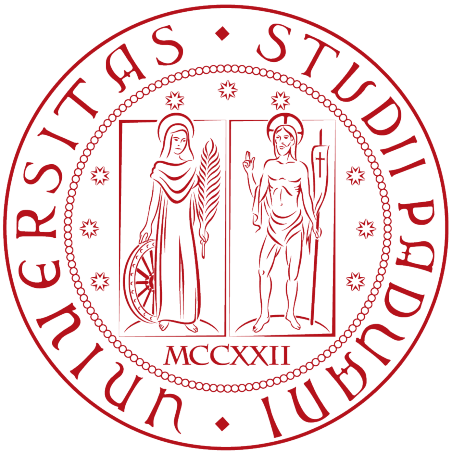
Università degli Studi di Padova

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. Lo scopo finale dello studio relativo alle due tecnologie appena citate 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 assieme a 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 della possibilità di automazione di quest'ultimo.

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

— Confucius

Ringraziamenti

Innanzitutto, vorrei esprimere la mia gratitudine al Prof. NomeDelProfessore, 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

Introduzione al contesto applicativo.

Esempio di utilizzo di un termine nel glossario Application Program Interface (API).

Esempio di citazione in linea **site:agile-manifesto**.

Esempio di citazione nel pie' di pagina citazione[1]

1.1 L'azienda

Descrizione dell'azienda.

1.2 L'idea

Introduzione all'idea dello stage.

1.3 Organizzazione del testo

Il secondo capitolo descrive ...

Il terzo capitolo approfondisce ...

Il quarto capitolo approfondisce ...

Il quinto capitolo approfondisce ...

Il sesto capitolo approfondisce ...

Nel settimo capitolo descrive ...

[1] **womak:lean-thinking**.

Riguardo la stesura del testo, relativamente al documento sono state adottate le seguenti convenzioni tipografiche:

* gli acronimi, le abbreviazioni e i termini ambigui o di uso non comune menzionati vengono definiti nel glossario, situato alla fine del presente documento;
* per la prima occorrenza dei termini riportati nel glossario viene utilizzata la seguente nomenclatura: *parola*[g];
* i termini in lingua straniera o facenti parti del gergo tecnico sono evidenziati con il carattere *corsivo*.

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. 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.

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, tre 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:

* Studio dell'architettura dello stack applicativo delle soluzioni software aziendali;

- Studio delle soluzioni software di containerizzazione ed orchestrazione, con focus sugli applicativi "Docker" e "Docker Compose";
- Creazione del primo container contenente un'immagine di HDA funzionante ed usabile;
- 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;
- Creazione, configurazione ed orchestrazione di un container dedicato all'applicativo "LokI";
- Creazione, configurazione ed orchestrazione di un container dedicato all'applicativo "Promtail"
- Aggiornamento del container contenente l'immagine di HDA con aggiunta e configurazione dell'applicativo "Telegraf";
- Studio delle Docker API per riuscire ad avere la lista dei container attivi in un dato host;
- 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).

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	**320**

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*, di molteplici applicativi, 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. Un ulteriore vantaggio della virtualizzazione si rivela nel 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*) in caso di guasto 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, come ad esempio **Hyper-V**

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

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 *VirtualBox*.
Nella seguente figura è rappresentato un sistema operativo Windows 10 pro virtualizzato con in esecuzione il browser web "Firefox" in virtualizzazione *native* tramite software Hyper-V: Di seguito, un esempio di sistema operativo Windows 10 Pro virtualizzato con in esecuzione il browser web "Firefox" in virtualizzazione *hosted* tramite le due soluzioni software appena descritte: Nello screenshot sottostante, invece, l'applicativo "Firefox" è eseguito tramite il programma di virtualizzazione "VMware":

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 mancante 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 *NAT* ad un tipo di 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].

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

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, 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 s.o. host. Ne consegue quindi, che il container applicativo risultante di un'applicazione *containerizzata* è di gran lunga di dimensione inferiore rispetto all' 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: 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 velicizzato rispetto alla stessa operazione svolta invece su un'architettura virtuale. 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 container, formando quindi un' unità atomica di più facile manutenzione e distribuzione. 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 relativamente 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

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

o carichi di lavoro, risulta di gestione più semplice anche per la figura sistemistica interna all'azienda.
La presente immagine ritrae l'applicativo "Firefox" in esecuzione all'interno di un container esposto all'host nella porta 5800:

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, dal Docker Daemon.

Tramite Docker è quindi possibile l'organizzazione e orchestrazione di tutti i container applicativi installati sul client.

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

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

operativo. La seguente schermata rappresenta la creazione della macchina virtuale con 8Gb di RAM e 80Gb di disco rigido dedicati:
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 di un sistema operativo, avviabile. Successivamente alla fase di avvio di installazione della VM, si passerà alla fase vera e propria di installazione del sistema operativo, riassunta nelle seguenti immagini: Una volta ottenuto un sistema operativo 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**. 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 correttamente un container, è necessario prima costruirne il relativo **dockerfile**. Le seguenti immagini mostrano i corretti passi per la costruzione di un container con, all'interno, l'applicativo "Firefox": 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, con relativi 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 (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.

Per abilitare il supporto di virtualizzazione Hyper-V su Windows basterà entrare nell'apposito menu "Abilita o disabilita funzionalità di Windows" e selezionare "Hyper-V" come da seguente immagine:
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:

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

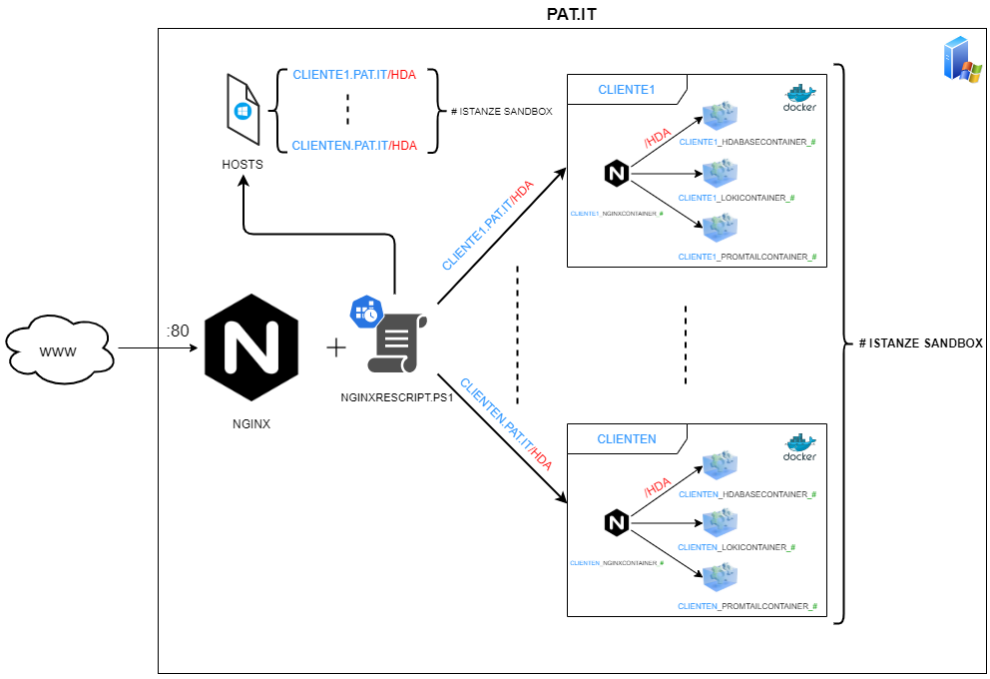


Figura 4.1: Architettura generale del progetto

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 in figura 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 esterna HTTP/HTTPS di connessione all'istanza di NGINX esterna attraverso la porta "80" o "443" nel caso di HTTPS, questo controllerà il relativo file di configurazione (nginx.conf) cercando una entry corrispondente al sotto-dominio di primo 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 "nginx.conf", si instraderà la richiesta all'indirizzo IP, letto nel file "host" del computer "PAT.IT", della relativa *sandbox* appartenente al cliente (la *sandbox* appunto "**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 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: 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 una 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 del programma HDA collegandosi alla web-interface propria dell'istanza di HDA in esecuzione. In aggiunta a questa istanza, è presente una installazione dell'applicativo "Telegraf", atto al monitoraggio in real time dell'istanza di HDA in esecuzione. Per favorire uno scambio di dati tra host 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 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*, *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 all'interno di uno specifico server **MS-SQL**. 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 una istanza del programma "Loki" atto al monitoraggio dei servizi di HDA. Il presente container preleva i dati popolati dall'istanza di HDA in esecuzione nel container "hdabasecontainer" nel volume mapping "lokishared" ad esso collegato, per inviarli direttamente al container "promtailcontainer".

4.2.5 Container "promtailcontainer"

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

Immagine in output: promtailimg

Descrizione: Lo scopo del seguente container è quello di eseguire una istanza del programma "Promtail" atto al monitoraggio dei servizi di HDA in esecuzione sul container "hdabasecontainer". Il seguente container espone i log collezionati dal container "lokicontainer" su una specifica porta precedentemente configurata da apposito file di configurazione "*promtail-local-config.yml*" 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".

4.2.6 Container "nginxcontainer"

S.O./immagine di base: NGINX official image

Immagine in output: nginximg

Descrizione: Lo scopo del seguente container è quello di eseguire una istanza del programma "NGINX" con il relativo file di configurazione "*nginx.conf* automaticamente importato. Tramite il resolver DNS interno di Docker, è possibile per un utente esterno, tramite NGINX, accedere ad un qualsiasi container facente parte della *sandbox* applicativa di HDA, quali "hdabasecontainer", "lokicontainer" e "promtailcontainer". Sempre tramite questo container, è possibile la gestione precedentemente accennata di load-balancing tra molteplici container di HDA.

4.3 Dockerfile container: analisi struttura e spiegazione

Come già visto precedentemente, ogni container è generato da un proprio dockerfile, il quale contiene tutte le istruzioni e comandi atti alla corretta costruzione di una immagine di un applicativo containerizzato. Di seguito, tramite l'ausilio di low-chart riassuntivi, viene fornita al lettore una panoramica sul contenuto di ogni dockerfile del progetto:

4.3.1 Dockerfile container: "hdaprepcontainer"

Il presente dockerfile costruisce con successo l'immagine relativa al container "hdaprepcontainer". La sequenza di passi per l'ottenimento dell'immagine è di seguito rappresentata:

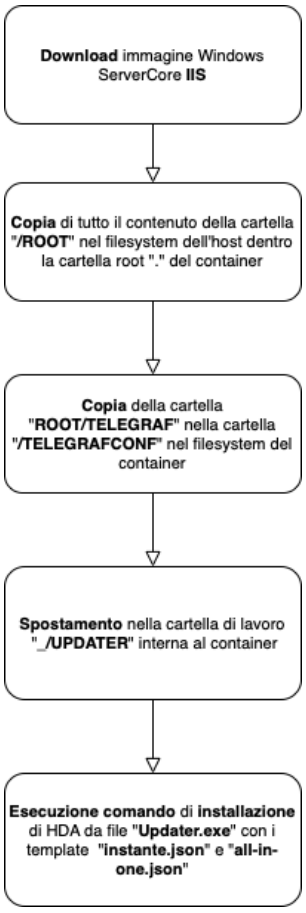


Figura 4.2: Flow-chart rappresentante la costruzione dell'immagine relativa al container "hdaprepcontainer"

Questo container non rimane in esecuzione dopo la fine dell'ultima istruzione, e non fa parte della *sandbox* applicativa.

4.3.2 Dockerfile container: "hdadbupdatercontainer"

Il presente dockerfile costruisce con successo l'immagine relativa al container "hdadbupdatercontainer". La sequenza di passi per l'ottenimento dell'immagine è di seguito rappresentata:

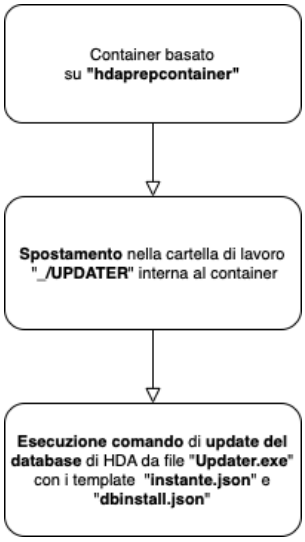


Figura 4.3: Flow-chart rappresentante la costruzione dell'immagine relativa al container "hdadbupdatercontainer"

Questo container non rimane in esecuzione dopo la fine dell'ultima istruzione, e non fa parte della *sandbox* applicativa.

4.3.3 Dockerfile container: "hdabasecontainer"

Il presente dockerfile costruisce con successo l'immagine relativa al container "hdabasecontainer". La sequenza di passi per l'ottenimento dell'immagine è di seguito rappresentata:

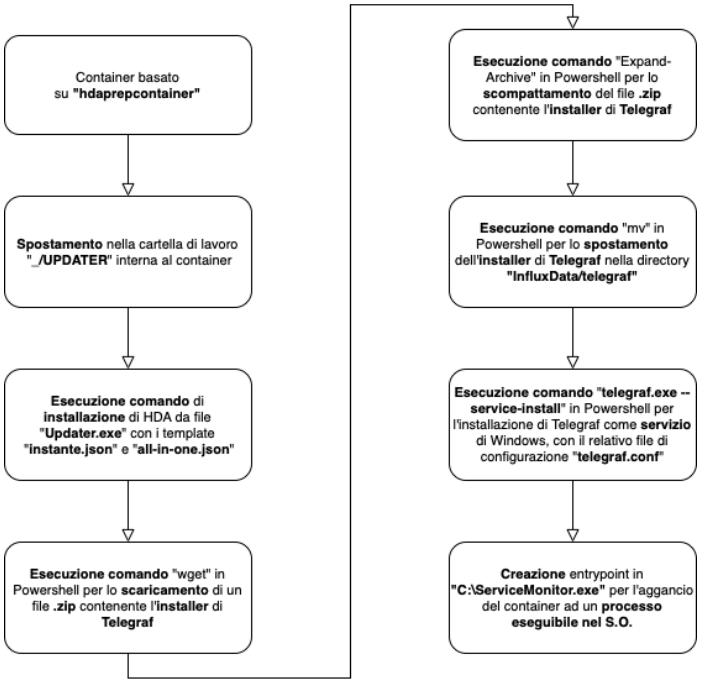


Figura 4.4: Flow-chart rappresentante la costruzione dell'immagine relativa al container "hdabasecontainer"

Questo container rimane in esecuzione dopo la fine dell'ultima istruzione agganciandosi al processo interno "ServiceMonitor.exe", e fa parte della *sandbox* applicativa.

4.3.4 Dockerfile container: "lokicontainer"

Il presente dockerfile costruisce con successo l'immagine relativa al container "lokicontainer". La sequenza di passi per l'ottenimento dell'immagine è di seguito rappresentata:

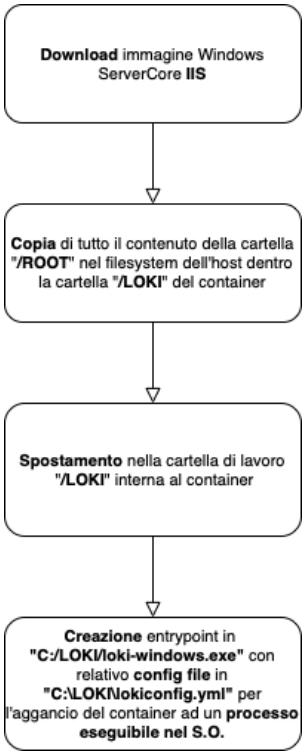


Figura 4.5: Flow-chart rappresentante la costruzione dell'immagine relativa al container "lokicontainer"

Questo container rimane in esecuzione dopo la fine dell'ultima istruzione agganciandosi al processo interno "loki-windows.exe", e fa parte della *sandbox* applicativa.

4.3.5 Dockerfile container: "promtailcontainer"

Il presente dockerfile costruisce con successo l'immagine relativa al container "promtailcontainer". La sequenza di passi per l'ottenimento dell'immagine è di seguito rappresentata:

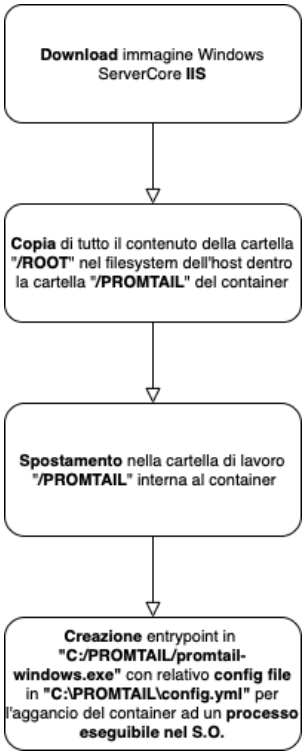


Figura 4.6: Flow-chart rappresentante la costruzione dell'immagine relativa al container "promtailcontainer"

Questo container rimane in esecuzione dopo la fine dell'ultima istruzione agganciandosi al processo interno "promtail-windows.exe", e fa parte della *sandbox* applicativa.

4.3.6 Dockerfile container: "nginxcontainer"

Il presente dockerfile costruisce con successo l'immagine relativa al container "nginxcontainer". La sequenza di passi per l'ottenimento dell'immagine è di seguito rappresentata:

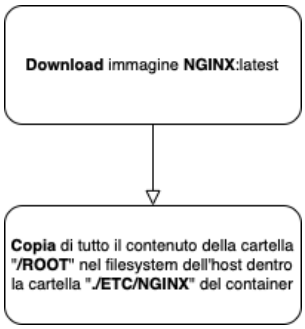


Figura 4.7: Flow-chart rappresentante la costruzione dell'immagine relativa al container "nginxcontainer"

Questo container rimane in esecuzione dopo la fine dell'ultima istruzione, e fa parte della *sandbox* applicativa.

4.4 Costruzione container tramite automazione: HDA Sandbox Builder

Testo iniziale:

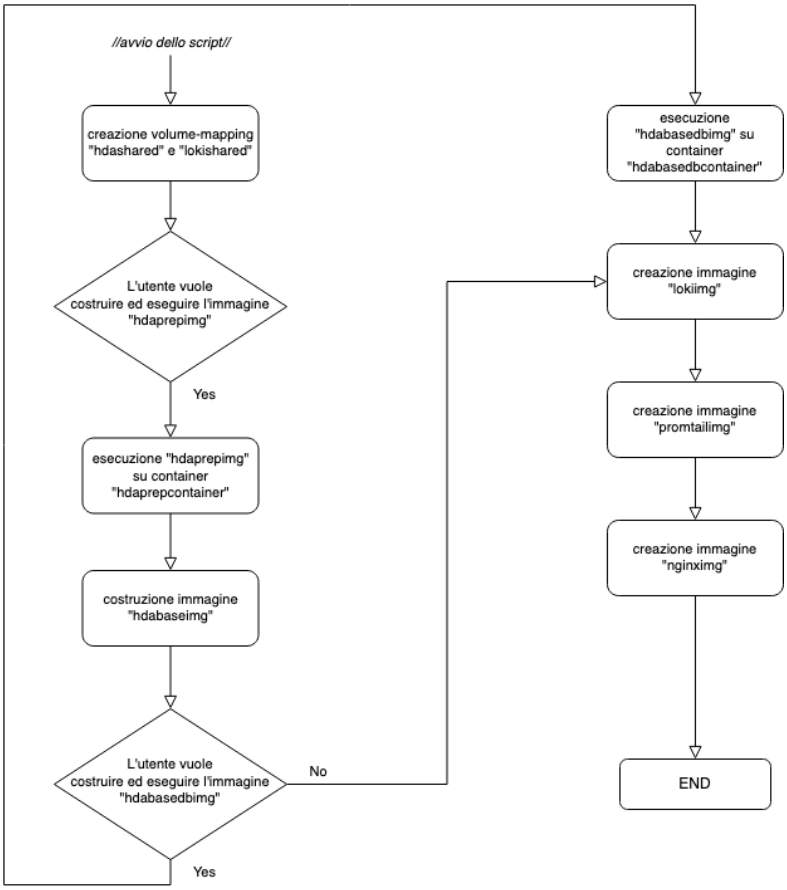


Figura 4.8: Flow chart del batch-file "hda_sandbox_builder.bat"

testo finale.

4.5 Installazione nel computer client

L'infrastruttura sopracitata, al fine di funzionare correttamente, necessita di una installazione di Docker all'interno del sistema operativo *host* dove verrà creata ed eseguita l'intera infrastruttura. I passi atti ad una corretta installazione di Docker sono reperibili nella pagina di "Download and Install" nella relativa documentazione di Docker. Un importante accorgimento che l'utente installatore dovrà obbligatoriamente eseguire è lo switching dai container Linux ai container Windows, effettuabile dall'icona di Docker posizionata nella barra Start di Windows, selezionando l'apposita voce "Switch to Windows Containers" come da immagine:

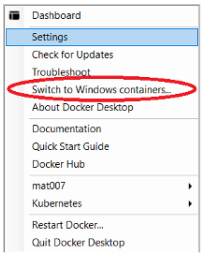


Figura 4.9: Voce da selezionare per cambiare la tipologia di container

Una volta selezionata la funzionalità sopra riportata, il Docker Engine si riavvierà, permettendo ora la creazione ed esecuzione dei container basati sul sistema operativo Microsoft Windows.
Installato con successo l'applicativo Docker, è necessario accedere al file di configurazione dello stesso ("docker.conf") o, tramite Docker Desktop, accedere alle impostazioni dello stesso, per abilitare la funzionalità "experimental features" che permette appunto l'esecuzione su Windows di container basati su Linux (necessario per il container "nginxcontainer"), ed esporre le relative Docker API su una determinata porta. L'esposizione delle Docker API sono di vitale importanza per l'esecuzione dello script Powershell "nginxREscript.ps1", e la porta precedentemente esposta andrà modificata manualmente anche all'interno dello stesso script. Il risultato finale sarà simile alla seguente immagine, dove si illustra graficamente, tramite Docker Desktop, il file di configurazione di Docker esponendo le Docker API nella porta "2390":

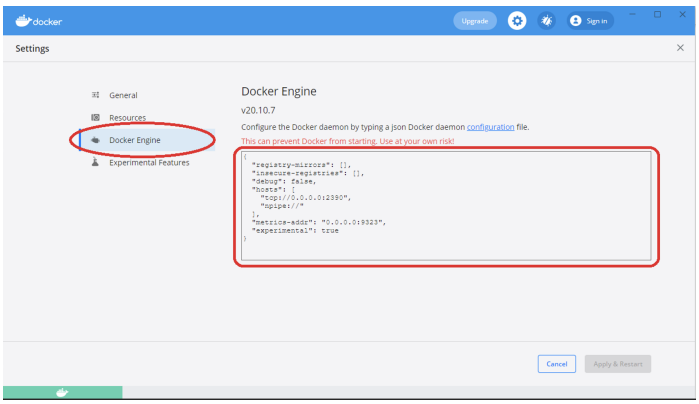


Figura 4.10: File di configurazione di Docker Desktop con le API esposte in porta "2390"

Ad ogni riavvio della macchina fisica è possibile che Docker Engine vada in errore, in quanto non riuscirà ad esporre le proprie API nella porta precedentemente configurata. Per risolvere questo problema, l'installatore dovrà configurare una nuova porta, non precedentemente utilizzata, esattamente come visto sopra.

Capitolo 5

Orchestrazione container

Breve introduzione al capitolo

5.1 Breve panoramica su Docker Compose

Di seguito viene data una panoramica delle tecnologie e strumenti utilizzati.

Tecnologia 1

Descrizione Tecnologia 1.

Tecnologia 2

Descrizione Tecnologia 2

5.2 Docker compose per la costruzione di una sandbox applicativa

5.3 Progettazione

Namespace 1

Descrizione namespace 1.

Classe 1: Descrizione classe 1

Classe 2: Descrizione classe 2

5.4 Integrazione di HDA Sandbox Builder con Docker Compose

5.5 Codifica

Capitolo 6

Reverse Proxy tramite NGINX sulle sandbox di HDA

6.1 Introduzione ad NGINX

6.2 Analisi e struttura file di configurazionr "nginx.conf"

6.3 Analisi NginxREScript.ps1

Capitolo 7

Conclusioni

7.1 Raggiungimento degli obiettivi

7.2 Conoscenze acquisite

7.3 Valutazione personale

Appendice A

Appendice A

Citazione

Autore della citazione

Bibliografia