A.D. 1308
unipg
DIPARTIMENTO
DI INGEGNERIA

Presentazione Progetto di

Virtual Networks and Cloud Computing

Corso di Laurea in Ingegneria Informatica e Robotica – A.A. 2022-2023

Dipartimento di Ingegneria

docente

Prof. Gianluca Reali

Body Performance Predictor

applicazione a microservizi Docker / Kubernetes

studenti

350400	**Tommaso**	**Martinelli**	tommaso.martinelli@studenti.unipg.it
348158	**Paolo**	**Speziali**	paolo.speziali@studenti.unipg.it

Data ultimo aggiornamento: 15 maggio 2023

0. Indice

1. Introduzione

Per questo progetto è stata sviluppata un'applicazione web per la classificazione delle performance fisiche degli utenti. L'utente inserisce i suoi parametri fisici e riceve in output la qualità della sua performance fisica, indicata da una lettera:

Classe	**Qualità**
A	Ottima
B	Buona
C	Non buona
D	Pessima

La classificazione avviene tramite un modello di machine learning addestrato utilizzando il dataset "Body performance"[metti riferimento].

In aggiunta, l'applicazione consente di inserire nuovi dati di training, che vengono salvati su un database PostgreSQL insieme al dataset originario.

Al fine di garantire la portabilità e semplificare la realizzazione e la gestione dell'applicativo, tutte le sue componenti sono state incapsulate in container Docker. La gestione dei container è stata affidata, nella sua seconda implementazione, all'orchestratore Kubernetes.

2. Container

Di seguito riportiamo la lista dei container che contengono i microservizi utlizzati per costruire l'applicativo e i relativi dettagli implementativi e di interconnessione. Tutti i container sono stati realizzati tramite Dockerfile eccetto per il container del database e di pgAdmin per cui sono state utilizzate immagini prese direttamente da Docker Hub.

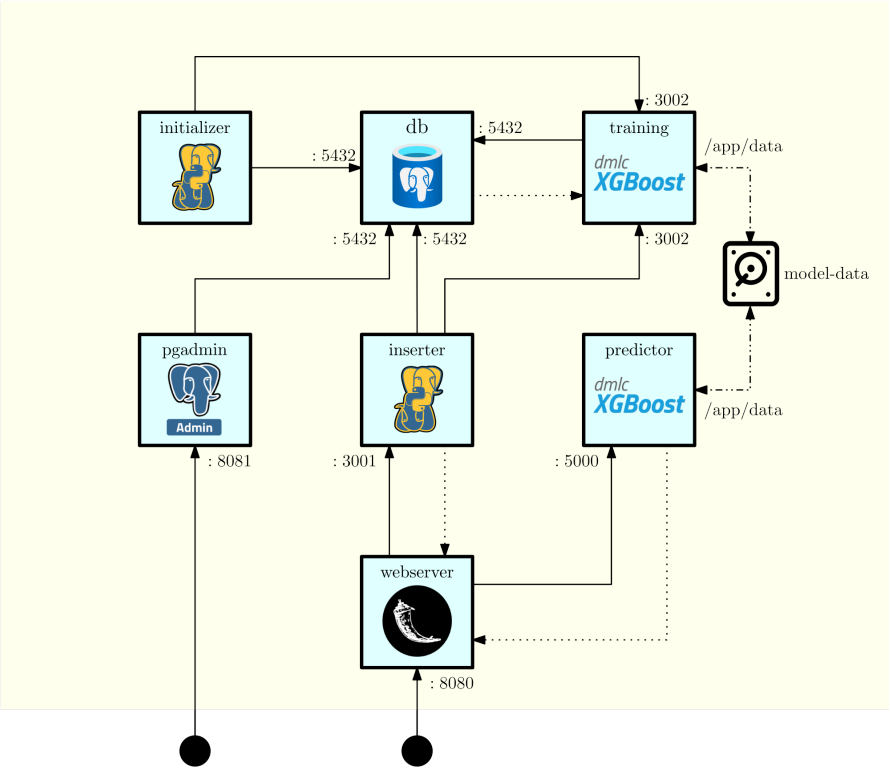

Figura 2.1: Architettura e collegamenti dell'applicazione

2.1 Web Server

Si tratta dell'unico componente che fornisce un'interfaccia grafica per l'interazione con l'utente. Per utilizzare l'applicazione web, gli utenti sono tenuti a inserire i seguenti parametri fisici:

- Età
- Sesso
- Altezza
- Peso
- Percentuale di grasso corporeo
- Pressione sanguigna diastolica
- Pressione sanguigna sistolica
- Sit and bend forward
- Gripforce
- Numero di sit-up
- Broad jump in centimetri

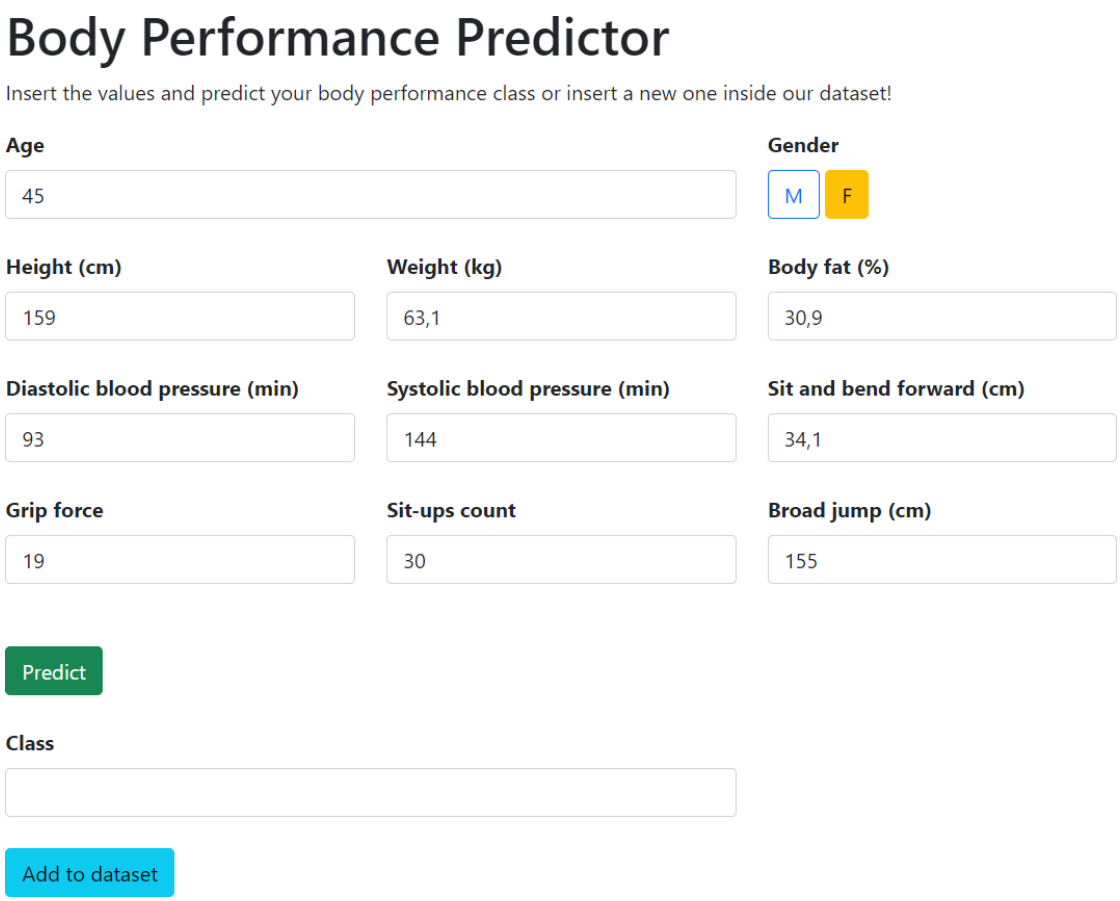

Figura 2.2: Screenshot dell'interfaccia web dell'applicazione

La classe della performance fisica (*A*, *B*, *C* o *D*) viene predetta e mostrata in un campo apposito, in alternativa può essere inserita direttamente dall'utente se è nota e la si desidera caricare nel database ed includerla negli allenamenti futuri del modello.

Il container è costruito su un'immagine di **Python** 3.9 e utilizza il framework **Flask** per gestire le funzionalità di un webserver, tra cui fornire tre endpoint con cui il client può interagire:

- /: chiamata GET che restutisce la pagina HTML.
- /PREDICT: chiamata POST che restutisce la classe predetta dal modello statistico in risposta ai dati inseriti dall'utente inviati in formato JSON.
- /INSERT: chiamata POST che restutisce l'esito dell'inserimento dei dati (classe inclusa) inseriti dall'utente inviati in formato JSON nel database .

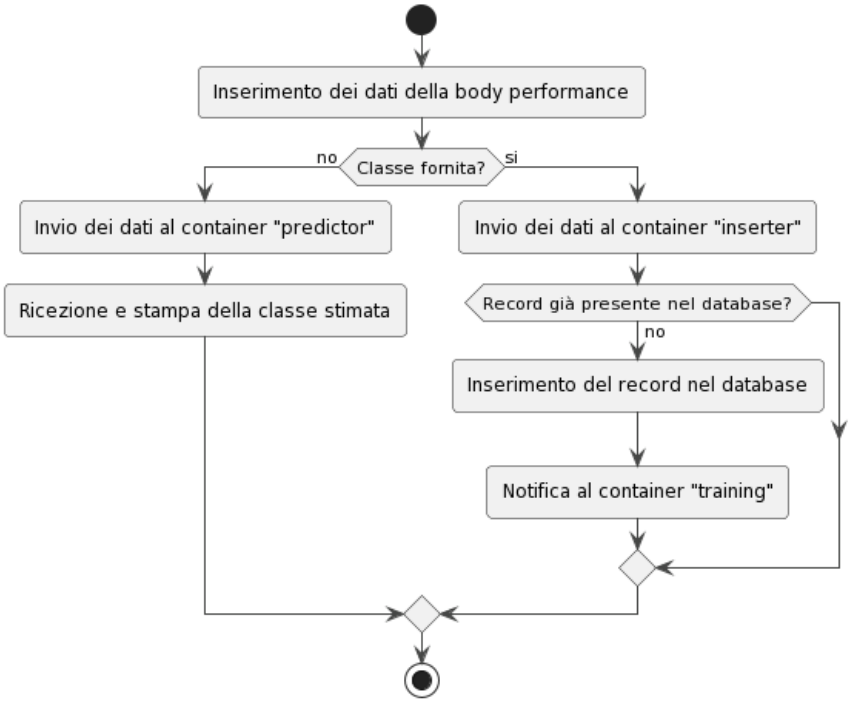

Figura 2.3: Flow chart che mostra l'interazione con la pagina web

2.2 Database

Si è scelto di utilizzare un container **PostgreSQL** per diverse ragioni. Innanzitutto, il dataset utilizzato presenta una struttura fissa e senza valori NaN, si è potuto quindi utilizzare senza problemi un database relazionale come PostgreSQL senza aggiungere ulteriore complessità all'applicativo che l'utilizzo di database NoSQL avrebbero potenzialmente portato.

Inoltre, non si è ritenuto necessario l'utilizzo dello sharding, poiché il volume dei dati e il carico di lavoro previsto non richiedevano una distribuzione su più nodi e l'operazione di training avrebbe comunque richiesto un'operazione di raggruppamento.

Contiene un'unica relazione, denominata **bodyPerformance**, che contiene la totalità del dataset. I record con stessi valori (eccetto la classe) non vengono memorizzati per evitare di influenzare negativamente le performance del modello. È possibile monitorare lo stato del database mediante l'uso di un container **pgAdmin**, accessibile tramite la porta 8081.

2.3 Training

Il container è costruito su un'immagine di **Python** 3.9 e si occupa, traimte uno script Python, dell'addestramento del modello di machine learning, inoltre implementa le funzionalità di un webserver tramite **Flask** per permettere agli altri container di notificarlo nel caso sia necessario un nuovo training. Accetta, per questo scopo, richieste GET sulla porta 3002.

Ad ogni richiesta ricevuta, interroga il database per farsi restituire il dataset ed effettua un nuovo addestramento. Il modello aggiornato è reso disponibile per le predizioni sui dati utente. Come classificatore è stata scelta la libreria **XGBoost**, che permette di ottenere, in tempi molto rapidi, risultati soddisfacenti.

2.4 Predictor

Il container è costruito su un'immagine di **Python** 3.9 e realizza, tramite **Flask**, un server che espone una API REST sulla porta 5000 e riceve chiamate POST dal web-server contenenti i dati inseriti dagli utenti. In seguito alla ricezione dei dati, il predictor esegue una predizione utilizzando il modello addestrato dal container di training. Prima di effettuare la predizione, vengono eseguite opportune trasformazioni sui dati. Il risultato viene infine inviato al web-server.

2.5 Inserter

Il container è costruito su un'immagine di **Python** 3.9 e realizza, tramite **Flask**, un server che espone una API REST sulla porta 3001 e riceve chiamate POST dal web-server contenenti i dati inseriti dagli utenti. Si connette al database e inserisce i dati, l'operazione andrà a buon fine solo se non esiste un altro record con quei valori (anche con classe differente). Al termine, restituisce un messaggio con l'esito e contatta il container di training per avvisarlo che è necessario effettuare un nuovo addestramento dopo l'inserimento dei nuovi dati.

2.6 Initializer

Il container è costruito su un'immagine di **Python** 3.9 ed esegue uno script di inizializzazione che ha il compito di creare, se non presente, la tabella per il dataset nel database e popolarla. Successivamente, notifica il container di training che può avviare l'addestramento.

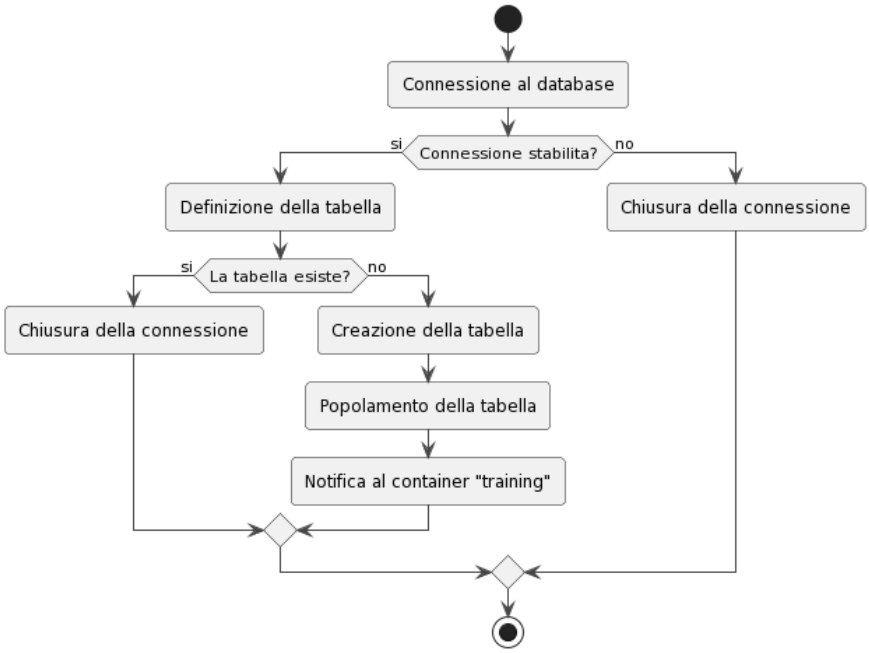

Figura 2.4: Flow chart che mostra il funzionamento dello script di initializer

3. Implementazioni

Sono state realizzate due diverse versioni dell'applicativo utilizzando due diverse modalità di orchestrazione, una con un semplice Docker Compose e l'altra con l'orchestratore Kubernetes.

3.1 Docker Compose

Docker Compose consente di lanciare contemporaneamente tutti i servizi, definendo le loro immagini, porte e altri dettagli necessari per il corretto funzionamento dell'applicazione containerizzata.

In Docker Compose è stato anche possibile definire una rete chiamata *app_network*, che consente ai container di comunicare tra loro. Sono stati utilizzati anche i volumi per garantire la persistenza dei dati. Ad esempio, per il database è stato necessario utilizzare un volume per evitare la perdita di eventuali nuovi dati inseriti dagli utenti e non presenti nel dataset originario Un altro volume è stato utilizzato per condividere i dati tra i container training e predictor, in cui vengono salvati il modello e lo scaler in formato *pickle*. Questo volume garantisce la disponibilità e la coerenza dei dati, poiché è sempre possibile richiedere una predizione. Tuttavia, se tale richiesta viene effettuata durante un addestramento, verrà utilizzato un modello non ancora aggiornato. Viene inoltre caricato nel database, come volume, il file CSV che contiene il dataset originario. L'intera applicazione è testabile grazie a un file `init.sh`, che contiene una serie di comandi per creare e avviare i container tramite il Docker Compose.

3.2 Kubernetes

La versione più recente dell'applicazione è stata orchestrata con Kubernetes, uno strumento più completo e complesso rispetto a Docker Compose. Sono stati creati oggetti Deployment, uno per ogni container Docker, ad eccezione di initializer, che è stato realizzato con un Job poiché deve essere eseguito solo una volta e non deve essere riavviato. Sono stati definiti anche dei Service, uno per ogni Deployment, per consentire la connettività tra Pod nella rete interna di Kubernetes e per esporre le porte 8080 (webserver) e 8081 (pgAdmin) verso l'esterno tramite il port forwarding di Kubernetes. In Kubernetes sono stati utilizzati i *PersistentVolumeClaim* per gestire i volumi, il volume che contiene il file CSV del dataset è stato incluso tramite un *config-map*. Tutti gli oggetti Kubernetes descritti sono stati definiti utilizzando file YAML. L'intera applicazione è testabile grazie a un file `init-k.sh`, che contiene una serie di comandi per creare e configurare gli oggetti Kubernetes utilizzando i loro file ed esegue le operazioni di port forwarding.