Report del Progetto “Film Database” – IUM-TWEB

Studente: *[Inserire Nome]*

Introduzione

Il progetto sviluppato ha l’obiettivo di fornire a fan e giornalisti un portale per l’esplorazione di dati cinematografici, offrendo ricerche su attori, film, premi, recensioni e statistiche. L’architettura è distribuita: un server centrale smista le richieste verso microservizi implementati in Express e Java Spring Boot, collegati a database distinti (MongoDB e PostgreSQL). Il frontend è stato sviluppato in HTML, CSS e JavaScript con l’uso di Handlebars.js. Il sistema include anche una chat real-time su stanze tematiche tramite Socket.io. In questo report analizziamo tutte le componenti tecniche implementate, seguendo la struttura richiesta.

1. Interfaccia utente (HTML, CSS, JS, Handlebars)

Soluzione e motivazioni progettuali

L’interfaccia è realizzata con HTML5, CSS3 e JavaScript vanilla. Si utilizzano i template Handlebars per generare dinamicamente contenuti HTML partendo dai dati JSON restituiti dalle API. Le richieste verso il backend sono gestite da Axios. L’obiettivo è separare presentazione e logica, facilitando riuso e leggibilità del codice. I template includono componenti riutilizzabili come schede film, intestazioni e sezioni dettagliate.

Problemi affrontati

L’interfaccia doveva essere dinamica e accessibile. Sono stati gestiti i focus per tastiera, i tag ARIA e la responsività CSS. Per i contenuti aggiornabili in tempo reale (es. recensioni), è stato necessario integrare meccanismi asincroni per aggiornare il DOM senza ricaricare la pagina.

Requisiti soddisfatti

È possibile cercare film o attori, ottenere dettagli, recensioni e visualizzare grafici. L’interfaccia è integrata con le API, è accessibile e reattiva.

Limitazioni ed estendibilità

Il caricamento dinamico è totalmente lato client, quindi soggetto a latenze. Si potrebbe aggiungere caching client-side per dati statici.

2. Server centrale (Express.js)

Soluzione e motivazioni progettuali

Il server centrale è un’app Express.js che funge da API Gateway: riceve richieste dal frontend e le inoltra ai microservizi Postgres (Java) o Mongo (Express). Usa Axios per invocare i servizi e instrada le risposte al frontend. L’uso di Node.js consente alta concorrenza grazie all’I/O non bloccante.

Problemi affrontati

Evitare carichi computazionali nel gateway è stato cruciale. Abbiamo delegato la logica pesante ai microservizi. L'aggregazione di risposte multiple è stata gestita con `Promise.all`, ottimizzando i tempi di risposta.

Requisiti soddisfatti

Tutte le rotte richieste sono esposte e intermediano le chiamate senza eseguire elaborazioni pesanti. Il server centrale è pronto per essere scalato orizzontalmente.

Limitazioni ed estendibilità

Attualmente non gestisce versioning o rate limiting. Potrebbe essere integrato un bilanciatore e strumenti come Consul o Docker per migliorare la scalabilità.

3. Microservizio PostgreSQL (Spring Boot)

Soluzione e motivazioni progettuali

Spring Boot è stato scelto per la rapidità di sviluppo e la struttura REST pronta all'uso. Il servizio espone controller che gestiscono film, attori, crew, premi e altro. Le entità sono modellate con JPA e le query complesse sono gestite con repository personalizzati.

Problemi affrontati

La modellazione delle relazioni tra entità (film-attori-premi) ha richiesto attenzione. Abbiamo ottimizzato le fetch strategies e normalizzato lo schema relazionale.

Requisiti soddisfatti

Gestisce tutti i dati statici richiesti: film, attori, premi Oscar, ecc. Espone rotte REST compatibili con il gateway.

Limitazioni ed estendibilità

Le modifiche di schema richiedono migrazioni. Lo scaling di PostgreSQL è più complesso rispetto a database NoSQL.

4. Microservizio MongoDB (Express.js)

Soluzione e motivazioni progettuali

Un servizio Express.js gestisce le recensioni, i poster e le release usando MongoDB. Le collezioni sono schemaless e permettono inserimenti flessibili. Il framework di aggregazione Mongo è usato per statistiche come le medie dei voti.

Problemi affrontati

Le review devono essere collegate ai titoli dei film e non agli ID. Questo ha richiesto naming coerente e disambiguazione manuale. Inoltre, la gestione degli ID per poster e release è stata centralizzata nel microservizio.

Requisiti soddisfatti

Tutti i dati dinamici (recensioni, poster, release) sono gestiti in Mongo, in linea con il requisito di separazione statica/dinamica dei dati.

Limitazioni ed estendibilità

Non supporta ACID completo. Si potrebbe in futuro aggiungere sharding o caching distribuito.

5. Sistema di chat (Socket.io)

Soluzione e motivazioni progettuali

Abbiamo usato Socket.io per creare una chat real-time con stanze tematiche (film, attori). Gli utenti entrano in una stanza basata sull'URL o su selezione. I messaggi sono trasmessi via WebSocket e gestiti lato server in tempo reale.

Problemi affrontati

La gestione delle stanze e degli eventi concorrenti ha richiesto validazione degli utenti e gestione dello stato minimale. È stato anche curato l'aspetto grafico e accessibile della chat.

Requisiti soddisfatti

Permette chat tra fan ed esperti su film o attori specifici. È integrata con la UI senza impattare il flusso principale.

Limitazioni ed estendibilità

Attualmente la chat è pubblica e non persistente. Potrebbe essere estesa con messaggi privati o moderazione.

6. Popolamento e analisi dati (Jupyter + Python)

Soluzione e motivazioni progettuali

Abbiamo usato notebook Jupyter per caricare i dataset nei database e per fare analisi statistiche preliminari. I dati sono letti da CSV e inseriti in PostgreSQL via `psycopg2` e in Mongo via `pymongo`.

Problemi affrontati

La normalizzazione dei dati e la verifica delle dipendenze (attori, film, premi) è stata cruciale. L'uso di Jupyter ha facilitato il debugging.

Requisiti soddisfatti

I database sono stati popolati con dati realistici e coerenti, rendendo la navigazione significativa per l'utente.

Limitazioni ed estendibilità

Il processo è manuale. Potrebbe essere automatizzato con uno scheduler (cron, Airflow).

7. Interfacciamento frontend-server (api.js)

Soluzione e motivazioni progettuali

Il modulo `api.js` incapsula tutte le chiamate asincrone (via Axios e fetch) verso il backend. Le funzioni sono organizzate per entità e permettono di mantenere il frontend pulito.

Problemi affrontati

Mantenere coerenza tra route e moduli, e gestire errori di rete. Alcune funzioni aggregano risposte da più microservizi in un singolo oggetto frontend-ready.

Requisiti soddisfatti

L'interfaccia gestisce chiamate parallele e asincrone, semplificando il lavoro del frontend.

Limitazioni ed estendibilità

Mix tra fetch e Axios non uniforme. Possibile introdurre caching o service layer.

8. Pagina attore e interazione asincrona (actor.js)

Soluzione e motivazioni progettuali

La pagina attore consente di cercare un attore, visualizzare filmografia, Oscar, paesi d'uscita e statistiche. Ogni film è cliccabile e apre un modale con dettagli recuperati asincronamente.

Problemi affrontati

La filmografia è un array annidato da processare. Il modulo deve combinare dati da più fonti (Postgres, Mongo) e costruire un'esperienza fluida.

Requisiti soddisfatti

La pagina consente l'esplorazione approfondita dei dati dell'attore. Integra tutto il sistema backend e garantisce un'esperienza UX/UI avanzata.

Limitazioni ed estendibilità

Le funzioni di impaginazione sono ripetute. Si potrebbe astrarre in componenti riutilizzabili. In futuro si può integrare con un router client per rendere le ricerche condivisibili via URL.

Conclusioni

Il progetto ha realizzato tutti i requisiti previsti, implementando un'infrastruttura distribuita efficiente e scalabile. Ogni componente è stato sviluppato con attenzione a prestazioni, modularità e accessibilità. L'esperienza ha permesso di apprendere best practice nella progettazione di sistemi full stack moderni.

Divisione del lavoro

Il progetto è stato realizzato interamente dal sottoscritto. Ho curato lo sviluppo backend e frontend, la configurazione dei server, la gestione dei dati e l'integrazione delle tecnologie. Mi sono avvalso, quando necessario, di strumenti AI per suggerimenti o debugging, ma tutte le scelte tecniche sono state effettuate in autonomia.

Informazioni aggiuntive

Per eseguire il progetto è sufficiente installare Node.js e Java JDK 17. I database usano configurazioni locali standard. Tutti i server si avviano con `npm start` o `mvn spring-boot:run`.

Bibliografia

- MongoDB vs PostgreSQL: Choosing the Right Database – SprinkleData Blog
- Express.js and MongoDB REST API Tutorial – MongoDB Docs
- Axios – GitHub Repository
- Spring Boot for Microservices – GeeksforGeeks
- Socket.io Chat Tutorial – GeeksforGeeks
- Postgres & Jupyter Integration – Analytics Vidhya
- Mastering the API Gateway – Medium
- Handlebar Template Engines – Medium