Technology Know-How: NGINX
Report finale

Elena Morelli
elena.morelli3@studio.unibo.it

Maggio 2023

Indice

Capitolo 1

Introduzione

Nginx, è un web server leggero ad alte prestazioni, disponibile per vari sistemi Unix, Unix-like e Windows. Rilasciato nel 2004 come software open source, Nginx è stato concepito con lo scopo esplicito di essere più performante del web server Apache.

La sua architettura ad eventi esibisce prestazioni consistenti in presenza di carichi elevati, con maggiore efficienza rispetto al modello multi-threaded adottato dal suo concorrente.

Sviluppato con l'obbiettivo di risolvere il problema C10k, ovvero gestire più di diecimila client contemporaneamente senza osservare alcun tipo di degradazione a livello prestazionale, Nginx può arrivare a sostenere il 400% di richieste in più al secondo rispetto ad Apache.

Nginx non è soltanto un server web, può agire da reverse proxy, da proxy TCP/UDP, da e-mail proxy e da load balancer. Queste caratteristiche ne hanno favorito una larga diffusione, soprattutto nelle mansioni di reverse proxy o load balancer.

1.1 Obbiettivi/Requisiti

Il progetto nasce con l'obbiettivo di utilizzare Nginx come un reverse proxy, con la capacità di inoltrare le richieste attraverso diverse tipologie di protocolli. In particolare, verrà considerata la necessità di configurare il proxy Nginx per convertire la richiesta adattandola ai vincoli dei servizi invocati.

Per la realizzazione dell'elaborato, nello specifico, si procederà come segue:

- studio di Nginx incentrato sull'architettura, i concetti chiave e le possibili configurazioni
- realizzazione della parte applicativa: il proxy, a fronte della ricezione di una chiamata, inoltra quest'ultima al servizio di destinazione criptando il traffico dove necessario, utilizzando i certificati richiesti dal servizio stesso.

Capitolo 2

NGINX

2.1 Architettura

I modelli process-based o thread-based, che gestiscono connessioni simultanee, utilizzano un processo o thread separato per operazioni di rete bloccanti o di tipo input/output. Quest'ultimo, a seconda dell'applicazione, può essere molto inefficiente in termini di memoria e consumo della CPU. Infatti, la generazione di un processo o thread separato, richiede la preparazione di un nuovo ambiente di runtime, inclusa l'allocazione di memoria heap e stack e la creazione di un nuovo contesto di esecuzione. Viene inoltre spesa ulteriore CPU time per creare questi elementi che possono portare a prestazioni scadenti a causa del thread thrashing dato da un eccessivo context switching. Tutte queste complicazioni si manifestano nelle vecchie architetture web server, ad esempio, quella di Apache, come un compromesso tra l'offerta di un ricco set di funzionalità e l'utilizzo ottimizzato delle risorse del server.

Fin dall'inizio, Nginx è stato pensato per essere uno strumento specializzato, in grado di ottenere prestazioni migliori rispetto all'utilizzo delle risorse presenti nel server. Ispirato dal continuo sviluppo di meccanismi event-based avanzati su svariati sistemi operativi, Nginx presenta un'architettura modulare, event-based, asincrona, single-threaded, non bloccante di tipo master-slave.

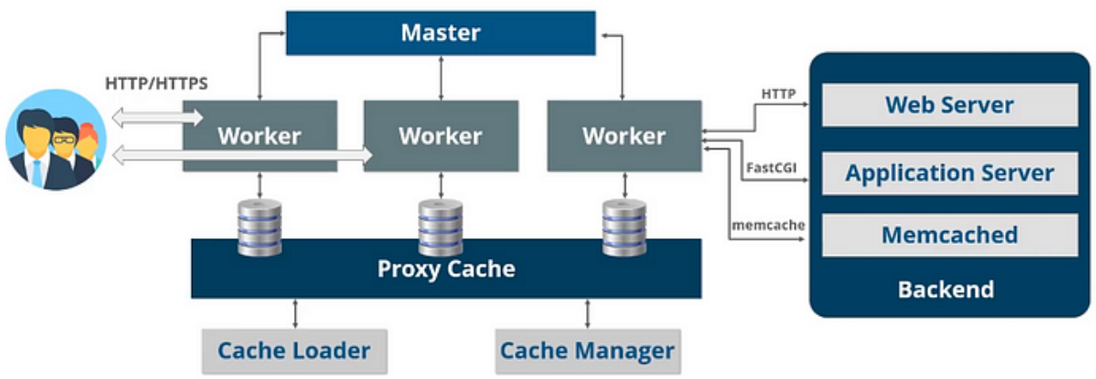


Figura 2.1: Architettura Nginx

Come mostra la figura 2.1, il web server presenta tre componenti:

- Master
- Worker
- Cache

2.1.1 Master

Il processo master è il processo principale che si occupa di creare e orchestrare tutti gli altri, essendo il primo ad entrare in funzione all'esecuzione di Nginx. E' responsabile della lettura e validazione della configurazione, creazione del Worker Process, *Cache Loader Process* e *Cache Manager Process*,

binding e chiusura dei socket, gestione dei processi worker, riconfigurazione senza interruzioni del servizio.

Il master assegna compiti ai worker in base alla richieste del client. Una volta che il lavoro è stato conferito, si rimetterà in attesa di ulteriori richieste, senza aspettare che i worker abbiano terminato l'elaborazione. Solo una volta ricevuta la risposta da quest'ultimi, il master invierà la risposta al client.

Figura 2.2: Processo master

2.1.2 Worker

I processi worker accettano, gestiscono ed elaborano le connessioni provenienti dai client, forniscono funzionalità di reverse proxy e filtraggio, portando a termine quasi tutto ciò di cui Nginx è capace. Come accennato in precedenza, Nginx non genera un processo o un thread per ogni connessione. Al contrario, i processi worker accettano nuove richieste da socket condivise ed eseguono internamente un run-loop altamente efficiente per elaborare migliaia di connessioni contemporaneamente.

Il principio di run-loop è di essere meno bloccante possibile. Per raggiungere questo obiettivo si affida principalmente all'idea di task asincroni, implementati tramite la modularità, notifiche di eventi, callback functions e timer; tuttavia è possibile che alcune operazioni siano comunque bloccanti, ad esempio nel caso in cui non ci sia più spazio a disposizione sul disco.

Questo approccio permette un utilizzo di memoria conservativo ed efficiente. Non solo, anche sul fronte della CPU si risparmiano cicli, data l'assenza di una continua creazione e distruzione di processi o thread.

Per ottimizzare e massimizzare l'utilizzo di architettura multicore, generalmente viene creato un processo worker per ogni core della CPU, soprattutto nel caso in cui ci sia un utilizzo intenso del processore (come, ad esempio, gestire molte connessioni TCP). Non è però vero in altre circostanze, dove si potrebbe preferire un numero 1,5 / 2 volte maggiore di worker per core, qualora le operazioni più intensive fossero di I/O, dove potrebbe essere necessario servire differenti storage.

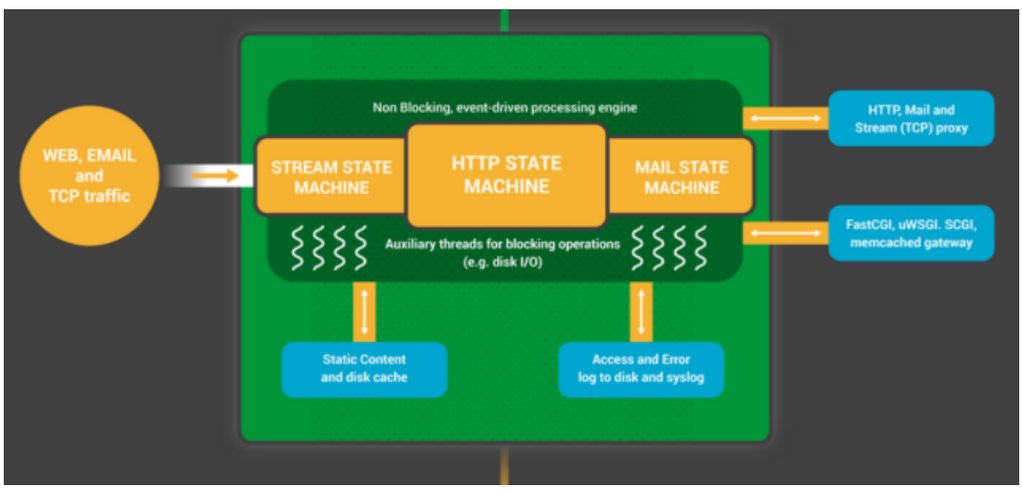


Figura 2.3: Funzionamento processo worker

I processi worker, una volta creati, attendono eventi sulle socket di ascolto (accept_mutex e kernel socket sharding). Questi ultimi vengono avviati da nuove connessioni in entrata. Le quali vengono sono assegnate ad una macchina a stati: la HTTP è la più utilizzata. Nginx implementa anche macchine a stati per il traffico in streaming (TCP non elaborato) e per una serie di protocolli di posta (SMTP, IMAP e POP3).

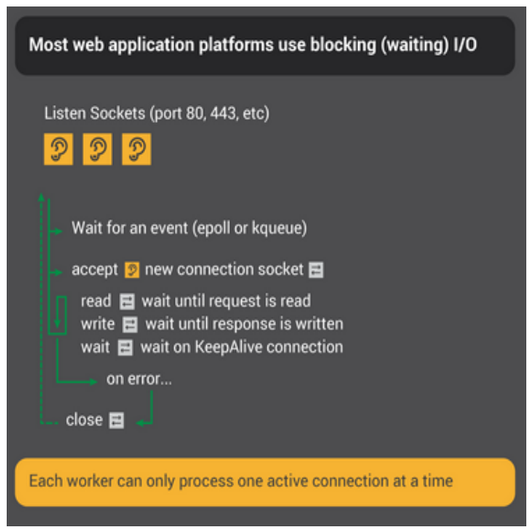


Figura 2.4: Funzionamento macchina a stati nei server classici

La macchina a stati è essenzialmente l'insieme di istruzioni che indicano a Nginx come elaborare una richiesta. La maggior parte dei web server che svolgono le stesse funzioni di Nginx utilizzano una macchina a stati simile: la differenza è solo nell'implementazione.

- Il processo del web server ascolta le nuove connessioni sulle socket di ascolto.
- Quando ottiene una nuova richiesta, la esegue, bloccandosi dopo ogni esecuzione per attendere la risposta del client.
- Una volta completata la connessione, il processo del web sever potrebbe attendere per verificare che il client desideri mantenerla attiva (questo corrisponde a una connessione keepalive) o meno. Se la connessione viene chiusa (il client scompare o si verifica un timeout), il processo del web server torna in fase di ascolto, in attesa di nuove connessioni.

Il punto importante da ricordare è che ogni connessione HTTP attiva, richiede un processo o thread dedicato. Questa architettura è semplice e facile da estendere tramite l'utilizzo di moduli di terze parti. Tuttavia, c'è un enorme squilibrio: una connessione HTTP leggera, rappresentata da un file descriptor e una piccola quantità di memoria, viene mappata ad un thread o processo separato che è un oggetto del sistema operativo molto pesante. Può essere una comodità di programmazione, ma è estremamente dispendiosa.

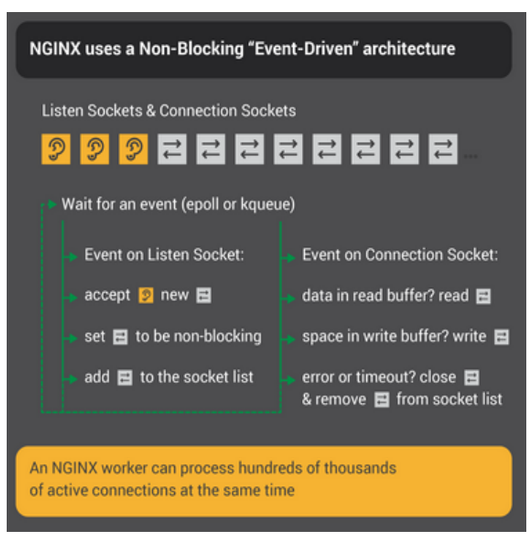


Figura 2.5: Funzionamento macchina a stati in Nginx

Come mostra la figura 2.5, la soluzione proposta da Nginx permette di mantenere al minimo l'utilizzo delle risorse senza dover creare un nuovo thread per ogni richiesta, garantendo una scalabilità non lineare sia sul numero di connessioni simultanee che per richieste al secondo.

- Il worker resta in attesa di un evento sui listen e connection socket;
- Viene scatenato un evento e il worker lo gestisce in base al socket di provenienza:
  - Un evento sul listen socket corrisponde ad una nuova richiesta da parte del client, per il quale il worker dovrà creare una nuova connection socket, aggiungendola al run-loop;
  - Un evento sul connection socket corrisponde ad una nuova risposta da parte del client, al quale il Worker può rispondere.
- Quando la richiesta viene soddisfatta, la connessione viene deallocata e rimossa dal run-loop;

In questo caso il nostro single-thread worker non resta mai fermo in attesa di una risposta, potendosi così occupare di più richieste.

Nginx quindi si adatta molto bene a supportare centinaia di migliaia di connessioni per worker process. Ogni nuova connessione crea un altro descrittore di file e consuma una piccola quantità di memoria aggiuntiva in esso. L'overhead aggiuntivo per connessione è minimo. I processi Nginx possono rimanere bloccati nelle CPU. I cambi di contesto sono relativamente rari e si verificano quando non c'è lavoro da fare.

Nell'approccio di blocco, connessione per processo, ogni connessione richiede una grande quantità di risorse aggiuntive e sovraccarico, i cambi di contesto (passaggio da un processo all'altro) sono molto frequenti.

2.1.3 Cache

La cache Nginx viene utilizzata per velocizzare il render della pagina, in modo da ottenerla dalla memoria invece che dal server. Le pagine infatti, vengono memorizzate nella memoria cache alla prima richiesta.

Ci sono due componenti principali della cache di Nginx:

- il Cache Loader: è il processo che gestisce il caricamento in memoria della cache disk-based, popolando il database in memoria con metadati e la preparazione dell'istanza di Nginx per lavorare con file storicizzati sul disco in una determinata struttura di directory. Caricare l'intera cache in una volta sola potrebbe consumare risorse sufficienti per rallentare le prestazioni di Nginx durante i primi minuti dopo l'avvio. Per evitare ciò, bisogna configurare il caricamento iterativo della cache includendo i seguenti parametri nella direttiva proxy_cache_path:
  - loader_threshold: durata di un'iterazione, in millisecondi
  - loader_files: numero massimo di elementi caricati durante un'iterazione
  - loader_sleeps: ritardo tra le iterazioni, in millisecondi

  Quando ha completato il proprio compito, termina la propria esecuzione mantenendo al minimo le quantità di risorse necessarie.
- il Cache Manager: viene attivato periodicamente per verificare lo stato della cache. Se la dimensione di questa supera il limite impostato dal parametro max_size nella direttiva proxy_cache_path, il gestore di quest'ultima rimuoverà i dati alla quale è stato effettuato l'accesso meno recentemente.

Caching

Il sistema di caching è implementato nella forma di uno storage gerarchico, configurabile, sul filesystem. Le chiavi della cache sono configurabili, determinati parametri nelle richieste possono essere usati per controllare ciò che deve essere inserito nella cache. Sia le chiavi che i metadati sono storicizzati nella memoria condivisa, accessibile da tutti i processi di Nginx.

Ogni elemento della cache è posto in un file differente sul filesystem, il cui nome e percorso vengono derivati dall'hash MD5 dell'url del proxy.

Non è presente nessuna metodologia di ottimizzazione dei file oltre a quella operata dal sistema operativo.

2.2 Struttura

Le impostazioni di base di Nginx sono configurate principalmente all'interno del file nginx.conf. Queste tipologie di file di configurazione sono strutturati tramite contesti.

I principali sono:

- Event Context
- HTTP Context

Ogni contesto può avere uno o più di questi nidificati al suo interno, che posso ereditare le impostazioni dai loro genitori, ma possono anche sovrascriverle secondo le necessità.

Oltre a questi sopra citati sono presenti alcuni parametri configurabili:

- worker_process
- worker_connection
- access.log, erro.log e gzip

2.2.1 Event Context

L'event context è contenuto all'interno del contesto principale, il main context. Viene utilizzato per impostare opzioni globali che influenzano il modo in cui Nginx gestisce le connessioni a livello generale. Ci può essere un solo event context definito all'interno del file di configurazione.

Viene rappresentato come segue:

```
#main context
events {
    #events context
    . . .
}
```

Nginx utilizza un modello di elaborazione delle connessioni basato su eventi, quindi le direttive definite in questo contesto determinano il modo in cui i processi di lavoro devono gestire le connessioni. Essenzialmente, le direttive vengono impiegate per selezionare la tecnica di elaborazione delle connessioni da utilizzare o per modificare il modo in cui questi metodi vengono implementati.

2.2.2 HTTP Context

Quando si configura Nginx come web server o reverse proxy, l' HTTP context conterrà la maggior parte della configurazione. Vi saranno tutte le direttive e i contesti necessari per definire il modo in cui il programma gestirà le connessioni HTTP o HTTPS. L'HTTP context è un elemento di pari livello all'event context ed entrambi sono figli del main context:

```
#main context
events {
    #events context
    . . .
}
http {
    #http context
    . . .
}
```

Server Context

Il server context viene dichiarato all'interno dell'HTTP context. E' un contesto nidificato, che consente più dichiarazioni. Il formato generico per il server context può essere simile a:

```
 #main context
http {
    #http context
    server {
        #first server context
    }
    server {
        #second server context
    }
}
```

La motivazione per cui è possibile dichiarare più server context è che ogni istanza definisce un server virtuale specifico per gestire le richieste client. È possibile disporre di blocchi in base ai server necessari, ognuno dei quali può gestire un sottoinsieme specifico di connessioni.

Location Context

I location context condividono molte qualità relazionali con i server context, in quanto è possibile definire più location context all'interno del server context. Ogni location viene utilizzata per gestire un determinato tipo di richiesta selezionata in virtù della corrispondenza tra la definizione della location, con la richiesta client, tramite un algoritmo di selezione. La sintassi generale è simile alla seguente:

```
location match_modifier location_match {
    . . .
}

```

I blocchi location risiedono all'interno di blocchi server (o ad altri blocchi location) e vengono utilizzati per decidere come elaborare l'URI, la parte della richiesta che viene dopo il nome di dominio o l'indirizzo IP/porta). Oltre ai contesti sopra citatati, ve ne sono altri opzionali come: upstream context, split_clients context, map context, geo context, types context, charset_map ecc.

2.2.3 Parametri Configurabili

Worker_process

I worker_processes sono l'impostazione che settano il numero di Nginx worker process (l'impostazione predefinita è 1).

Nella maggior parte dei casi, l'esecuzione di un processo di lavoro per core CPU funziona bene, l'impostazione consigliata è "auto".

Worker_connections

E' il numero massimo di connessioni che ogni worker_processes può gestire contemporaneamente. L' impostazione predefinita è 512, molti sistemi hanno risorse sufficienti per supportarne un numero più elevato (cioè 1024, 2048 per processi di lavoro o anche di più)

Access.log, error.log & gzip

Access.log ed error.log vengono utilizzate per registrare qualsiasi errore in Nginx oltre al debug. Gzip indica le impostazioni per la compressione gzip sulla risposta Nginx.

2.3 Configurazioni

2.3.1 Load Balancing

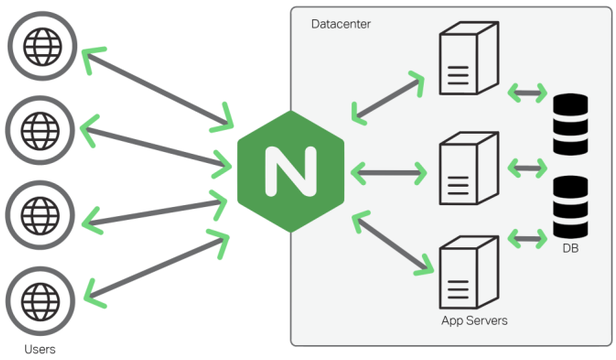


Figura 2.6: Nginx come load balancer

Il bilanciamento del carico di un'applicazione su più istanze è una tecnica comunemente utilizzata per ottimizzare l'impiego delle risorse, massimizzare il throughput, ridurre la latenza di risposta e garantire una certa resilienza in caso di guasti. Il principio di funzionamento consiste nel far pervenire tutte le richieste ad un unico server, il quale provvede a distribuirle ai vari application server che compongono l'infrastruttura, agendo di fatto da reverse proxy.

Le modalità con le quali il server individua l'application server al quale redirigere le richieste, dipendono dalla configurazione.

Nginx può essere configurato per svolgere il ruolo di load balancer in modo molto efficiente, in particolare, supporta tre differenti modalità di load balancing:

- Round Robin;
- Least-Connected;
- IP-Hash.

Round Robin

Questo è il tipo predefinito di load balancer per Nginx. Utilizza il tipico algoritmo round-robin per decidere come distribuire la richiesta in arrivo.

```
http {
    upstream backend {
        server backend1.example.com;
        server backend2.example.com;
    }

    server {
        listen 80 default_server;
        server_name _;

        location / {
            proxy_pass http://backend;
        }
    }
}
```

Come mostra l'esempio nella figura 2.6, per prima cosa viene specificato un upstream, utilizzato per indicare gli indirizzi degli application server impiegati dal reverse proxy. Bisognerà poi specificare il nome dell'upstream come proxy_pass all'interno della sezione location, la quale si troverà a sua volta all'interno della sezione server. Questo farà sì che si bilanci il carico tra i due server dichiarati precedentemente.

Nginx può agire da distributore di carico (e quindi reverse proxy) per una serie di protocolli oltre ad HTTP. In particolare, Nginx supporta HTTPS, FastCGI, uwsgi, SCGI, memcached e gRPC.

Per utilizzarlo come reverse proxy HTTPS, ad esempio, è sufficiente sostituire "http" con "https" nella direttiva proxy_pass.

Least-Connected

La modalità least-connected permette di distribuire il carico in maniera più uniforme rispetto alla modalità round-robin, specialmente in quei casi in cui le richieste necessitino di più tempo per essere evase. Specificando questa modalità, Nginx cercherà di non sovraccaricare un server già impegnato, distribuendo il carico su quelli con meno connessioni attive in quel particolare momento. Per fare ciò, aggiungiamo la parola chiave *least_conn* in alto, all'interno della dichiarazione dell'upstream.

```
http {
    upstream backend {
        least_conn;
        server backend1.example.com;
        server backend2.example.com;
    }

    server {
        listen 80 default_server;
        server_name _;

        location / {
            proxy_pass http://backend;
        }
    }
}
```

Se risultano esserci diversi server elencati come least-connected che presentano un numero altrettanto basso di connessioni attive esistenti, ne verrà scelto solamente uno in base all'algoritmo round-robin ponderato.

IP-Hash

Sia nella modalità round-robin che in quella least-connected non vi è garanzia della persistenza delle sessioni. HTTP infatti è per sua natura stateless, pertanto la gestione delle sessioni è sempre a carico del server. In uno scenario in cui è presente un load balancer, è possibile che le richieste in arrivo da uno stesso client, che logicamente appartengono ad una stessa sessione, siano distribuite su application server differenti e pertanto non saranno in grado di riconoscere gli ID o i cookie di sessione generati dagli altri.

Per ovviare a questo inconveniente, bisogna configurare il load balancer in modo tale da assicurarsi che le richieste in arrivo da uno stesso client vengano sempre servite dallo stesso application server. Tale schema è implementato in Nginx con la modalità ip-hash.

In quest'ultimo, l'indirizzo IP del client è utilizzato come chiave di hashing per la determinazione dell'application server da selezionare.

```
http {
    upstream backend {
        ip_hash;
        server backend1.example.com;
        server backend2.example.com;
    }

    server {
        listen 80 default_server;
        server_name _;

        location / {
            proxy_pass http://backend;
        }
    }
}
```

In questo caso, i primi tre ottetti dell'indirizzo IPv4 o l'intero indirizzo IPv6 vengono utilizzati per calcolare il valore hash. Il metodo garantisce che le richieste provenienti dallo stesso indirizzo arrivino allo stesso server, tranne il caso in cui non sia disponibile. Se uno dei server deve essere

temporaneamente rimosso dalla rotazione del bilanciamento del carico, può essere contrassegnato con il parametro down, per mantenere l'hashing corrente degli indirizzi IP client. Le richieste che dovevano essere elaborate da questo verranno inviate automaticamente a quello successivo nel gruppo.

```
upstream backend {
    ip_hash;
    server backend1.example.com;
    server backend2.example.com;
    server backend3.example.com down;
}
```

Specifica dei pesi

Nginx permette di attribuire un peso numerico a ciascun application server indicato nella direttiva upstream. I server, con un peso maggiore verranno selezionati più frequentemente di quelli con un peso inferiore. Per default, in assenza di una esplicita indicazione, si assume un peso pari ad 1.

Specificare dei pesi permette di distribuire il carico in maniera più consona rispetto alle reali capacità di elaborazione degli application server. Ciò è particolarmente utile nel caso in cui essi girino su macchine con caratteristiche differenti.

Si supponga che backend3 sia in grado di gestire più richieste rispetto ai server backend2 ed backend1, in tal caso si può assegnare al backend3 un peso più alto per assicurarsi che esso venga selezionato più spesso dal load balancer.

```
http {
    upstream backend {
        server backend1.example.com;
        server backend2.example.com;
        server backend3.example.com weight=3;
    }

    server {
        listen 80 default_server;
        server_name _;

        location / {
            proxy_pass http://backend;
        }
    }
}
```

In questo modo, ricevute cinque richieste, queste verranno distribuite come segue: tre a backend3, una a backend2 ed un'altra a backend1.

Resilienza

Nginx include anche un meccanismo di monitoraggio passivo degli application server. Ciò significa che se un application server non risponde correttamente alle richieste, esso verrà marcato come non funzionante ed Nginx non lo selezionerà per le richieste successive.

È possibile specificare il numero di tentativi da effettuare prima di considerare l'application server non funzionante tramite la direttiva max_fails. Per default, questo parametro è impostato ad uno, ciò comporta che un server venga immediatamente riconosciuto come non funzionante al verificarsi di un errore.

Nginx è configurabile affinché monitori lo stato dei server non funzionanti, tramite controlli periodici. Il tempo che intercorre tra un controllo e l'altro viene impostato con la direttiva fail_timeout. Per default, l'intervallo di fail_timeout à impostato a 10 secondi.

2.3.2 Load Balancer TCP e UDP

Nginx può operare come load balancer ad un livello inferiore rispetto a quello applicativo. Questa modalità viene spesso definita Level 4 load balancing, riferendosi allo stack ISO/OSI. In questa circostanza, il server non opera più su richieste, ma su generici flussi di dati detti stream.

In altre parole, a differenza del load balancing di livello applicativo, in quello di livello 4 il server è ignaro del contenuto dei messaggi scambiati tra client e server e del loro significato, limitandosi semplicemente a reindirizzare i flussi opportunamente.

Nginx può operare sia su connessioni TCP che su datagrammi UDP, reindirizzandoli opportunamente ai server.

La configurazione avviene mediante la specifica dei server applicativi, con l'aggiunta della direttiva upstream e con la configurazione del server in ingresso. Tuttavia, a differenza del load balancing HTTP, queste direttive andranno inserite nel blocco stream invece che nel blocco http.

```
stream {
    upstream backend {
        server backend1.example.com:53;
        server backend2.example.com:53;
        server backend3.example.com:53;
    }

    server {
        listen 53 udp;
        proxy_pass backend;
    }
}
```

In impostazione predefinita, il server si metterà in ascolto, sulla porta 53 UDP, su tutte le interfacce di rete presenti nella macchina. Nel caso in cui si volesse restringere il binding ad un'interfaccia specifica, è possibile utilizzare la direttiva *proxy_bind*, specificando l'indirizzo IP da usare.

```
stream {
    # ...
    server {
        listen 127.0.0.1:53 udp;
        proxy_pass my_servers;
        proxy_bind 127.0.0.1:53;
    }
}
```

E' possibile inoltre controllare la dimensione del buffer utilizzato da Nginx per mantenere in memoria i dati in transito, da e verso i client, utilizzando la direttiva *proxy_buffer_size*.

```
stream {
    # ...
    server {
        listen 8080;
        proxy_pass my_servers;
        proxy_buffer_size 16k;
    }
}
```

Diminuire la dimensione del buffer può essere utile per ridurre la latenza introdotta dal load balancer, ciò però comporta un aumento del numero di operazioni di I/O effettuate dal server, con un conseguente aumento del carico.

Una dimensione ottimale va ricercata in relazione al particolare caso d'uso, tenendo conto della grandezza media dei pacchetti in transito e delle esigenze in termini di latenza.

In modalità TCP/UDP, Nginx supporta diversi metodi per la selezione del server di upstream al quale reindirizzare i flussi in ingresso.

- Round-Robin: la modalità di default. I server vengono selezionati secondo una sequenza ciclica, in base all'ordine delle connessioni dei client;
- Least-Conn: viene selezionato il server con meno connessioni attive in quel momento;
- Hash: una funzione di hashing è utilizzata come funzione di lookup per assegnare univocamente un server ad un determinato client. La variabile da utilizzare come chiave di hashing viene impostata nella configurazione;
- Random: seleziona il server di upstream in modo casuale;

Il metodo round-robin è attivo per default e pertanto non è associato ad alcuna direttiva. Ogni blocco upstream funziona in questa modalità , tranne nel caso in cui non sia esplicitamente indicata.

La modalità hash viene utilizzata per implementare la persistenza delle sessioni. L'obiettivo è quello di evitare che i dati provenienti da un client possano essere gestiti da più server. Il concetto di sessione, così come le modalità di gestione della stessa, dipendono essenzialmente dal protocollo applicativo in uso, senza lasciare ad Nginx nessuna supposizione in merito. Viene quindi data la facoltà di definire un criterio opportuno, specificando una chiave per la funzione di hashing da utilizzare.

```
stream {
    upstream backend {
        hash $remote_addr;
        server backend1.example.com:123;
        server backend2.example.com:123;
        server backend3.example.com:123;
    }
}
```

Come nel caso del load balancing HTTP, è inoltre possibile assegnare dei pesi specifici ai singoli server in upstream, mediante il parametro weight. Si può specificare un limite al numero massimo di connessioni che un server di upstream può gestire, con il parametro *max_conns*. Ad esempio:

```
stream {
    upstream backend {
        hash $remote_addr consistent;
        server backend1.example.com:12345  weight=5;
        server backend2.example.com:12345;
        server backend3.example.com:12346  max_conns=3;
    }

    upstream dns_servers {
        least_conn;
        server 192.168.136.130:53;
        server 192.168.136.131:53;
        # ...
    }
}
```

Con questa configurazione verrà scelto il server con meno connessioni attive. Tuttavia, se il server scelto è il backend3, questo verrà selezionato solo se ha meno di tre connessioni in corso. Nel caso in cui vi siano più sever che rispondano al criterio di selezione (ad esempio, due server con nessuna connessione in corso), questi verranno selezionati tenendo conto del peso specificato nel parametro weight.

2.3.3 Reverse Proxy

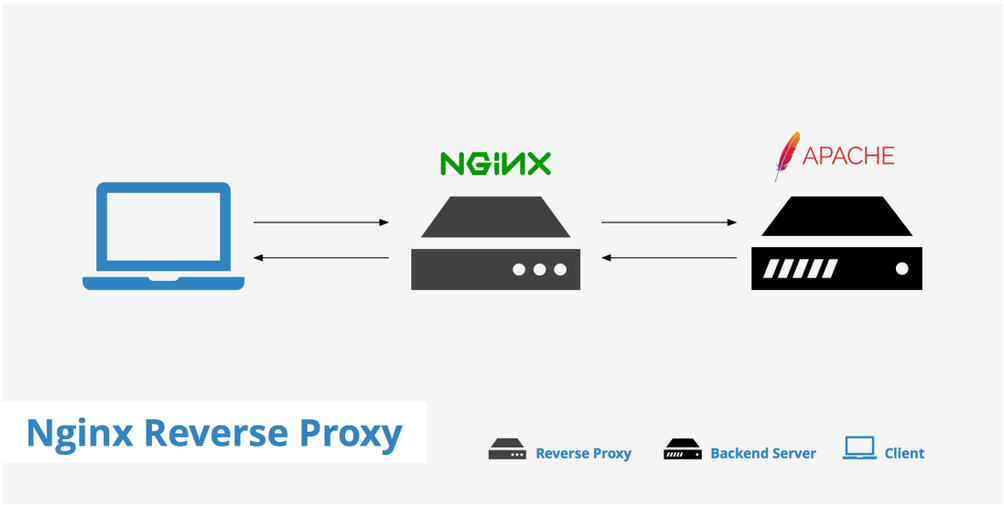


Figura 2.7: Nginx come reverse proxy

Uno degli utilizzi più comuni del server Nginx è quello di reverse proxy, quest'ultimo è un server web che si frappone tra i client HTTP(s) ed il/i server utilizzato/i. I client interrogano il server proxy inviando le proprie richieste HTTP, il quale a sua volta interroga il server interno che contiene la risorsa richiesta, elabora la risposta e la ritrasmette al client.

Si può ricorrere ad un reverse proxy qualora le risorse di un'applicazione o di un sito web siano servite da più server ma si voglia accedere alle stesse da un dominio comune; utilizzando un reverse proxy il client effettuerà le richieste ad un unico server, senza sapere che queste sono in realtà reindirizzate altrove. Permette di aggiungere delle funzionalità ad applicazioni esistenti, ad esempio, si può ricorrere ad un reverse proxy per dotare di supporto a TLS un'applicazione che ne sia sprovvista.

La configurazione di Nginx come reverse proxy avviene semplicemente specificando una *Location* con la direttiva proxy_pass.

```
location /user/location/ {
   proxy_pass http://www.example.com/link/;
}
```

Quando i client richiederanno al server l'indirizzo /user/location, Nginx effettuerà una richiesta al server www.example.com, richiedendo la risorsa /link e restituendo il risultato della richiesta al client.

Si noti che l'intera location viene mappata sul server esterno; il funzionamento del reverse proxy si basa sul matching di URI. In altre parole, bisogna specificare quali URI sono serviti direttamente dal server Nginx e quali invece sono mappati su risorse esterne.

La direttiva proxy_pass accetta un URL valido, è pertanto possibile effettuare delle richieste su porte differenti da 80 o 443. Per farlo è sufficiente specificare il numero di porta nell'URL indicato nella direttiva proxy_pass. Supponendo che il server example.com sia in ascolto sulla porta 3000, la configurazione da utilizzare sarà:

```
location /user/location/ {
   proxy_pass http://www.example.com:3000/link/;
}
```

La direttiva proxy_set_header consente di personalizzare le intestazioni inviate al server interno, permettendo di ridefinire il valore di quelli inviati dal client. Nel caso volessimo ridefinire il valore dell'intestazione Accept-Encoding per prevenire una risposta compressa (tipicamente i client moderni supportano almeno la compressione gzip): per farlo è sufficiente inserire la direttiva proxy_set_header, come mostrato di seguito:

```
location /user/location/ {
   proxy_set_header Accept-Encoding "";
   proxy_pass http://www.example.com/link/;
}
```

Per impostazione predefinita, Nginx utilizza un buffer per elaborare le risposte ricevute dai server interni. Le risposte vengono quindi prima memorizzate in un buffer e poi successivamente inviate al client. In alcuni casi può rendersi necessaria un'attenta configurazione di questa opzione. Se da una parte il buffering aiuta a migliorare le performance con client lenti, dall'altra impiega risorse del server proxy per ogni connessione ed in generale aumenta la latenza.

La direttiva proxy_buffers permette di regolare il numero di buffer da utilizzare e la loro dimensione, mentre proxy_buffer_size indica la dimensione del buffer contenente solo la prima parte della risposta.

Proxy buffer si rende necessario per poter eventualmente elaborare le intestazioni prima di inviarle al client.

Con i parametri utilizzati nell'esempio sottostante, le richieste indirizzate a /user/location impiegheranno fino a 16 buffer ognuno da 4 KB ed un buffer di 2K per le intestazioni.

```
location /user/location/ {
   proxy_buffers 16 4k;
   proxy_buffer_size 2k;
   proxy_pass http://www.example.com/link/;
}
```

E' possibile, inoltre, disattivare completamente il buffering con la direttiva proxy_buffering impostata su off:

```
location /user/location/ {
   proxy_buffering off;
   proxy_pass http://www.example.com/link/;
```

```
}
```

2.3.4 Mail Proxy Server

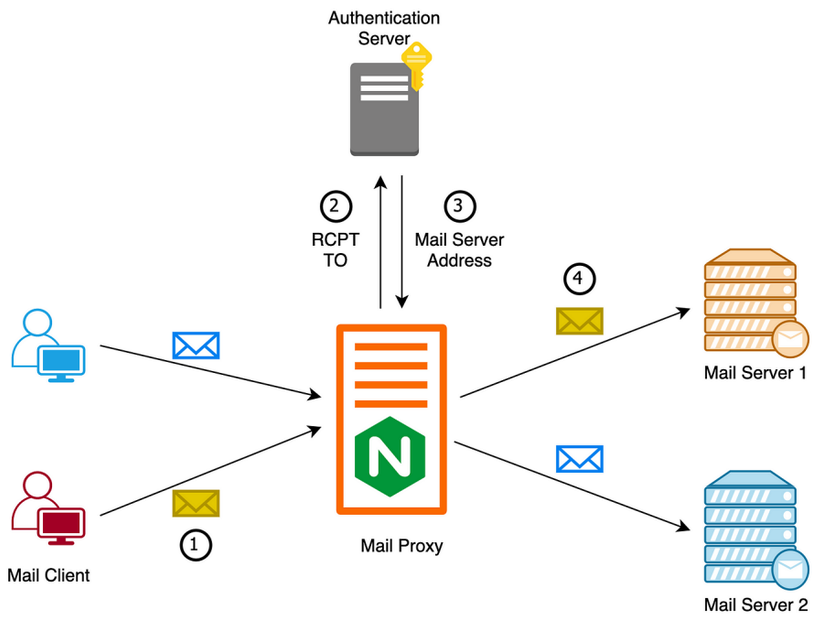


Figura 2.8: Nginx come mail proxy

Nginx può essere impiegato come proxy server per i protocolli IMAP, POP3 e SMTP, rendendolo di fatto un Mail Proxy Server. L'impiego di un unico endpoint pubblico per l'accesso ai servizi di posta permette di distribuire il carico su più server di backend e di implementare politiche avanzate di selezione del server (ad esempio, basate su geolocalizzazione IP).

I parametri di configurazione del mail proxy server sono inclusi nel contesto mail. La prima direttiva da inserire nel blocco mail serve ad indicare il nome FQDN del server e si chiama appunto server_name. Ad esempio:

```
mail {
    server_name mail.example.com;
    #...
}
```

Nginx permette di configurare le opzioni supportate dai protocolli in ingresso. Con la direttiva imap_capabilities è possibile specificare le estensioni IMAP che il server accetterà, con tale configurazione verranno abilitate le estensioni IMAP4rev1 e UIDPLUS:

```
mail {
    server_name mail.example.com;
    imap_capabilities "IMAP4rev1"  "UIDPLUS";
    #...
}
```

Analogamente, è possibile specificare le estensioni da attivare per il protocollo POP3 con la direttiva pop3_capabilities. Ad esempio, la configurazione seguente attiverà le estensioni TOP e USER:

```
mail {
    server_name mail.example.com;
    pop3_capabilities  "TOP"  "USER";
    #...
}
```

L'aspetto più importante nella configurazione di Nginx come mail proxy server è dato dalla particolare modalità di autenticazione implementata, infatti, la selezione del server di backend non è effettuata direttamente da Nginx ma viene affidata ad un servizio terzo.

Richiede, inoltre, l'implementazione di un servizio HTTP che si faccia carico di validare le credenziali degli utenti e di selezionare il server di backend adeguato.

Il processo di autenticazione si articola nelle fasi elencate di seguito:

1. il client si connette al server Nginx ed inizia la fase di autenticazione secondo il protocollo in uso (ad esempio, con SMTP invierà un messaggio AUTH);
2. il server Nginx contatta il servizio HTTP di autenticazione, inviando le credenziali ricevute dal client direttamente via HTTP (metodo di autenticazione HTTP Basic;
3. Il servizio HTTP verifica le credenziali secondo un'opportuna logica e nel caso in cui esse sian valide restituisce una risposta HTTP contenente tre header HTTP speciali:
   - Auth-Status, che indica che l'autenticazione è andata a buon fine;
   - Auth-Server, che riporta l'indirizzo IP del server di backend che Nginx dovrà utilizzare;
   - Auth-Port, che indica il numero di porta al quale contattare il server di backend;

   Nel caso in cui le credenziali non siano valide, il servizio HTTP risponderà senza i tre header indicati sopra;
4. Ottenuta una risposta dal servizio HTTP, Nginx provvederà a contattare il server di backend indicato. Nel caso di IMAP e POP3, verranno nuovamente inviate le credenziali al server di backend, dato che il processo di autenticazione è necessario per individuare univocamente le risorse alle quali accedere. Nel caso di SMTP, invece, Nginx non invierà alcun messaggio di autenticazione, anche se questo è stato precedentemente inoltrato dal client.

Il blocco mail deve essere configurato opportunamente, in modo da specificare il servizio HTTP di autenticazione da utilizzare. Ciò viene eseguito utilizzando la direttiva auth_http.

```
mail {
    server_name mail.example.com;
    auth_http localhost:9000/mail/auth.php;
    #...
}
```

Il servizio HTTP di autenticazione può essere implementato in qualunque linguaggio e contenere logiche di autenticazione piuttosto articolate. In casi più semplici, è sufficiente uno script PHP che può essere servito dalla stessa istanza di Nginx.

Nel blocco mail verranno indicati i protocolli da utilizzare, mediante il blocco server. Quest'ultimo indica il numero di porta TCP da usare, il protocollo ed i metodi di autenticazione. Ad esempio, nel caso in cui si vogliano attivare tutti i protocolli supportati (IMAP, POP3, SMTP), bisognerà specificare tre blocchi server.

```
mail {
    server_name mail.example.com;
    #...
    server {
        listen    25;
        protocol  smtp;
        smtp_auth login plain cram-md5;
    }
    server {
        listen    110;
        protocol  pop3;
        pop3_auth plain apop cram-md5;
    }
    server {
        listen   143;
        protocol imap;
    }
}
```

I protocolli IMAP, POP3 ed SMTP possono essere affiancati da SSL/TLS. Per attivare queste opzioni bisogna specificare nel blocco mail il percorso del certificato e della relativa chiave privata, elencare i protocolli supportati ed i relativi cifrari e abilitare il modulo SSL.

```
mail {
    server_name mail.example.com;
```

```
    #...
    ssl                  on;
    ssl_certificate      /etc/ssl/certs/server.crt;
    ssl_certificate_key  /etc/ssl/certs/server.key;
    ssl_protocols        TLSv1 TLSv1.1 TLSv1.2;
    ssl_ciphers          HIGH:!aNULL:!MD5;
}
```

Nel caso in cui si voglia utilizzare STARTTLS (per IMAP ed SMTP) e STLS (per POP3) al posto di SSL/TLS, bisognerà specificare la direttiva starttls al posto di SSL:

```
 server_name mail.example.com;
    #...
    starttls             on;
    #...
}
```

2.4 Confronto con Apache

La tabella sottostante mostra le principali differenze tra Nginx e Apache.

Apache	**Nginx**
Apache viene progettato per essere un web server	Nginx è un web server oltre ad essere un proxy server
Un singolo thread può processare una sola connessione	Un singolo thread può processare connessioni multiple
Segue un approccio multi-thread per processare le richieste del client	Utilizza un approccio event-driven per servire le richieste del client
Gestisce il contenuto dinamico all'interno del web server stesso	Non è in grado di elaborare contenuti dinamici in modo nativo
Non riesce a processare richieste multiple contemporaneamente quando il traffico in rete risulta essere intenso	Può elaborare più richieste client contemporaneamente ed efficientemente con risorse hardware limitate
I moduli vengono caricati o scaricati dinamicamente, rendendolo più flessibile	I moduli non possono essere caricati dinamicamente. Devono essere compilati all'interno del software di base
Esegue su tutti i sistemi Unix come Linux, BSD, ecc. oltre a supportare completamente Windows	Gira su sistemi moderni Unix like; tuttavia ha un supporto limitato per Windows

Capitolo 3

Introduzione a Docker

Docker è una piattaforma per lo sviluppo, l'esecuzione e il rilascio di applicazioni, con esso è possibile automatizzare la distribuzione di un'applicazione od un servizio, all'interno di componenti standardizzati, leggeri, portabili e autosufficienti definiti container che renderanno indipendente il servizio sviluppato dal suo ambiente di esecuzione.

I container, sono quindi l'insieme dei dati di cui necessita un'applicazione per essere eseguita: librerie, altri eseguibili, rami del file system, file di configurazione, script, ecc.

Il processo di distribuzione di un'applicazione si riduce alla semplice creazione di una Immagine Docker, ovvero di un file contenente tutti i dati elencati sopra. L'immagine viene utilizzata da Docker per creare un container, un'istanza dell'immagine che eseguirà l'applicazione in essa contenuta.

I container sono autosufficienti perché contengono già tutte le dipendenze dell'applicazione e non richiedono quindi particolari configurazioni sull'host, sono soprattutto portabili, distribuiti in un formato standard (le immagini appunto), che può essere letto ed eseguito da qualunque server Docker. Sono leggeri perché sfruttano i servizi offerti dal kernel del sistema operativo ospitante, invece di richiedere l'esecuzione di un kernel virtualizzato come avviene nel caso delle VM. Ciò permette di ospitare un gran numero di container nella stessa macchina fisica.

Possiamo quindi definirli come base o unità minima, con cui poter definire applicazioni moderne, in particolare quelle che presentano un'architettura a microservizi.

Docker offre una serie di tool ed una piattaforma per gestire il ciclo di vita dei container, dalla creazione all'eventuale rimozione.

3.1 Concetti Fondamentali

Docker Engine

Il Docker Engine è il layer sul quale Docker viene eseguito. E' un ambiente leggero per il rilascio e la gestione di container, immagini e servizi. Nativamente è eseguito in ambiente Linux ed è costituito da tre parti principali:

- Un Docker Daemon eseguito nella macchina host;
- Un client Docker che comunica con il daemon per eseguire comandi;
- Un'API REST per interagire con il Docker Daemon da remoto.

Docker Client

Il client Docker è il modo principale con cui gli utenti interagiscono con Docker. Questo comunica con il demone Docker per l'esecuzione di istruzioni.

Docker Daemon

Il Docker Daemon ascolta le richieste API Docker e gestisce gli oggetti Docker come immagini, contenitori, reti e volumi. Esegue i comandi richiesti dal client, come la build o il run di un container e può comunicare con altri demoni. Il demone Docker viene eseguito sulla macchina ospitante, ma non è possibile comunicarvi.

Docker Image

Una Docker image, o immagine, è una sorta di template predefinito di un container e contiene le istruzioni per definirlo all'interno di un DockerFile. Solitamente un'immagine Docker si basa su un'altra immagine e aggiunge poi dei layer superiori atti a creare la nuova immagine. Questo meccanismo di layering nelle immagini rende Docker molto leggero e veloce. Un'immagine può essere istanziata dando origine ad un container.

3.2 Docker Container

Un container è un'istanza eseguibile di un'immagine. Puoi creare, avviare, arrestare, spostare o eliminare un contenitore utilizzando l'API Docker o l'interfaccia a riga di comando. Puoi connettere un contenitore a una o più reti, allegare spazio di archiviazione ad esso o persino creare una nuova immagine in base al suo stato corrente.

Per impostazione predefinita, un contenitore è relativamente ben isolato dagli altri e dalla sua macchina host.

E' definito dalla sua immagine e da qualsiasi opzione di configurazione che gli viene fornita quando viene creato o avviato. Quando un contenitore viene rimosso, tutte le modifiche al suo stato, che non sono archiviate nella memoria permanente, scompaiono.

Esso possiede un'interfaccia di rete per la comunicazione con l'host ed un proprio indirizzo IP che, di default, è accessibile solamente all'interno del Docker Engine. Per far si che un container sia accessibile anche esternamente dal Docker host, sarà necessario eseguire un mapping tra le porte.

```
$ docker run -d -p 8000:80 nginx
```

Il comando -d viene utilizzato per specificare l'esecuzione del container in background nel Docker Engine, mentre -p 8000:80 serve per esporre il server esternamente così da potervi accedere tramite la porta specificata.

Volumi

I volumi sono il modo in cui i dati all'interno di un container sono resi persistenti. Senza uno o più volumi, qualsiasi dato all'interno di un container attivo che viene rimosso non è persistente. Per far si che i dati all'interno persistano è necessario mappare una directory all'esterno del container con una all'interno, un concetto simile al port mapping per il networking.

Variabili d'ambiente

Le variabili d'ambiente sono un modo conveniente per esternalizzare la configurazione dell'applicazione; pertanto, sono utili anche per la creazione di contenitori Docker. Il più grande vantaggio dell'utilizzo di queste variabili è la flessibilità. Possiamo creare un solo Dockerfile che verrà configurato in modo diverso a seconda dell'ambiente utilizzato per costruire un contenitore. L'altro importante vantaggio riguarda i problemi di sicurezza. Memorizzare password o altre informazioni sensibili direttamente nel Dockerfile probabilmente non è l'idea migliore. Le variabili d'ambiente aiutano a superare questo problema.

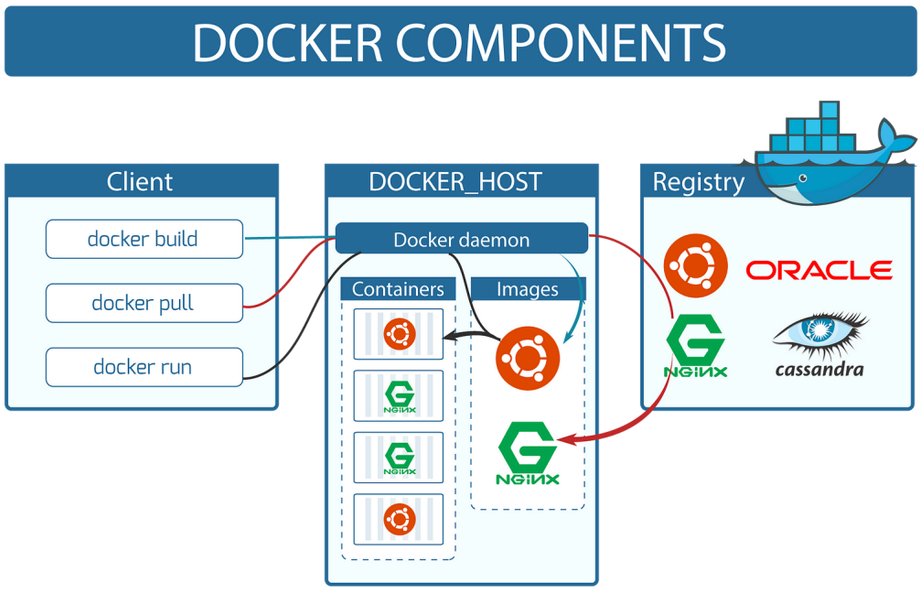


Figura 3.1: Componenti Docker

3.3 Docker compose

Compose è una procedura Docker che permette di definire ed eseguire applicazioni costituite da più container. La configurazione di Compose avviene tramite file in formato YAML e di default ha nome docker-compose.yml. All'interno del file si inseriscono le informazioni utili alla creazione dei vari container che compongono l'applicazione come image, porte, volumi, cosa che avveniva prima a linea di comando. Attraverso Compose quindi è possibile istanziare un ambiente costituito da più container che potenzialmente comunicano tra loro. Un altro grande vantaggio è il fatto che la configurazione è modificabile e, di conseguenza, verranno riavviati solamente i container modificati. Un esempio di struttura per il docker-compose file è il seguente:

```
version: "3.9"
services:
  web:
    build: .
    ports:
      - "8000:5000"
    volumes:
      - .:/code
    environment:
      FLASK_DEBUG: "true"
  redis:
    image: "redis:alpine"
```

L'esempio vuole solo spiegare come può essere definito un file docker-compose. Per eseguire l'ambiente configurato tramite compose è necessario utilizzare il comando:

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

Non è necessario specificare il nome del file perché di default Docker cerca un file denominato docker-compose.yml ma è comunque possibile specificare un altro nome.

Capitolo 4

Utilizzo di NGINX come Reverse Proxy

4.1 Ambiente e strumenti

Per lo sviluppo del progetto sono state utilizzate diverse tecnologie e funzionalità, tra cui Docker. Il servizio Nginx è stato utilizzato come immagine Docker, perciò non è necessario installare l'istanza stand-alone di Nginx. Per riprodurre il progetto è bisognerà installare solamente Docker. Di seguito le tecnologie utilizzate:

- Docker-compose
- NGINX
- GitLab
- OpenSSL
- Docker Registry

4.2 Configurazione di Nginx

4.2.1 Configuration file

Come anticipato nella sezione precedente, per configurare Nginx come reverse proxy occorre solamente specificare la/e location su cui reindirizzare la richiesta del client.

Il file di configurazione risulta quindi tale:

```
stream {

    upstream dockergitlabssh {
       server gitlab:22;
    }

    server {
       listen 22;
       proxy_pass dockergitlabssh;
       ssl_preread on;
    }

}

http {
    client_max_body_size 3000M;

    upstream dockergitlab {
        server gitlab:443;
    }

    upstream dockerregistry {
         server ui:80;
    }
```

```
25
26      server {
27          listen 80 default_server;
28          listen 443 ssl;
29
30          ssl_certificate /etc/nginx/ssl/certificate.crt;
31          ssl_certificate_key /etc/nginx/ssl/privateKey.key;
32
33          server_name gitlab.localhost;
34          server_tokens off;
35
36          location / {
37              proxy_pass https://dockergitlab;
38          }
39      }
40
41      server {
42           #...
43           location / {
44               proxy_pass http://dockerregistry;
45           }
46      }
47  }
48
```

Come mostra la figura, sono presenti un blocco http e uno stream. All'interno di quello http troviamo altri due blocchi sever, utilizzati per reindirizzare le richieste rispettivamente a GitLab e al Registry.

In entrambi i server troviamo il *server_name* , quest'ultimo determina quale blocco server debba venire impiegato per una determinata richiesta, le porte in ascolto e i certificati. Il server è in ascolto sulla porta 443 ch è indispensabile per avere trasferimenti sicuri. Tramite i comandi *ssl_cetificate* e *ssl_certificate_key* vengono allegati i certificati necessari per proteggere la trasmissione. La keyword *client_max_body_size* stabilisce la dimensione massima del body rispetto alla richiesta del client.

Il secondo blocco è quello dell'upstream, utilizzato per stabilire una connessione SSH con il client. Il modulo *ssl_preread*, impiegato per rilevare più di un protocollo, consente l'inoltro proxy a diversi server back-end, in base al nome host SSL richiesto.

4.2.2 Docker compose

Definisce le informazioni necessarie per la creazione del container Nginx come segue:

```
1  nginx:
2      image: nginx:latest
3      container_name: nginx
4      ports:
5        - "80:80"
6        - "443:443"
7        - "22:22"
8      restart: always
9      volumes:
10       - ./proxy/nginx/nginx.conf:/etc/nginx/nginx.conf
11       - ./proxy/nginx/ssl/:/etc/nginx/ssl/:ro
12     depends_on:
13       - gitlab
14       - registry
15
```

Nginx è in ascolto sulle porte 80, 443 e 22 per garantire un servizio HTTP/S e SSH. Viene settato il *restart:always* in modo che il container cerchi sempre di ripartire nel caso in cui si verifichino degli errori.

Vengono utilizzati i volumi per condividere il file di configurazione di Nginx e i certificati con il container, in modo da avere persistenza.

Il blocco si conclude specificando che il container che verrà creato dipenderà dai servizi GitLab e Registry, i quali dovranno essere già attivi al momento della creazione di quest'ultimo.

4.3 GitLab

All'interno del file Docker compose si determina la struttura del container relativo a GitLab come segue:

```
gitlab:
    image: gitlab/gitlab-ee:latest
    restart: always
    container_name: gitlab
    hostname: 'localhost'
    ports:
      - '8022:22'
      - '8080:80'
      - '8443:443'
    volumes:
      - './gitlab/config:/etc/gitlab'
      - './gitlab/logs:/var/log/gitlab'
      - './gitlab/data:/var/opt/gitlab'
```

Vengono abilitate le porte necessarie per garantire tutti e tre i tipi di servizi, esattamente come con Nginx.

Si utilizza la condivisione per mantenere persistenti le modifiche effettuate al file di configurazione, oltre a rendere possibile la lettura di file di log, creati durante l'esecuzione.

4.3.1 Configurazione SSL

L'abilitazione della comunicazione HTTPS in GitLab avviene modificando il file *gitlab.rb*. Questo viene creato durante la prima esecuzione del servizio, all'interno della directory */gitlab/config*.

Di default la comunicazione non è abilitata, ma può diventarlo in due modi diversi. Tramite:

- utilizzo di Let's Encrypt, una Certificate Authorithy che rilascia gratuitamente certificati TLS della durata di 90 giorni;
- configurazione manuale di HTTPS con certificati creati dall'utente.

Nel progetto è stata scelta la prima modalità.

```
external_url 'https://gitlab.example.com'
#...
letsencrypt['enable'] = true
letsencrypt['contact_emails'] = ['foo@email.com']
```

Let's Encrypt è abilitato di default se un*external_url* è impostato con il protocollo HTTPS e non sono configurati altri certificati.

Gli unici prerequisiti richiesti riguardano l'accesso alle porte 80 e 443 da parte dei server Let's Encrypt pubblici, che eseguono i controlli di convalida. La validazione non funziona con porte non standard.

I certificati scadono ogni 90 giorni. Gli indirizzi e-mail specificati per *contact_emails* riceveranno un avviso quando si avvicina la data di scadenza. Per ovviare a questa problematica, vengono pianificati i rinnovi dopo la mezzanotte di ogni quarto giorno del mese. Il minuto è determinato dal valore in external_url, per aiutare a distribuire il carico sui server Let's Encrypt upstream.

Per impostare in modo esplicito i tempi di rinnovo:

```
#...
letsencrypt['auto_renew'] = true
letsencrypt['auto_renew_hour'] = 12
letsencrypt['auto_renew_minute'] = 30
letsencrypt['auto_renew_day_of_month'] = "*/7"
```

Nel caso si preferisca configurare i certificati manualmente, basterebbe togliere il commento alle seguenti righe di codice, modificando il path:

```
nginx['ssl_certificate'] = "/etc/gitlab/ssl/gitlab.example.com.crt"
nginx['ssl_certificate_key'] = "/etc/gitlab/ssl/gitlab.example.com.key"
```

Una volta concluse le modifiche basterà eseguire:

```
sudo docker exec gitlab gitlab-ctl reconfigure
```

oppure all'interno del container :

```
gitlab-ctl reconfigure
```

4.4 Docker Registry

Il Registry è un'applicazione lato server stateless altamente scalabile che archivia e consente di distribuire immagini Docker.

Per facilitare la visualizzazione dei file al suo interno, è stata aggiunta un'interfaccia grafica.

```
registry:
    image: registry:latest
    container_name: registry
    restart: always
    ports:
      - "5000:5000"
    environment:
      REGISTRY_STORAGE_FILESYSTEM_ROOTDIRECTORY: /registry/data
      REGISTRY_HTTP_ADDR: 0.0.0.0:5000
      REGISTRY_HTTP_TLS_CERTIFICATE: /etc/ssl/certificate.crt
      REGISTRY_HTTP_TLS_KEY: /etc/ssl/privateKey.key
      REGISTRY_HTTP_HEADERS_Access-Control-Allow-Origin: "['*']"
    volumes:
      - ./registry/data:/data
      - ./proxy/nginx/ssl/:/etc/ssl

ui:
    image: joxit/docker-registry-ui:latest
    container_name: ui
    ports:
      - '8088:80'
    environment:
      REGISTRY_TITLE: 'My Private Docker Registry'
      REGISTRY_URL: 'https://localhost:5000'
    depends_on:
      - registry
```

Il blocco Registry, oltre ad avere diverse directory in condivisione con il container, definisce delle variabili d'ambiente necessarie per comunicare in modalità sicura. E' possibile aggiungere eventuali variabili di login se lo si desidera.

L'interfaccia grafica, che è dipendente dal Registry, definisce il path e il titolo di quest'ultimo.

4.5 Certificati

Dato lo sviluppo dell'applicazione in locale, i certificati sono stati creati manualmente tramite OpenSSL.

OpenSSL viene infatti utilizzata per creare una o più CA di tipo self signed, tale modalità si usa quando non si rende necessario che una root CA esterna firmi i nostri certificati, oppure non possa firmarli essendo l'applicazione sviluppata in locale.

Una volta installato Openssl, per la generazione dei propri certificati è stato utilizzato il seguente comando:

```
openssl req -x509 -sha256 -nodes -days 365 -newkey rsa:2048
-keyout privateKey.key -out certificate.crt
```

Capitolo 5

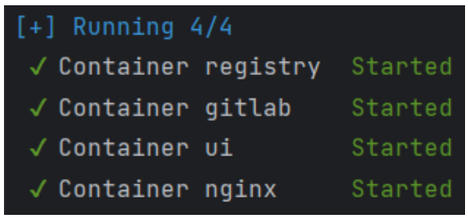
Testing

In questa sezione, a partire dalla configurazione di Nginx e dei vari container discussi in precedenza, si testerà il funzionamento dell'applicazione, ovvero, effettuare la connessione a GitLab, tramite HTTP e SSH, e al Registry.

Inizialmente vengono istanziati i diversi container utilizzando Docker. Nel caso in cui si stia lavorando con Windows, sarà necessario far partire Docker Desktop prima di lanciare il comando sottostante.

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

Ottenendo come risultato le seguenti figure:



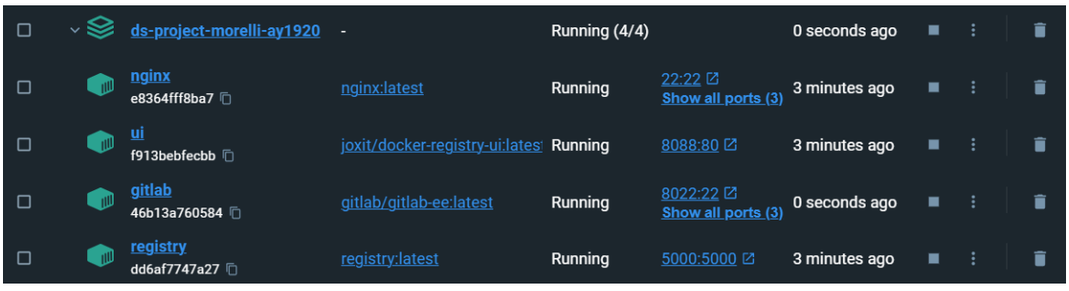


Figura 5.1: Container attivi

La prima, rappresenta la vista della linea di comando e la seconda la dashboard di Docker Desktop.

Prima di procedere con l'accesso ai vari servizi, se l'utente lo desidera, potrà sostituire i certificati presenti all'interno della cartella *proxy/nginx/ssl* del progetto con quelli di sua creazione. Le istruzioni sono presenti nel capitolo precedente.

5.1 GitLab

5.1.1 Connessione HTTP

Una volta partito il container GitLab, per connetterci tramite HTTP, ci basterà digitare l'url del servizio sulla barra di ricerca in un browser a piacimento.

Scrivendo quindi `gitlab.localhost` otterremo:

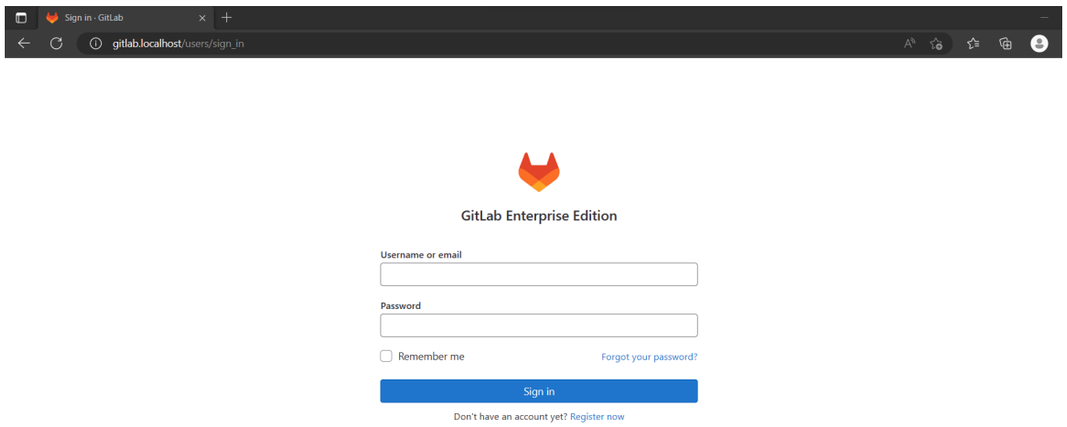


Figura 5.2: Servizio GitLab

Per effettuare il login saranno necessari alcuni passaggi in più. Una volta creato il container, per il primo accesso, occorrerà utilizzare l'utente `root` e definire una nuova password.

All'interno del container GitLab bisognerà lanciare il seguente comando:

```
gitlab-rake 'gitlab:password:reset[root]'
```

Verranno quindi mostrate le seguenti richieste a riga di comando.

```
gitlab-rake 'gitlab:password:reset[root]'
Enter password:
Confirm password:
Password successfully updated for user with username root.
```

Figura 5.3: Cambio password in GitLab

Una volta inserita la nuova password, se questa risulterà soddisfare il criterio di lunghezza di 8 caratteri, si riceverà il messaggio di conferma del cambiamento avvenuto con successo.

Si potrà quindi procedere ad effettuare il login nel browser.

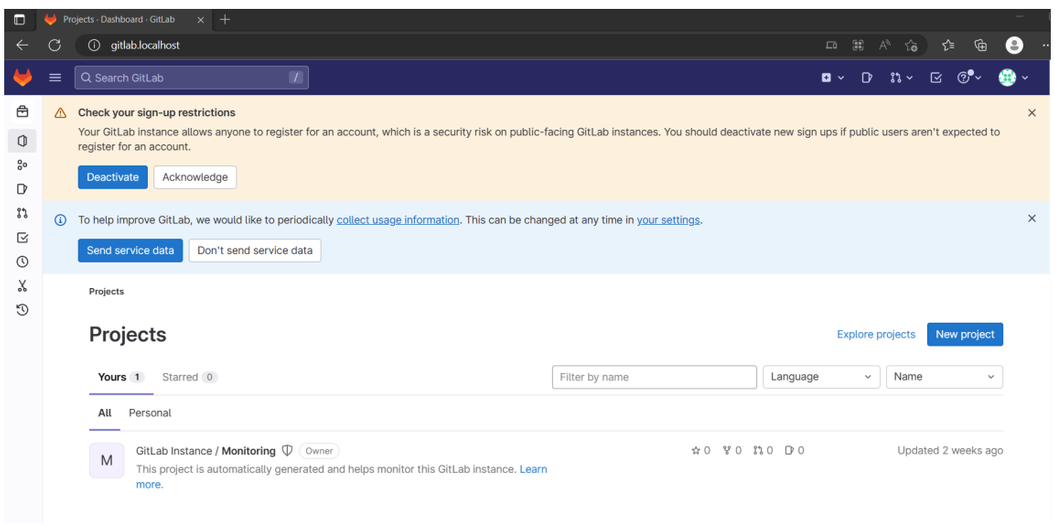


Figura 5.4: Schermata home GitLab

5.1.2 Connessione SSH

Per effettuare una connessione SSH sarà necessario completare i passaggi elencati di seguito

Generazione chiavi SSH

Il protocollo SSH utilizza la crittografia a chiave pubblica per l'autenticazione di host e utenti. Le chiavi di autenticazione, chiamate chiavi SSH, vengono create utilizzando il programma `keygen`nel seguente modo:

```
ssh-keygen -o
```

Le chiavi verranno salvate all'interno della directory dell'utente nella cartella */.ssh*.

Aggiunta chiave pubblica a GitLab

Una volta create, bisognerà inserire la chiave pubblica (.pub) all'interno dell'account GitLab con cui si desidera connettersi, tramite il path `http://gitlab.localhost/-/profile/keys`.

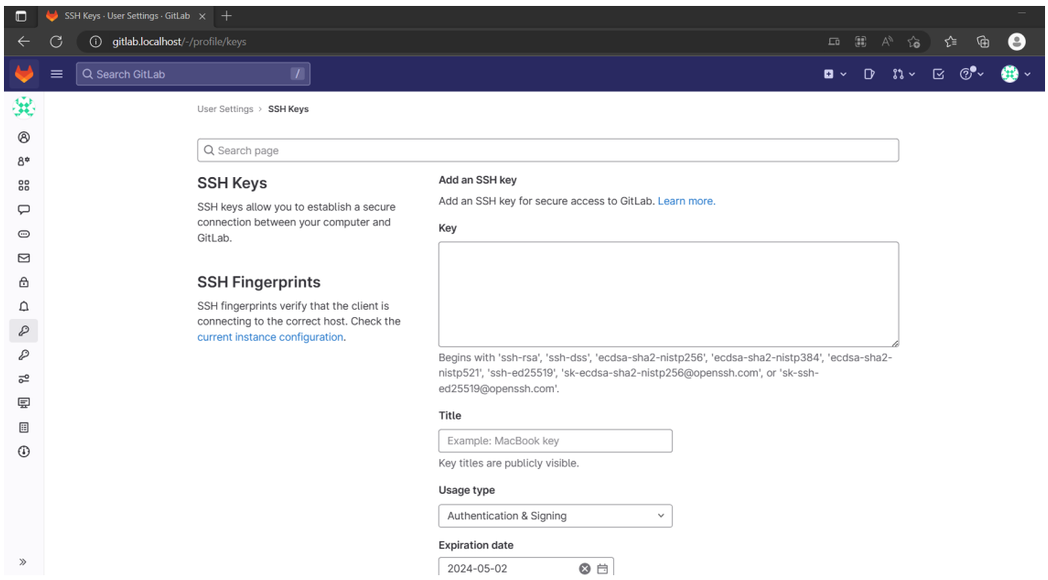


Figura 5.5: Schermata menu aggiunta chiavi SSH

Le chiavi aggiunte al profilo sono visibili in fondo alla schermata mostrata sopra.

Configurazione dell'host

Servirà poi creare un file di configurazione all'interno della cartella */.ssh*, necessario per definire l'host, l'utente, la chiave di autenticazione. Una volta aperto un terminale bash all'interno della cartella, bisognerà eseguire i seguenti comandi:

```
$ touch config
$ nano config
```

e aggiungere le righe riportate qui sotto:

```
Host localhost
HostName localhost
Port 22
User root
PreferredAuthentications publickey
IdentityFile ~/.ssh/id_rsa
```

I seguenti comandi sono validi se le chiavi sono state salvate con il nome `id_rsa` e se si vuole accedere con l'utente `root`, altrimenti i campi dovranno essere modificati in modo appropriato.

Creazione del tunnel SSH

Terminata la configurazione, per connettersi tramite SSH, sarà necessario aprire un nuovo terminale ed eseguire:

```
ssh -T git@localhost
```

Se la connessione verrà stabilita correttamente, si riceverà la conferma, come mostra la seguente immagine.

Figura 5.6: Connessione SSH avvenuta con successo

5.2 Registry

Esattamente come per GitLab, è possibile usufruire del servizio Registry digitando il seguente link `http://registry.localhost/` nella barra di un browser a piacimento.

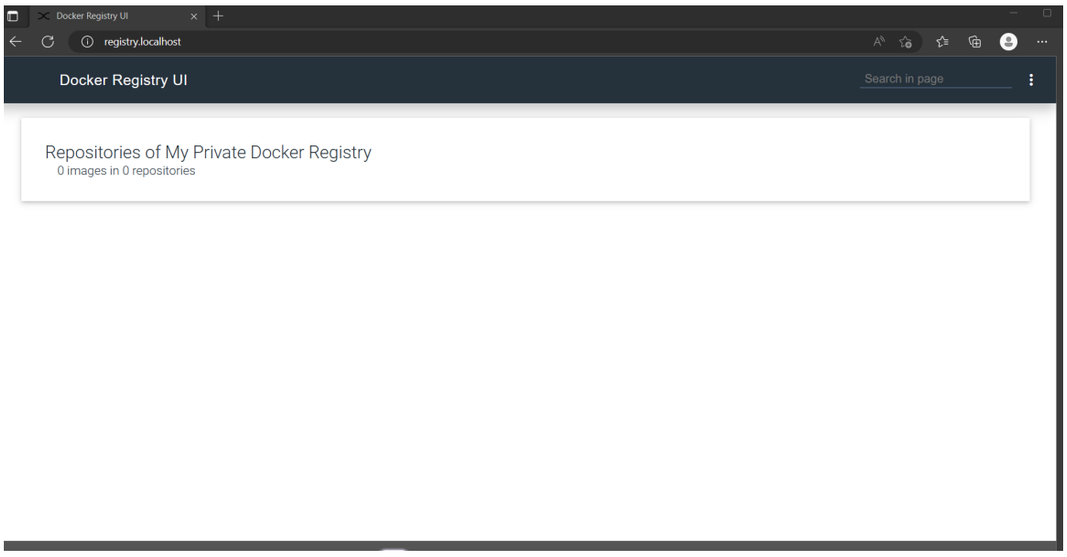


Figura 5.7: Schermata servizip Registry

E' stata aggiunta, inoltre, un'interfaccia grafica, come già menzionato nel capitolo precedente, per facilitarne l'utilizzo.

Per verificare il funzionamento del Registry è possibile aggiungere un'immagine tramite l'esecuzione dei seguenti comandi in un terminal:

```
docker login localhost:5000
docker tag <source-image> locahost:5000/<target-image>
```

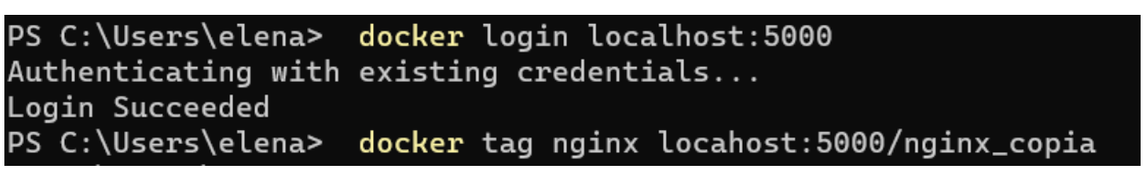


Figura 5.8: Schermata servizio Registry

Una volta effettuato il login, verrà scelta un'immagine, già presente all'interno di Docker, da aggiungere al Registry, settando un nuovo nome al posto di `<target_image>`

```
docker images
docker push <target-image>
```

Figura 5.9: Visualizzazione delle immagini e push nel Registry

Il primo comando permetterà di controllare se la nuova immagine è stata aggiunta correttamente alla lista di quelle esistenti. Il secondo effettuerà il push all'interno del Registry.

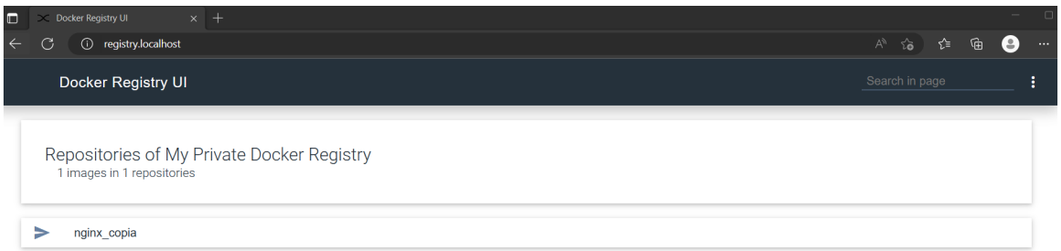


Figura 5.10: Schermata del Registry dopo l'aggiunta di un'immagine

Una volta caricata, l'immagine sarà visibile anche all'interno dell'interfaccia grafica del Registry.

Capitolo 6

Conclusioni

Nginx è stato progettato per gestire un elevato volume di richieste contemporaneamente. Pertanto, offre tempi di caricamento più rapidi e prestazioni migliori rispetto alla maggior parte degli altri web server.

Utilizza meno risorse e hardware rispetto ai competitor, rendendolo una soluzione più economica. È compatibile con una vasta varietà di applicazioni web. E' inoltre dotato di un'interfaccia moderna e di impostazioni di configurazione user-friendly.

Tuttavia, Nginx presenta anche alcuni aspetti negativi. Ad esempio, ha solo un singolo file di configurazione, rendendolo meno flessibile di Apache.

Ciò nonostante, se le prestazioni sono la priorità assoluta, Nginx potrebbe essere la scelta giusta. Risulta ideale per tutti i siti web con grandi quantità di traffico, in quanto si ridimensiona meglio di Apache o di altri concorrenti.

Un potente software server come questo può aiutare a ridurre al minimo i tempi di inattività e a prevenire lunghi tempi di caricamento. Significa che Nginx può servire contenuti ai suoi utenti in modo più efficiente, portando a più conversioni.

Nel 2019 Nginx è stato acquisito dall'azienda F5, diventano così una loro unità di business. Da allora hanno iniziato a fornire una suite di prodotti che insieme costituiscono il nucleo di ciò di cui le organizzazioni hanno bisogno per creare app e API con prestazioni, affidabilità, sicurezza e scalabilità. Tra gli open source troviamo Nginx unit, Nginx service mesh ed Nginx amplify.

Nonostante Apache continui ad essere, in termini assoluti, il web server più utilizzato, si può notare come Nginx si sia creato una notevole fetta di mercato tra i siti di fascia più alta, diventando a tutti gli effetti uno standard nel mercato.

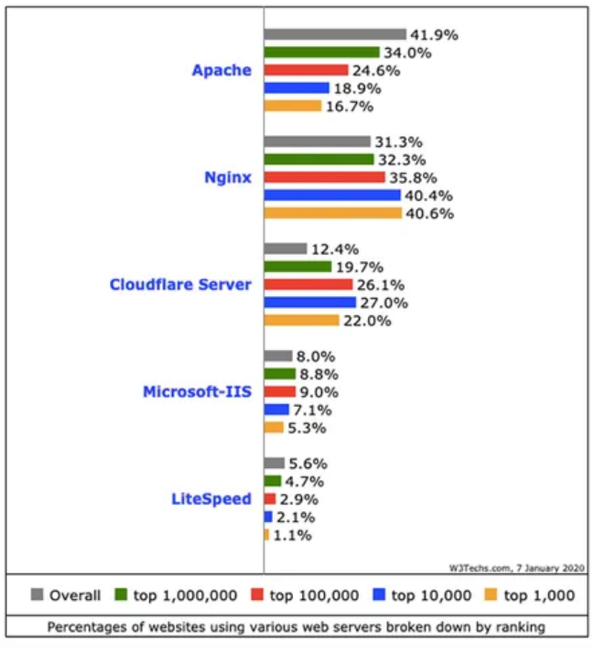


Figura 6.1: Schermata del Registry dopo l'aggiunta di un'immagine

6.1 Progetto

Il progetto è stato svolto in maniera più o meno lineare, con la comparsa di alcune problematiche nate dalla scarsa documentazione presente online sulla gestione della comunicazione SSH.

L'unico elemento mancante nel progetto è stato l'impiego di Certbot per la creazione automatica di certificati Let's Encrypt su siti web amministrati manualmente, per abilitarne HTTPS, in quanto non utilizzabile in uno sviluppo puramente locale.

Bibliografia

[1] Nginx *Nginx Documentatation*, URL: https://nginx.org/en/docs/

[2] Nginx *Nginx Documentatation*, URL: https://www.html.it/guide/nginx-guida-al-web-server/

[3] Nginx *Nginx Documentatation*, URL: https://it.wikipedia.org/wiki/Nginx

[4] Nginx *Nginx Documentatation*, URL: https://www.slideshare.net/DhrubajiMandal/nginx-176332309

[5] Nginx *Nginx Architecture Documentatation*, URL: https://aosabook.org/en/v2/nginx.html

[6] Nginx *Nginx Architecture Documentatation*, URL: https://tech.giuneco.it/nginx-levoluzione-dei-web-server/

[7] GitLab *GitLab Documentation*, URL: https://docs.gitlab.com/ee/

[8] Registry *Doker Registry Documentation*, URL: https://docs.docker.com/registry/deploying/