Laboratorio di Ingegneria Informatica
Relazione sull'esonero 2024/25

Gabriele Cappellaro
cappellaro.2044279@studenti.uniroma1.it

1 Obiettivo del progetto

L'obiettivo di questo progetto è poter fare delle query SQL su un database e aggiungervi dati, anche tramite interfaccia grafica; tutto questo basandosi sull'architettura client-server di FastAPI e sulla containerizzazione in Docker.

2 Struttura del progetto

Il progetto è suddiviso in tre sezioni principali, ciascuna corrispondente a un servizio Docker separato:

- **frontend** [8003:8000]: serve una pagina HTML per interagire il backend tramite interfaccia grafica
- **backend** [8001:8000]: espone gli endpoint per poter consultare e manipolare la base di dati
- **database** [3307:3306]: esegue un'immagine di MariaDB

Ognuna di queste sezioni è eseguita all'interno di un container Docker dedicato, in modo da isolare le funzionalità e facilitare la gestione e la scalabilità del progetto.

L'infrastruttura si basa su Docker Compose, che consente di orchestrare i tre container, mantenendo attiva la comunicazione tra di essi.

Inoltre sono stati creati due ambienti Conda separati per backend e frontend, così da gestire facilmente le dipendenze di ciascun pacchetto nel rispettivo file `requirements.txt` da far usare poi a Docker.

3 Organizzazione della base di dati

Il database relazionale è stato organizzato con l'obiettivo di avere una risorsa scalabile e senza ridondanza di dati; pertanto, sono state create quattro tabelle:

- **directors** [director_id, name, age]
  PK(director_id)
  UNIQUE(name)
- **movies** [movie_id, name, release_year, genre, director_id]
  PK(movie_id)
  UNIQUE(name, release_year)
  FK(director_id) $\rightarrow$ directors(director_id)
- **platforms** [platform_id, name]
  PK(platform_id)
- **watch_on** [movie_id, platform_id]
  PK(movie_id, platform_id)
  FK(movie_id) $\rightarrow$ movies(movie_id)
  FK(platform_id) $\rightarrow$ platforms(platform_id)

Si è scelto di indicizzare tramite un ID univoco ogni tabella per avere una maggiore scalabilità ed efficienza del database.

Inoltre è stato creato un utente SQL `py` con password `esonero` e con privilegi minimi, esclusivamente sul database `movies_db`.

4 Organizzazione del codice

Nella directory del progetto abbiamo le due cartelle per il backend e per il frontend, che analizzeremo nel dettaglio in seguito, il file `docker-compose.yaml` e due cartelle necessarie a MariaDB, `mariadb_data` e `mariadb_init`.

Il file `docker-compose.yaml` definisce le proprietà dei container che ospiteranno i vari servizi, come il Dockerfile (o un'immagine già esistente) da utilizzare, l'associazione delle porte esterne e interne, le variabili di ambiente e i volumi da associare.

I volumi sono molto comodi per garantire la persistenza dei dati del database anche al riavvio del container; per questo, la cartella

`mariadb_data` è stata associata a quella del container `/var/lib/mysql` e in modo simile per la cartella `mariadb_init`, contenente il file `init.sql` che crea il database, è stata abbinata con la `/docker-entrypoint-initdb.d`.

4.1 Backend

In questa cartella abbiamo il Dockerfile e la cartella `src` in cui ci sono i vari sotto-pacchetti Python.

Nel Dockerfile definiamo che useremo un'immagine specifica adatta ad eseguire Python 3.12, successivamente installiamo delle dipendenze per poter comunicare con MariaDB attraverso Python, installiamo tutte le librerie Python necessarie, copiamo i file nel container e infine eseguiamo il server ASGI `uvicorn`.

Analizziamo ora i vari moduli `.py` del pacchetto:

- **backend.py**: si occupa di creare la classe FastAPI e aggiungere i vari `APIRouter` degli endpoint,
- **db/mariadb.py**: fornisce le funzioni per eseguire una query sul database e per aggiungervi dei dati,
- **endpoints/add/add.py**: crea l'endpoint `/add` per gestire la richiesta POST che prende in input una stringa CSV e ritorna in output la conferma dell'aggiunta delle informazioni al DB. Inoltre ha una semplice funzione per convertire la stringa CSV in un dizionario, così da validarla con un modello Pydantic,
- **endpoints/schema/schema.py**: crea l'endpoint `/schema_summary` che non prende nulla in input e ritorna in output la lista delle tabella del DB e per ogni tabella mostra le sue colonne,
- **endpoints/nl2sql.py**: una classe che, come si evince dal nome, si occupa di "tradurre" un stringa (tra quelle fornite) dal linguaggio naturale in una query SQL,
- **endpoints/search.py**: crea l'endpoint `/search` che prende in input una stringa in linguaggio naturale e restituisce i risultati della rispettiva query SQL.

Per brevità, sono stati omessi alcuni moduli denominati `models.py`, che contengono dei modelli Pydantic per serializzare e validare le varie strutture dati necessarie al progetto.

4.2 Frontend

Qui abbiamo il Dockerfile che svolge gli stessi identici compiti di quello del backend, ma in più copia nel container anche i file delle cartelle `templates` e `static`, contenenti i file per il funzionamento della pagina web attraverso Jinja2.

Analizziamo anche qui le responsabilità dei moduli presenti:

- **frontend.py**: crea i vari endpoints del sito; anche qui abbiamo gli stessi del backend e la root `/` che mostra la pagina principale,
- **api/api.py**: una classe con dei metodi per comunicare con le API REST del backend,
- **api/models.py**: un semplice modello Pydantic per la struttura dati che riceve i dati dal backend,
- **static/style.css**: un file di stile CSS per la pagina web,
- **templates/index.html**: il file HTML per il sito, contenente le parti che Jinja2 va a "riempire" con i dati da mostrare nella pagina.