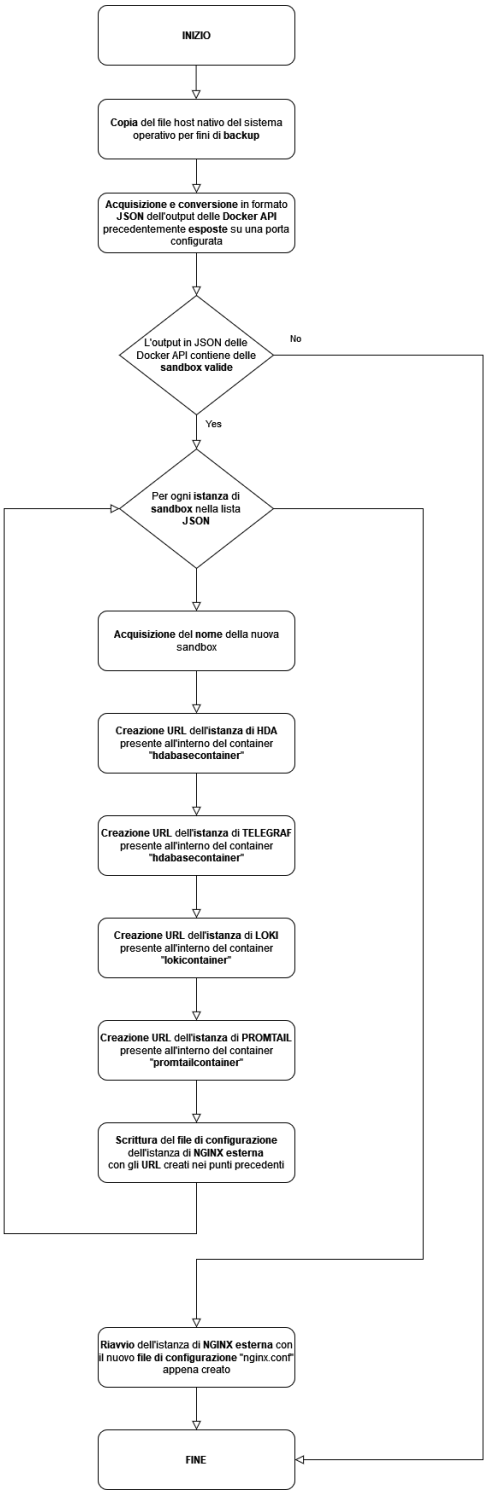


Figura 6.2: Flowchart riassuntivo relativo allo script Powershell nginxREscript.ps1

Capitolo 7

Conclusioni

Breve introduzione al capitolo Il presente capitolo finale vuole esporre una panoramica generale sul raggiungimento generale degli obiettivi dello stage curricolare, valutando le conoscenze acquisite dallo stagista e la qualità generale dell'esperienza di stage effettuata all'interno dell'Azienda.

7.1 Raggiungimento degli obiettivi

Come riportato nel capitolo 2.2, causa mole di lavoro troppo impegnativa in rapporto al tempo già rilevato ad inizio del percorso di stage, sotto completo suggerimento del tutor aziendale Sig. Maffei Ruggero, è stato consigliato al candidato di concentrarsi esclusivamente sulla creazione di una immagine *Dockerizzata* del solo prodotto HDA con le relative dipendenze come descritto nell'apposito capitolo prima citato. Avendo l'Azienda stessa modificato già ad inizio stage la lista degli obiettivi obbligatori e desiderabili che il candidato avrebbe dovuto portare a termine, lo stagista è riuscito a portare a termine ed a soddisfare **tutti** gli obiettivi ed i requisiti con una settimana d'anticipo, **ad eccezione** del requisito che richiedeva la creazione di un container per l'esecuzione degli **unit test** automatici delle immagini create. La tabella sottostante rappresenta, graficamente, gli obiettivi e requisiti con il relativo stato di raggiungimento (R) o non raggiungimento (NR):

Descrizione requisito	**Quantitativo orario (h)**	**Stato**
Introduzione al tema Container vs Virtual machine: differenze tra le due tecnologie e illustrazione dei vantaggi derivanti dall'adozione dei Container	40	R
Docker ed estensioni (Docker, Compose e Swarm, Kubernetes) e relative API di automation: analisi delle componenti dell'ecosistema e delle opportunità di utilizzarle ai fini progettuali	40	R
Approfondimento sull'architettura su due casi studi da containerizzare: declinazione della soluzione tecnologica identificata ai punti precedenti su due applicazioni PAT	120	R
Creazione dei container ed automatizzazione del processo di building: realizzazione del processo di creazione delle immagini ed automazione dello stesso	80	R
Utilizzo di un container per verifica dell'esecuzione degli unit test: avvio delle immagini per effettuazione degli unit test in automatico	40	NR
TOTALE	**320**	**4/5**

La settimana avanzata è stata usata dal candidato per la scrittura di tutta la documentazione di supporto interna all'Azienda, per permettere ad eventuali dipendenti di capire ed ampliare in futuro quanto sviluppato durante tutta l'esperienza di stage. La documentazione redatta è stata inserita nell'apposita sezione OneNote dell'Azienda assieme a tutta la documentazione interna dei loro prodotti.

7.2 Conoscenze acquisite

Le conoscenze conseguite a seguito dell'esperienza di stage sono state molteplici, in primis tra tutte la tecnologia che ho implementato. L'aver studiato in maniera approfondita la tecnologia Docker mi ha permesso di rendermi conto dell'esistenza di una tecnologia completamente alternativa, e per molti aspetti altrettanto valida, alle classiche architetture a macchine virtuali, con conseguente mia possibilità di ulteriore approfondimento e studio individuale futuro nell'ottica di poter proporre, alle future aziende con le quali collaborerò, delle soluzioni architetturali non più basate solo ed esclusivamente sulle virtual machine. A livello umano, ho trovato persone estremamente preparate e professionali con cui sono tutt'ora in contatto. Queste persone hanno contribuito alla mia crescita interiore facendomi rendere conto che, in un clima di reciproca fiducia e collaborazione, qualsiasi progetto, se ben supportato da colleghi attenti e preparati, potrà essere sviluppato con successo senza un rischio elevato di fallimento immediato ed a lungo termine. Avendo lavorato in un team così dinamico, ho inoltre rivisto la mia metodologia di lavoro, tipicamente molto autonoma, nell'ottica di una mia sempre più aperta visione ad eventuali osservazioni ed insegnamenti dettati da dipendenti con molta più preparazione ed esperienza alle loro spalle rispetto al sottoscritto. Questi insegnamenti rientreranno senz'altro nel mio bagaglio culturale, nella speranza di *saperli* e *poterli* attuare in ulteriori esperienze lavorative.

7.3 Valutazione personale

La mia personale opinione su questa esperienza di stage non può che essere assolutamente positiva. Ho avuto l'opportunità di lavorare a contatto con un team altamente qualificato, e di conoscere persone molto competenti nel loro ambito che spero continuino a far parte della mia rete di amicizie. L'aver fatto questa esperienza di stage mi ha, inoltre, permesso di apprendere com'è organizzata internamente un'azienda informatica di medie dimensioni. L'avermi dovuto rapportare con diverse persone di grado differente ha fatto sì che abbia potuto mettere alla prova la mia capacità di relazionarmi con le persone, diversificando, naturalmente, la mia metodologia di interlocuzione a seconda del ruolo del componente dell'Azienda con cui mi stavo confrontando.
Sempre grazie a questa esperienza di tirocinio, ho imparato a gestire il mio tempo in maniera nettamente migliore rispetto a tutto il percorso universitario fatto. Il dovermi rapportare con scadenze settimanali e vincoli, obbligatori e non, hanno contribuito a sviluppare in me una capacità migliore di gestione temporale, facendomi imparare, in primis, che il tempo utilizzato per lo studio di una particolare tecnologia è forse la parte più importante di un progetto stesso, e che quindi va eseguita senza alcuna fretta o pregiudizio.

Appendice A

Appendice A

Citazione

Autore della citazione

Bibliografia