Università degli Studi di Milano

Master in Data Science for Economics, Business and Finance

**Cloud, Distributed and Parallel Computing**

Prof. Anisetti, Ardagna, Gaudenzi

Progetto di

Sabrina Cenghialta, Laura Romano, Guido Ravasi, Matteo Serra

**Task progetto di Analisi Dati**

Il progetto di esame deve comprendere le seguenti componenti: Relazione di progetto (che riporta gli snippet di codice commentato e descrive come le scelte di sviluppo si rifanno ai 12Factor, manuale di build e deployment, Repository Git con codice, script di installazione e dockerfile).

I progetti disponibili per Analisi dati sono rintracciabili:

- <https://github.com/awesomedata/awesome-public-dataset>

- <https://github.com/italia/awesome-italian-public-datasets>

**Dataset e Repository**

Nel nostro caso abbiamo svolto un progetto utilizzando il dataset seguente: FIFA 2021 Complete Player Dataset rintracciabile su Kaggle: [FIFA 2021 Complete Player Dataset | Kaggle](https://www.kaggle.com/aayushmishra1512/fifa-2021-complete-player-data/tasks)

Il progetto è presente sul seguente repository:

<https://github.com/MrTeoTZR/Cloud-project-Fifa>

**Twelve Factor**

Considerato che nel corso e nella richiesta del progetto viene posta enfasi ai Twelve Factor, riteniamo opportuno una breve descrizione delle 12 caratteristiche che un’applicazione Cloud Native deve avere e che dobbiamo tenere rigorosamente presente nello svolgimento del progetto:

1. *Codebase*: è l’insieme di tutto il codice necessario alla mia app (codice di installazione, codice di configurazione, codice di promozione ecc). Il codice deve essere versionato e deve poter gestire tutti i deploy di cui abbiamo bisogno.
2. *Dependencies*: occorre poter gestire in maniera opportuna le dipendenze, quindi dovremo esplicitamente dichiarare le dipendenze (e le loro versioni). Le dipendenze sono tutte le librerie necessarie all’applicazione per poter funzionare. Ogni microservizio isola e impacchetta le proprie dipendenze, includendo cambiamenti senza influenzare l’intero sistema.
3. *Configuration*: Nel codebase deve essere salvato tutto quello che riguarda la configurazione. File di configurazione significa che l’applicazione deve poter essere compatibile con diversi ambienti. Le informazioni di configurazione vengono rimosse dai microservizi ed esternalizzate al di fuori del codice, attraverso uno strumento di gestione della configurazione.
4. *Backing Services*: I backing services sono quei servizi dove appoggiamo le informazioni: sono servizi che ci rendono l’applicazione statefull (come i database, le code, il file system, gli object storage come AWS S3). Sono i luoghi in cui la nostra applicazione va a prendere informazioni. Dobbiamo gestire i backing services come risorse disaccoppiate dall’applicazione, che possono essere modificate attraverso le configuration.
5. *Build, Release, Run*: Build, release e run devono essere separate, quiondi. Occorre riuscire a costruire una catena dove l’operazione di build, l’operazione di release e l’operazione di run sono 3 operazioni ben distinte.
6. *Processes*: Ogni microservizio dovrebbe essere eseguito nel proprio processo, isolato da altri servizi in esecuzione. L’applicazione che sviluppiamo deve avere uno stato [un salvataggio] che si trova fra i backing services. Ogni macchina deve essere stateless: deve salvare lo stato da un’altra parte.

Lo step 6 è molto importante perché l’applicazione Cloud Native deve poter scalare orizzontalmente.

1. *Port Binding*: occorre sapere quali porte vengono esportate dal mio servizio. Il servizio ha un indirizzo, delle porte e delle route e occorre sapere quale porta viene esportata.
2. *Concurrency*: questo fattore esamina le pratiche per il ridimensionamento dell'app. Tali pratiche vengono utilizzate per gestire ciascun processo nell'app in modo indipendente, ad esempio avvio / arresto, clonazione su macchine diverse ecc. Il fattore si occupa anche di suddividere la tua app in pezzi molto più piccoli. La concurrency prevede anche che ci sia un modello che scali gestendo la concorrenza in maniera opportuna.
3. *Disposability*: la disposability esamina la robustezza dell'app con metodi di avvio e spegnimento rapidi. Poiché devo scalare orizzontalmente (quindi aggiungere risorse e toglierle), l’applicazione deve poter avere nuove risorse in pochissimo tempo e, quando tolgo risorse, non devo creare problemi all’infrastruttura. Le applicazioni Cloud sono applicazioni nate per gestire il fallimento: devono poter gestire il fatto che non ci sia il backing service o che ci siano picchi di richieste inaspettate. Devono quindi essere app robuste, sviluppate con in mente la security by design (sviluppate pensando agli aspetti di sicurezza).
4. *Development and Production Parity*: riprende i primi 4 punti: le dipendenze non devono essere costruite per l’ambiente di development e per l’ambiente di sviluppo ma ci vuole un solo file di dipendenze che sia efficiente sia per la produzione, sia per lo sviluppo. In caso di distribuzione continua, è necessario disporre di un'integrazione continua basata su ambienti corrispondenti per limitare deviazioni ed errori.
5. *Registri:* dobbiamo poter fare monitoraggio di ciò che succede. I meccanismi di registrazione sono fondamentali per il debug. I logs (file dove vengono tracciate le operazioni) devono essere gestiti come flussi di eventi facilmente riportabili per fare analisi.
6. *Admin Processes*: tutte le operazioni di amministrazione dell’applicazione (e quindi tutte le operazioni di configurazione e set up) devono essere fatte in un colpo, altrimenti si rischiano problemi di sincronizzazione.

**Contenuto del repository e descrizione delle attività svolte**

Nel repository indicato sono presenti diversi file, come Requirement, Main, DockerFile, oltre al ReadMe che andiamo a presentare puntualmente per illustrare il lavoro svolto.

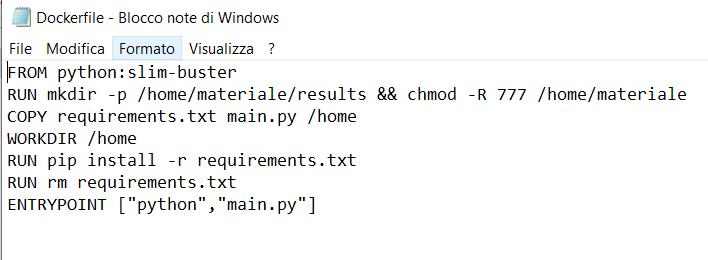


Figura 1 – Dockerfile

Il file Dockerfile è il file che viene richiamato durante la fase di build per la creazione del container. Quando viene richiamato va a costruire il container Docker. Una volta che il container è pronto, si va poi a far partire il progetto in Python. Vediamo cosa fa il Docker file nello specifico osservando le righe contenute nel documento riprodotto nella Figura 1.

Con la prima riga *FROM python: slim-buster* andiamo a creare una immagine docker a partire da slim-buster. Si tratta di una copia di una macchina slim-buster che permette di accedere ad un sistema operativo ubuntu. La macchina è però al momento è vuota e dobbiamo inserire nel container tutti gli elementi necessari per far partire il nostro programma di analisi

Con la seconda riga RUN mkdir -p /home/materiale/results && chmod -R 777 /home/materiale

Andiamo a creare con il comando con mkdir sul Docker le due directory seguenti: home/materiale (con sottodirectory “results”), e home/materiale/

Variabili d’ambiente

Dockerfile

Requirements

