Deep dive in Docker: trucchi e suggerimenti per costruire immagini Docker ottimizzate per sicurezza e dimensioni.

3 Gennaio 2022 - 7 min. read

Amazon's Elastic Container Registry | *Containers* | *Docker*

Una Breve Introduzione

Oggigiorno abbiamo sempre più workload basati su Docker ed è di grande importanza tenere le nostre immagini Docker sicure ed ottimizzate per il cloud. Abbiamo già discusso su Docker nel nostro blog, potete approfondire l'argomento con le origini di Docker o con un caso d'uso con Wordpress.

Nei prossimi paragrafi vedremo alcuni suggerimenti e trucchi su come evitare le falle di sicurezza più comuni e come ottimizzare le dimensioni delle nostre immagini Docker.

Ma prima dobbiamo capire uno dei concetti chiave di Docker: il sistema di layering.

Il Dockerfile è il file principale che definisce il modo in cui un'immagine Docker è costruita. In esso viene definita l'immagine di base da cui partire, il modo in cui l'applicativo viene pacchettizzato e come viene eseguito.

Quando avviamo il processo di build a partire da un Dockerfile, il motore di Docker crea una serie di layer, uno per ciascun comando nel Dockerfile. Ogni volta che eseguiamo un comando all'interno del Dockerfile, un nuovo layer viene creato al di sopra del precedente. Il motore di Docker esegue i comandi in modo sequenziale e infine unisce tutti i layer per creare l'immagine finale.

Come posso ridurre le dimensioni?

Usa un'immagine di base piccola. Alpine è una buona scelta

Generalmente, avere un'immagine piccola velocizza la fase di build, deploy ed esecuzione. Per ottenere questo, una buona idea è partire da un'immagine di base che è essa stessa piccola. Puoi usare `alpine:3.15` che è di circa 5.6 MB invece di `ubuntu:20.04` che è (al momento di scrittura di questo articolo) di 72 MB.

Alpine è una distribuzione Linux costruita con musl libc e BusyBox che punta alla semplicità, sicurezza e all'efficienza delle risorse. Essa contiene soltanto i pacchetti assolutamente necessari perché... *se una cosa non c'è, non può rompersi*.

Bisogna tenere in mente che ha anche alcuni svantaggi. Per esempio, visto che Alpine è basata sulla libreria C musl, invece che sulla libreria più diffusa GNU C Library, ci possono essere problemi con alcune dipendenze in C.

Usa i Dockerfile multi-stage

Oltre a usare Alpine come immagine di base, un altro metodo per ridurre le dimensioni delle immagini Docker è l'uso delle build multistage. Una build multistage consiste in un unico Dockerfile in cui definiamo più istruzioni FROM, ed ognuna di esse è uno stage.

In ogni FROM possiamo partire da uno stage precedente per trarre vantaggio dagli artefatti di build in cache oppure possiamo partire da un'immagine di base totalmente diversa e copiare solamente alcuni artefatti dallo stage precedente.

Vediamo un veloce esempio:

```
# Base image with dependencies
FROM node:17.3.0-alpine3.12 AS base
WORKDIR /app
# Copy package.json and package-lock.json
COPY package*.json ./
# Install dependencies
RUN npm install
```

```
# Build Stage
FROM base AS build
WORKDIR /app
COPY . ./
# Build and bundle static files
RUN npm run build



# Release Stage
FROM node:17.3.0-alpine3.12 AS release
WORKDIR /app
COPY --from=base /app/package.json ./
# Install app dependencies
RUN npm install --only=production
COPY --from=build /app/dist/ ./
CMD [ "npm", "run", "start" ]
```

In questo Dockerfile multistage abbiamo creato un'immagine di base che ha tutte le dipendenze della nostra applicazione, abbiamo creato uno stage per fare la build dell'applicazione e infine, per l'immagine di runtime, abbiamo usato un'immagine di base pulita, ci abbiamo installato le dipendenze di produzione e copiato gli artefatti necessari.

Di default vengono costruiti tutti gli stage ma possiamo specificarne uno in particolare con il flag `--target <stage-name>`

`docker build --target release --tag image .`

Puoi fare riferimento alla documentazione ufficiale per ulteriori dettagli.

Sfrutta il caching

Per trarre pieno vantaggio dal caching dei layer, scrivi il Dockerfile in modo tale che i comandi che non cambiano spesso vengano eseguiti prima degli altri. In questo modo i layer che vengono creati prima saranno tenuti in cache e riutilizzati. Sebbene questo non cambi la dimensione finale dell'immagine, farà in modo che le build siano più veloci.

Usa .dockerignore

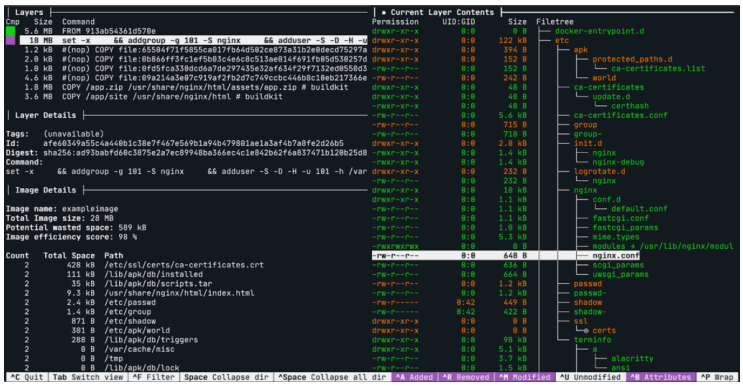
Aggiungi i file temporanei, gli artefatti di build intermedi, i file di sviluppo locale e le informazioni sensibili nel .dockerignore per escluderli. La prossima volta che farai partire una build sarà più veloce perchè il `build context` di Docker sarà più piccolo.

Comunque non dovresti avere informazioni sensibili nella tua repository :)

Tieni un occhio sui layer – Dive

Dive è uno strumento a linea di comando che ti permette di ispezionare il contenuto di ciascun layer in un'immagine Docker. Può essere molto utile vedere cosa è cambiato ad ogni step nel Dockerfile, perchè in questo modo puoi vedere se ci sono ridondanze o file non necessari nell'immagine.

Per vederlo in azione esegui `dive <image-name>`



Le feature principali di Dive sono:

Mostrare il contenuto di ciascun layer: Fornisce una vista ad albero di tutto il contenuto dell'immagine in uno specifico layer.

Filtra per file nuovi, modificati, rimossi: con i comodi shortcut puoi abilitare/disabilitare la visualizzazione di specifici file.

Stimare l'efficienza dell'immagine: Una feature sperimentale che ti mostra quanto spazio viene sprecato.

Bonus: automatizziamo!

Puoi mettere la variabile di ambiente `CI=true` per disabilitare la visualizzazione interattiva e integrare Dive con una pipeline CI/CD.

Puoi creare un file .dive-ci file per personalizzare 3 parametri per l'accettazione dei test:

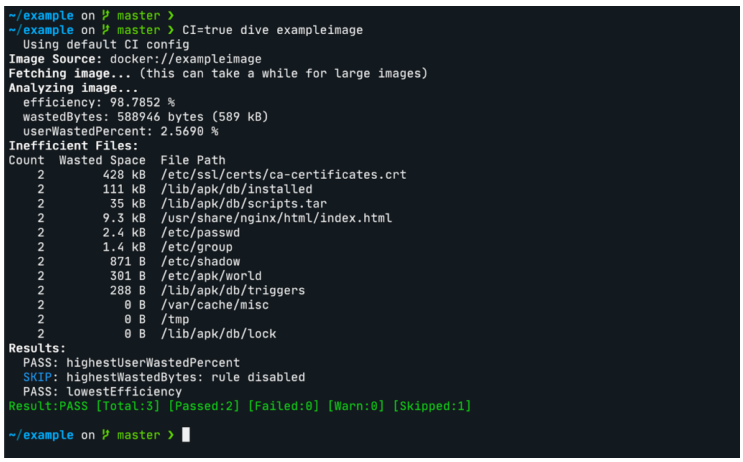
`lowestEfficiency`, `highestWastedBytes` e `highestUserWastedPercent`.

.dive-ci file:

```
rules:
 # Lowest allowed efficiency percentage (value range between 0-1).
 lowestEfficiency: 0.95

 # If the amount of wasted space is at least X or larger than X, mark as failed.
 # Expressed in B, KB, MB, and GB.
 highestWastedBytes: 20MB

 # If the amount of wasted space makes up for X% or more of the image, mark as failed.
 # Note: the base image layer is NOT included in the total image size.
 # Expressed as a ratio between 0-1; fails if the threshold is met or crossed.
 highestUserWastedPercent: 0.10
```



Regole di base per la sicurezza

Una regola del pollice base è partire dalla **versione stabile più recente** dell'immagine.

In questo modo, sarai sicuro di avere le ultime patch di sicurezza per gli strumenti e le librerie native. La stessa regola è applicabile anche alle dipendenze a livello di applicazione.

Un'altra regola di base è **ridurre al minimo la superficie di attacco** assicurandoti di avere esclusivamente il set minimo e strettamente necessario di strumenti e librerie per eseguire l'applicazione. Per far ciò, puoi iniziare da un'immagine di base molto piccola (qualcosa come Alpine) e, quindi, installare le dipendenze applicative su di essa.

Un'altra buona pratica consiste nel sfruttare gli **strumenti di scansione delle vulnerabilità** per verificare la presenza di criticità note nelle immagini Docker. Per farlo AWS mette a disposizione il servizio Elastic Container Registry, il quale consente di scansionare le immagini senza costi aggiuntivi!

AWS ECR Scan

Amazon's Elastic Container Registry (ECR) è un registro di container completamente gestito che, tra le numerose feature, offre la possibilità di identificare vulnerabilità attraverso la scansione delle immagini.

Amazon ECR offre due opzioni di scansione

Basic scanning

Questa opzione, completamente gratuita, permette la scansione sia automatica, che manuale delle immagini basandosi sul database Common Vulnerabilities and Exposures (CVEs) del progetto open-source *Clair*.

Enhanced scanning

Integrandosi con il servizio Amazon Inspector, Amazon ECR è in grado di scansionare in modo automatico i repository, in modo che le immagini dei container vengano continuamente analizzate. In caso di nuove vulnerabilità rilevate, Amazon Inspector ci notificherà attraverso il servizio Amazon EventBridge.

È anche possibile scansionare manualmente un'immagine direttamente dalla console AWS, oppure eseguendo il seguente comando da CLI:

```
aws ecr start-image-scan --repository-name nginx-test --image-id imag
eTag=base
{
        "registryId": "424242424242",
        "repositoryName": "nginx-test",
        "imageId": {
        "imageDigest": "sha256:61191087790c31e43eb37caa10de1135b002f1
0c09fdda7fa8a5989db74033aa",
        "imageTag": "base"
        },
        "imageScanStatus": {
        "status": "IN_PROGRESS"
        }
}
```

Eseguiamo poi il seguente comando per verificare lo stato della scans ione:

```
aws ecr describe-image-scan-findings --repository-name nginx-test --i
mage-id imageTag=base
{
        "imageScanFindings": {
        "findings": [...],
        "imageScanCompletedAt": "2021-12-22T11:44:55+01:00",
        "vulnerabilitySourceUpdatedAt": "2021-12-22T01:26:43+01:00",
        "findingSeverityCounts": {
                "HIGH": 8,
                "MEDIUM": 41,
                "LOW": 23,
                "UNDEFINED": 1,
                "INFORMATIONAL": 67
        }
```

```
        },
        "registryId": "4242424242424242",
        "repositoryName": "nginx-test",
        "imageId": {
        "imageDigest": "sha256:61191087790c31e43eb37caa10de1135b002f1
0c09fdda7fa8a5989db74033aa",
        "imageTag": "base"
        },
        "imageScanStatus": {
        "status": "COMPLETE",
        "description": "The scan was completed successfully."
        }
}
```

Scan on push

Puoi configurare una repository per eseguire la scansione delle immagini al momento del push attivando la feature `ScanOnPush`. In alternativa puoi abilitare questa feature su tutti i repository attraverso questo settaggio sulla Console:

`Amazon ECR > Private registry > Scanning configuration > Basic scanning`

Abilitando questa funzione si potrà beneficiare anche dell'Integrazione con Amazon EventBridge: dato che ogni scansione triggererà un evento, questo potrà essere facilmente intercettato utilizzando EventBridge. Saremo così in grado di costruire un sistema di notifica event-driven in grado di avvisarci ogniqualvolta venga rilevata una criticità di sicurezza nelle immagini.

In conclusione

In questo articolo abbiamo analizzato le fasi fondamentali del processo di build in Docker fornendo consigli e best practices per ottimizzare la creazione e il mantenimento delle immagini, con particolare attenzione a performance, costi e sicurezza.

Per quanto riguarda la dimensione delle immagini e la durata del processo di build, abbiamo svelato alcuni trucchi e suggerimenti relativi ai Dockerfile.

Abbiamo poi visto come utilizzare Dive per ispezionare i layer e come integrarlo in una pipeline.

Infine, ci siamo focalizzati sul servizio Amazon ECR per automatizzare la scansione delle nostre immagini.

Se siete ancora curiosi sull'argomento e volete approfondire ulteriormente uno o più punti trattati nell'articolo, ecco alcuni link utili:

Alpine Linux

AWS ECR

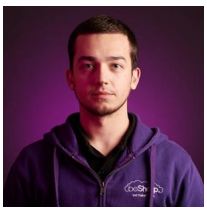
Docker Multi Stage Build

Clair Project

Dive

Se invece "vi siete già sporcati le mani" con Docker e volete condividere le vostre considerazioni, commentate o scriveteci! Saremo felici di discuterne con voi! ;)

A presto su Proud2beCloud con molti nuovi articoli per veri Cloud-addicted!



Mehmed Dourmouch

DevOps Engineer. Very Dev, not so Ops. I like to break things and see what happens, I also automate everything. I often participate in cybersecurity CTFs and in my free time I produce cacophony with my guitar.
