Introduzione

Per il progetto abbiamo realizzato una piattaforma per il mondo cinematografico, **Cineverse**.

La piattaforma fornisce principalmente due macro servizi: un sistema di ricerca e consultazione di dati relativi al mondo del cinema e la possibilità di chattare con altri utenti in apposite aree tematiche. Il sito prevede un sistema di registrazione, di accesso, e di richiesta di ruoli giornalistici diversi da quelli di un utente normale. Con il login si ha accesso alla sezione della chat e alla possibilità di lasciare recensioni; questo vincolo è stato posto per permettere solamente agli utenti identificabili di scrivere messaggi. I dati cinematografici rappresentano:

- **Statistiche e dati generali sui film**
- **Statistiche e informazioni su attor**i
- **Gli studios e le crew che hanno preso parte alla realizzazione di un film**(registi, compositori, sceneggiatori).
- **Recensioni**
- **Un sistema di chat in stanza relative a film o attori**

Il sistema di autenticazione prevede una **gestione della sessione tramite token**, valida per l'intera durata della navigazione. Tale sessione consente l'accesso a funzionalità come la chat tematica, disponibile per tutti gli utenti registrati. L'abilitazione alla scrittura è vincolata all'identità utente, al fine di garantire la moderazione e la tracciabilità delle conversazioni. Alcune funzionalità amministrative (moderazione messaggi, accesso a sezioni nascoste) sono riservate a ruoli utente con privilegi specifici (moderatore o giornalista).

Il sistema offre diverse pagine per l'esplorazione dei contenuti e delle loro informazioni:

1. **Home:** dove trovare gli ultimi vincitori di oscar, le ultime uscite e qualche film top rated
2. **Top movies**: dove navigare tra 200 films top rated e ordinarli, filtrarli per genere
3. **Top Actors**: dove navigare tra i 500 top actors per numero film e rating medio, ordinarli e filtrarli per generi recitati
4. **Sistema di ricerca di attori e movies da searchbar**

Technical Tasks

1: Meccanismo di routes per la richiesta di dati, pagine e gestione degli errori

-Soluzione:
Seguendo le direttive apprese al momento della consegna e quanto spiegato durante il modulo abbiamo cercato di mantenere il main server il più leggero possibile.
Il **Mainserver**, l'addetto alla rappresentazione dei dati restituiti dalla comunicazione con gli altri due server attraverso le view (*page*.hbs) e all'intercettazione delle richieste dell'utente di pagine e dati, identificati tramite id o altri modi in base al database di riferimento, tramite un meccanismo di *routes*, di controllo dati ed errori.
Se l'utente ricerca una pagina non esistente l'errore è gestito dal *MainserverFile* caricando ad una pagina *NotFound*:

Se invece l'utente cerca una pagina esistente ma passando un *"fake"* id tramite url il controllo dell'errore avverrà su più livelli: il catch finale nella route permette, appunto, di catturare l'errore generato dalla richiesta di dati del server gestore, caricherà la stessa pagina *NotFound* con un titolo adatto all'errore (per esempio con un *"fake"* id per route movies *Movie not found*): questa catch viene triggerata da un errore generato in primis dal controller java designato chiamato dalla richiesta get della route. I dati richiesti dai controller del Javaserver vengono, dove vi è incertezza dell'esistenza di un risultato, trattati come *Optional* restituendo quindi 404 alla richiesta *axios* se il DB ritorna dati vuoti per una ricerca che non ha avuto riscontri.

Invece gli errori derivati dalle funzioni dinamiche del sito come chat, recensioni e registrazioni utente vengono gestiti in diversi modi dal Dataserver che restituisce codici di errore standardizzati 201o200 se la richiesta di registrazione è andata a buon fine, 400/401se la richiesta non è completa o non si è autorizzati, 404 se il dato cercato non è presente e 500 se è un errore del server. In base all'errore restituito il Mainserver si comporterà e lo gestirà diversamente.
.

-Requisiti: Riteniamo di aver soddisfatto tutti i requisiti dal momento che l'utente è in grado di esplorare liberamente richiedendo i dati scelti facilmente senza terminare in zone non gestite del sito

-**Limitazioni:** Ogni rotta possiede blocchi try catch e gestione degli errori ripetuta, come richieste api differenti per piccole differenze di dati (un campo transient presente o meno in una richiesta al server Springboot) portando quindi a ripetitività . Inoltre il Mainserver è totalmente dipendente dal Backend per sapere se l'utente sta richiedendo dati inesistenti senza avere nessun filtro.

- **2: Gestione, restituzione e divisione delle responsabilità dei dati**

  -Soluzione: I dati statici (come film, attori, premi Oscar, ecc.) sono stati modellati in PostgreSQL utilizzando relazioni tramite **chiavi esterne**, con la tabella "movies" come nodo centrale a cui la maggior parte delle altre tabelle è collegata. Un'eccezione è rappresentata dalla tabella degli attori, in cui è stato utilizzato un identificatore interno generato automaticamente, anziché una chiave esterna verso "movies".
  Oltre alle tabelle derivate direttamente dai file CSV forniti, è stata aggiunta una sola tabella supplementare: actor summaries, descritta al punto 3 del report.

  Il server Springboot "**Javaserver**" (porta 8080)sulla porta responsabile della restituzione dei dati dal DB PostgreSQL al Mainserver opera attraverso 3 controller (Movies, Actors e Crew) di cui solo i primi due si occupano della restituzione della gran parte dei dati richiesti dalle pagine: tutte le info relative ad un film (Movies Controller esempio immagine sopra:)

Crew Controller viene utilizzato per la popolazione delle informazioni all'interno della pagina movie singolo e così *Actors Controller* sfrutta query JPQL senza join complessi appoggiandosi ai modelli delle rispettive tabelle.
Difatti I tre controller e i relativi servizi operano su modelli JPA che rispecchiano quasi perfettamente le tabelle del database.
Il *model Movie*: è stato adattato per essere riutilizzato in contesti diversi, accorpando in un'unica classe anche le informazioni provenienti da tabelle esterne a cui è collegato. Questi campi aggiuntivi, oggetti di tipo corrispondente al model della "sua tabella esterna" sono annotati con @Transient poiché non devono essere persistenti per ogni richiesta. Ad esempio, nel mostrare la lista dei film in cui ha recitato un attore, non è necessario includere le lingue o i paesi di distribuzione: l'uso di @Transient consente quindi di mantenere il model Movie flessibile e riutilizzabile in più contesti senza obbligare la popolazione completa di ogni campo.
Il *model Actor* dotato di @Transient come Movie ha in aggiunta, rispetto alla tabella che rappresenta, la lista dei ruoli e l'average rating dei film in cui ha recitato (sfruttati per la pagina top Actors).
L'unico model non rappresentazione diretta di una tabella ma nato per necessità di rappresentare l'unione tra le tabelle actors e movies

Il server express "**DataServer**" (porta 3001) è il responsabile per la gestione dei dati dinamici dell'applicazione quali: recensioni, chat, messaggi e utenti.

Il collegamento a MongoDB è gestito centralmente tramite mongoose, e ogni tipo di informazione è modellata attraverso **collezioni dedicate** sfruttando per la collection chat(vedi punto 4) la funzionalità di mongoDB per avere strutture flessibili. I dati restituiti dalle API REST sono formattati direttamente in JSON, e sfruttano .populate() per integrare documenti referenziati, ad esempio i messaggi in una chat e il sender tramite ObjectIds.

-Problemi: Un problema significativo emerso durante la fase di integrazione riguarda la difficoltà nel collegare in modo univoco i dati provenienti da **fonti diverse**, per esempio, tra attori e premi Oscar. E La mancanza di identificatori univoci condivisi nei file CSV di origine ha reso impossibile discriminare tra **attori omonimi**, poiché la tabella actors è stata costruita a partire da un file che "descrive le recitazioni", e non da un'entità che rappresenta l'attore come soggetto.

Di conseguenza: non è stato possibile associare con certezza un premio Oscar a un attore specifico se il nome era condiviso con altri; l'elenco dei film in cui un attore ha recitato è stato costruito filtrando per name nelle recitazioni, con il rischio di includere film di **omonimi** non legati all'attore ricercato limitando l'affidabilità di alcune visualizzazioni.

-Requisiti: L'architettura richiesta prevedeva dati dinamici gestiti su MongoDB sfruttando un server express diverso dal Mainserver presentando API REST accessibili per offrire persistenza e consultazioni rapide per dati in rapido cambiamento. E di un server Springboot per la gestione dei dati statici importati da file csv e trattati su un DB PostgreSQL tramite le funzionalità offerte da springboot come jpa con la creazione di api rest tramite controllers, services e models. Pertanto i requisiti sono stati rispettati nonostante alcuni problemi di "good practices".

-Limitazioni: L'utilizzo eccessivo dei campi @Transient pur utile alla flessibilità può portare a inconsistenze nei dati in quanto non sempre tutti i campi sono popolati e non vi è un controllo, seppur per il momento non utile, centralizzato.

- **3: Gestione dei tempi di caricamento e ottimizzazione**

  -Soluzione: La divisione delle richieste tra server e API dedicate consente di restituire, per ogni pagina, solo i dati essenziali. Il database PostgreSQL, che gestisce la mole maggiore di dati, opera su tabelle indicizzate sui campi sfruttati per le ricerche con query limitate e filtrate, garantendo risposte rapide per le pagine movie e actor singoli. L'uso dei campi @Transient permette di evitare il caricamento inutile di dati non necessari.

  Per ottimizzare il caricamento della pagina **Home**, che necessita di dati costosi da ottenere (come i film premiati con Oscar, che richiedono join complessi o ricerche su larga scala), è stata introdotta una **cache temporanea** che memorizza i risultati ottenuti alla prima richiesta per un intervallo di tempo definito(5 min.) migliorando significativamente i tempi di risposta.

  Invece per le pagine movies e actors i risultati sono stati limitati ai 500 attori e 300 films; le operazioni di ordinamento e filtraggio per genere, o il caricamento di più dati, non porta ad una nuova richiesta al JavaServer, ma vengono richiesti una volta sola e gestiti solo visualmente, in quanto non renderebbe più fluida l'esperienza durante l'applicazione di diversi filtri e abbiamo notato che con questi numeri si riusciva ad avere un buon numero di dati per essere pagine "generali" non per una ricerca precisa senza avere caricamenti troppo lunghi.

  -Problemi: La cache della pagina home non diminuisce il primo caricamento della home che è ancora molto lungo, in un'applicazione "realistica" questa cache sarebbe distribuita e persistente almeno in alcune sue parti.

  -Requisiti: La gestione per l'ottimizzazione del caricamento delle pagine, quindi richiesta dei dati permette all'utente di esplorare con facilità e in diversi modi i dati, pertanto riteniamo di aver soddisfatto le richieste.

  -Limitazioni: La limitazione delle query per il caricamento delle pagine movie e actors, a 500 e 300 rispettivamente potrebbero limitare le potenzialità di due pagine che potrebbero permettere di ricercare su tutti gli attori/film presenti.

- **4: Gestione del sistema delle chat e delle recensioni**

  -Soluzione: Il progetto gestisce chat in tempo reale e recensioni utente tramite una struttura a due server: il MainServer gestisce l'interfaccia e le connessioni WebSocket, mentre il DataServer gestisce la logica applicativa e la persistenza dei dati. La chat usa socket.io: i client si connettono attraverso socketManager a cui inviano eventi, lui inoltra al DataServer per la validazione, il salvataggio in MongoDB e si occuperà dell'inoltro agli altri utenti che sono dentro la chat gestiti in stanze identificate con l'id e tipo della chat. L'autenticazione avviene tramite token JWT per validare gli eventi sia nelle connessioni socket sia nelle API. Le recensioni sono gestite via API REST: ogni utente autenticato può inviare una recensione per film, modificabile e collegata sia all'utente sia al film nel database. Le recensioni importate sono state normalizzate con quelle create dal sito a nome di campi, rappresentazione ma differiscono per gestione di collegamento al film.

  -Problemi: Il sistema di moderazione recensioni e chat per utenti master è macchinoso e "abbozzato". In caso di importazione di altre reviews da un altra fonte di dati ci sarebbero problemi di omonimia tra identificatori per le reviews in quanto solo quelle create dal sito hanno un identificatore del film uguale al database postgreSQL, le importate usano il title.

  -Requisiti: Il sistema permette la comunicazione in tempo reale tra utenti in stanze basate su attori o film come da consegna e la possibilità di recensire film pertanto riteniamo di aver soddisfatto le richieste.

  -Limitazioni: La chat non è in grado di allegare messaggi multimediali o di rispondere ad un messaggio preciso, inoltre è mancante di un sistema di notifiche per nuovi messaggi della chat. E le recensioni non hanno un sistema di ordinamento preciso.

- **5: Gestione degli utenti e dei loro ruoli**

  -Soluzione*:* Il sistema di gestione utenti prevede due componenti principali: il **login con sessione autenticata tramite JWT** e la **gestione dei ruoli**, inclusa la promozione al ruolo "journalist".

  L'autenticazione avviene tramite l'invio di username e password che il server confronta con le credenziali memorizzate nel database (password salvate in forma hashata tramite bcrypt). Se il login ha successo, viene

generato un token JWT contenente l'identificativo dell'utente e il suo ruolo. Questo token viene poi associato alla sessione Express avviata dal MainServer e conservato tramite cookie: ogni richiesta successiva può quindi essere autenticata e autorizzata in modo stateless tramite il token. In questo modo, l'utente può accedere a funzionalità riservate (es. scrivere in chat, inviare recensioni), mentre l'identità rimane verificabile.

Gli utenti registrati con ruolo base hanno la possibilità di inviare una **richiesta per diventare "journalist"**, operazione eseguibile da interfaccia tramite apposito pulsante. La richiesta viene registrata sul database come "in attesa" e rimane visibile all'utente con ruolo "master", che funge da amministratore della piattaforma.

Il master può visualizzare tutte le richieste pendenti in una pagina dedicata e approvarle o rifiutarle. L'esito della richiesta viene notificato in tempo reale all'utente richiedente tramite **WebSocket**, permettendo aggiornamenti istantanei della UI (es. badge "Journalist" accanto al nome o messaggi di conferma).

Il ruolo "giornalista" abilita funzionalità aggiuntive, come la possibilità di scrivere recensioni, mentre in chat il ruolo è evidenziato da un badge grafico che distingue l'utente dagli altri. Questo meccanismo consente una gestione semplice, ma efficace, dei permessi e delle gerarchie all'interno della piattaforma.

-Problemi*:* Durante l'implementazione si è reso necessario mantenere la coerenza tra sessione, JWT e WebSocket, soprattutto nei casi di aggiornamento in tempo reale del ruolo utente. La gestione della sessione e dei token ha richiesto particolare attenzione per evitare condizioni in cui l'interfaccia mostrasse informazioni obsolete o incoerenti rispetto al ruolo effettivo. Inoltre, l'invio in tempo reale di aggiornamenti all'utente (es. cambio ruolo) tramite WebSocket ha introdotto complessità aggiuntive nella sincronizzazione client-server.

-Requisiti: Il sistema soddisfa i requisiti richiesti per la gestione degli utenti, ovvero:

- Autenticazione sicura con salvataggio delle password in hash.
- Uso di JWT per autenticazione stateless tra i microservizi.
- Sessione Express per persistenza locale del login nel MainServer.
- Meccanismo per la richiesta di ruolo "giornalista".
- Interfaccia di approvazione per utenti master.
- Notifiche in tempo reale tramite WebSocket.

Queste funzionalità garantiscono una gestione completa e sicura degli accessi, dei ruoli e delle relative autorizzazioni.

-Limitazioni: Il sistema, pur funzionante, presenta alcune limitazioni:

- Non esiste un sistema di notifica persistente (cronologia o archivio delle notifiche inviate via WebSocket).
- Il master ha pieno potere decisionale senza alcun filtro o log degli eventi, riducendo la tracciabilità delle azioni amministrative.
- Le funzionalità per i diversi ruoli sono gestite a livello applicativo e non tramite middleware o policy centralizzate, con il rischio di duplicazioni logiche nella gestione dei permessi.
- Le richieste di promozione non hanno un sistema di timeout o scadenza automatica: restano in sospeso finché il master non le gestisce manualmente.

Jupyter Notebooks

-Soluzione: Abbiamo deciso di implementare 4 differenti notebook:
-genre_relations: per analizzare correlazioni nel tempo tra genere e oscars e rating dei film

-review_analisis: analisi sul dataset cdv delle reviews di rotten tomato
-rating_analysis: analisi dell'impatto che hanno diversi fattori (diversità linguistica, mese di uscita, colonne sonore, temi sociale e politici) sul rating dei film.
-notebook_traversa: distribuzione geografica attori e film, analisi su carriera attori, registi e studi

-Problematiche: Le query dei notebook, escluso review analysis operano su db quindi non dispongono di un modulo per la pulizia dei dati in quanto è già avvenuta al caricamento dei dati durante le prime fasi del progetto

-Requisiti: Riteniamo di aver soddisfatto le richieste della consegna in quanto per ogni notebook si hanno le statistiche che permettono l'analisi dei dati cinematografici sotto diversi aspetti visualizzabili tramite diversi grafici.

-Limitazioni: Le query di actor_crew_relations riguardanti alle crew non si sono potute fare "più complete" per la durata di ogni query(10+min) rendendole molto complicate da debuggare e hanno portato ad una semplificazione eliminando la divisione in generi rispetto alle query di studio sugli attori vincitori di oscar. Inoltre la "semplicità" della maggior parte dei dati ha impedito l'utilizzo di grafici più articolati.

Conclusioni

Durante la realizzazione di questo progetto ci siamo resi conto di quanto un progetto così articolato e composto da cosi tante parti debba essere organizzato con precisione e curato in ogni sua parte dato che, nonostante la struttura comune dei vari componenti, ognuno di questi ha caratteristiche e problematiche proprie portando così a diverse iterazioni sugli stessi.

Divisione del lavoro

Abbiamo svolto il progetto lavorando insieme quasi sempre insieme in chiamata o aggiornandoci costantemente via chat.
Le uniche parti che abbiamo realizzato interamente separatamente sono stati i Jupyter Notebook.
Nonostante un focus iniziale diviso tra i membri per delle pagine specifiche, non ci sono state intere parti o pagine scritte interamente da una persona singola in quanto spesso capitava di dover correggere o dover modificare "moduli"(routes, services o views) scritte da un altro membro per adattarli alle aggiunte, concluderle o correggerle.

Informazioni extra

Come si può notare dai vari commit spesso abbiamo avuto problemi per la sincronizzazione dei diversi commit e specialmente dei loro merge, spesso causati (inizialmente) da versioni di pacchetti e/o compilatori diversi o di rimozioni accidentali durante conflitti di codice.

Bibliografia

Per lo svolgimento del compito abbiamo consultato diversi siti tra i quali https://www.geeksforgeeks.org per springboot; https://www.w3schools.com specialmente per pandas e Mongo. Intelligenze artificiali sono state sfruttate per CSS, animazioni JavaScript (per esempio slideshows) e debug di errori di configurazione server e dataBase.