Progetto Basi di Dati: Cinema Web App

Guidobene Davide, Rosin Giacomo

Lista funzionalità

Cliente (client)

- Interfaccia applicazione (pagine pubbliche)
    - Home page
    - Elenco film
    - Informazioni
- Autenticazione utenti
    - Registrazione
    - Login
    - Logout
- Elenco film disponibili
    - Lista con informazioni principali sui film disponibili
    - Filtro
        - per titolo
        - per genere
- Pagina per visualizzare i dettagli di un film comprese le proiezioni programmate
- Acquisto biglietti
    - Selezione della proiezione tra quelle disponibili
    - Selezione posti
    - Riepilogo acquisto con prezzo totale
    - Selezione metodo di pagamento tra quelli salvati o eventuale aggiunta di un nuovo metodo
- Area riservata
    - Visualizzazione cronologia acquisti
    - Visualizzazione profilo e possibilità di cambiare le credenziali
    - Visualizzazione metodi di pagamento salvati e possibilità di rimuoverli

Operatore (operator):

- Tutte le funzionalità disponibili per il client
- Dashboard di amministrazione
    - Operazioni sui dati
        - Inserimento
        - Visualizzazione
            - sistema di paginazione con limite di risultati per pagina
        - Modifica
        - Cancellazione
    - Statistiche

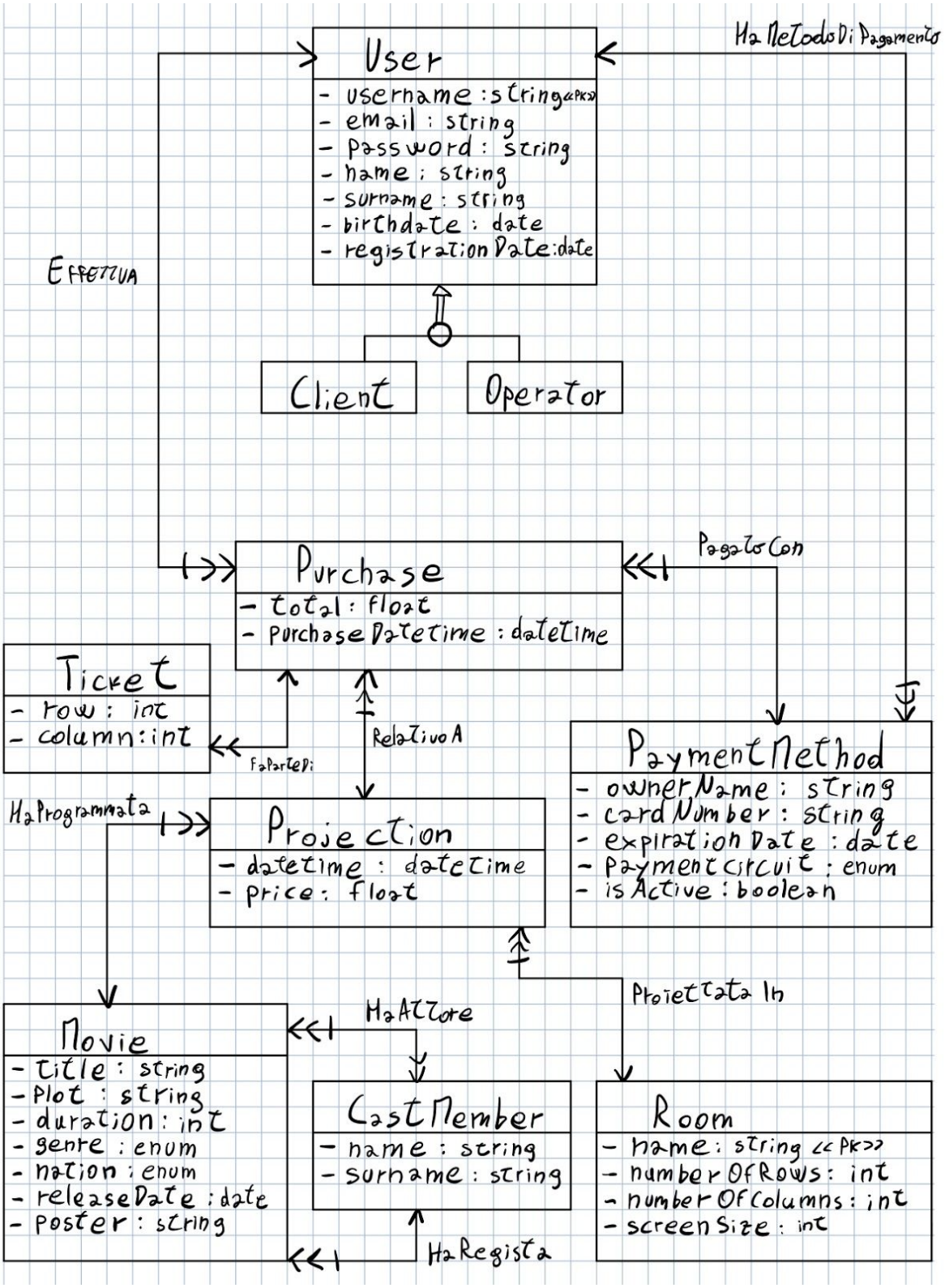
Progettazione schema base di dati

Da una prima analisi, le funzionalità elencate hanno come minimo bisogno delle seguenti relazioni:

- Una tabella per modellare le informazioni sugli utenti, in particolare per quanto riguarda la fase di autenticazione, email e password. Inoltre si rende necessario distinguere tra i 2 tipi di utenti: clienti e operatori del cinema.
- Una tabella per rappresentare i film disponibili e alcune loro informazioni principali.
- Una tabella per la programmazione delle proiezioni di un film.
- Una tabella per rappresentare i posti disponibili / biglietti acquistati.
- Una tabella per modellare i metodi di pagamento.
- Una tabella per memorizzare lo storico degli acquisti, in modo da poter fare delle valutazioni commerciali.

Dopo un’attenta analisi, e varie iterazioni, abbiamo prodotto il seguente schema concettuale, che, dal nostro punto di vista è sufficientemente espressivo per la realtà che dobbiamo modellare.

Progettazione concettuale



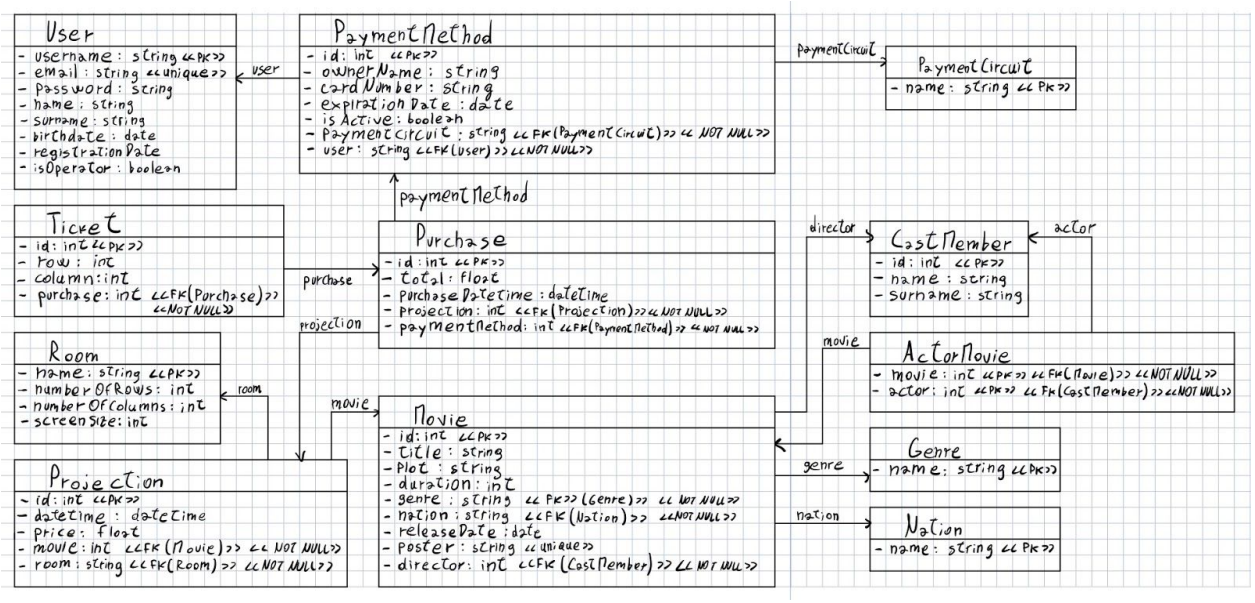
Entità

- **User**: modella un utente registrato della web app, in particolare siamo interessati a memorizzare informazioni utili per l'autenticazione, e alcuni dati generici che descrivano l'utente.
- **Client**: modella un cliente ed è sottoclasse di User; non siamo interessati a memorizzare nessuna informazione particolare oltre a quelle di User.
- **Operator**: modella un operatore ed è sottoclasse di User; non siamo interessati a memorizzare nessuna informazione particolare oltre a quelle di User.
- **Movie**: modella un film.
- **CastMember**: modella Attori e Registi; siamo interessati a mantenere solo nome e cognome. Abbiamo deciso di mantenere sia gli Attori che i Registi in un'unica tabella, senza distinguerli, per vari motivi, tra cui il fatto che spesso e volentieri attori sono anche registi e viceversa, e che non siamo interessati ad avere informazioni specifiche né per attori, né per registi.
- **Room**: modella le sale del cinema.
- **Projection**: modella la programmazione delle proiezioni dei film nelle varie sale e gli orari.
- **Purchase**: modella lo storico degli acquisti.
- **PaymentMethod**: modella i metodi di pagamento usati dagli utenti per effettuare gli acquisti.
- **Ticket**: modella i biglietti acquistati dagli utenti.

Altre scelte di design

- L'associazione tra User e Purchase è già modellata tramite le associazioni User-PaymentMethod e PaymentMethod-Purchase, per ciò per evitare ridondanza la ignoreremo.
- Abbiamo deciso che un acquisto riguarda un'unica proiezione, per semplificare il flow di acquisto, ed accorciare il più possibile il tempo tra la scelta di un biglietto e l'acquisto effettivo in modo da ridurre le probabilità che qualcuno tenti di acquistare lo stesso biglietto contemporaneamente.
- Tra CastMember e Movie vi sono due associazioni (HaAttore e HaRegista), in quanto appunto siamo interessati a conoscere sia gli attori che il regista (principale) di un film.
- User-PaymentMethod: uno a molti. Un utente può avere molti metodi di pagamento, ma un metodo di pagamento si può considerare personale e quindi usato da un unico utente.

Progettazione logica



Relazioni

- **User**: modella gli utenti registrati della web app; contiene sia i clienti che gli operatori, usando un discriminante booleano per distinguerli. Abbiamo adottato questa scelta perchè non vi erano informazioni specifiche ne dei clienti, ne degli operatori, inoltre ha pesato anche il fatto che non vi erano vincoli particolari (specifici di cliente o di operatore) con le altre entità in gioco. Il campo ‘username’ (come del resto lo sarebbe email) è una buona chiave primaria naturale.
- **Ticket**: modella i biglietti acquistati; ha una chiave esterna che punta al Purchase di cui fa parte. Usa una chiave primaria artificiale con autoincremento ‘id’.
- **PaymentMethod**: modella i metodi di pagamento usati dagli utenti; ha una chiave esterna verso User e una verso PaymentCircuit. Usa una chiave primaria artificiale con autoincremento ‘id’.
- **PaymentCircuit**: modella i circuito di pagamento accettati; questa relazione è nata nella fase di progettazione logica per modellare un tipo enumerato. Il nome del circuito di pagamento è certamente univoco, per cui ‘name’ è una buona chiave primaria naturale.
- **Purchase**: modella gli acquisti, in particolare ha una chiave esterna verso PaymentMethod per memorizzare il metodo di pagamento utilizzato e di conseguenza l’utente; inoltre ha una chiave esterna verso Projection ad

indicare a quale proiezione si riferisce l'acquisto. Usa una chiave primaria artificiale con autoincremento 'id'.

- **CastMember**: modella Attori e Registi. Usa una chiave primaria artificiale con autoincremento 'id'.
- **ActorMovie**: modella l'associazione molti a molti tra CastMember e Movie; per fare ciò utilizza 2 chiavi esterne, una verso CastMember e una verso Movie. Da notare che questi 2 attributi formano anche la chiave primaria, in quanto un attore non può prendere parte più di una volta ad un film.
- **Genre**: modella i generi dei film; questa relazione è nata nella fase di progettazione logica per modellare un tipo enumerato. Certamente non vi saranno 2 generi con lo stesso nome, per cui 'name' è una buona chiave primaria naturale.
- **Nation**: modella le nazioni di provenienza dei film; questa relazione è nata nella fase di progettazione logica per modellare un tipo enumerato. Non ci sono nazioni con lo stesso nome, per cui 'name' è una buona chiave primaria naturale. (Nota: Nation non è in relazione con CastMember ne con User, perchè abbiamo valutato che il cinema non aveva davvero un interesse nel modellare questo tipo di informazione).
- **Movie**: modella i film; ha una chiave esterna verso Nation, una verso Genre e una verso CastMember (regista). Tra gli attributi di movie non sembra esserci nessuna chiave primaria ragionevole, per cui usa una chiave primaria artificiale 'id'.
- **Projection**: modella la programmazione delle proiezioni dei film nelle varie sale e gli orari; perciò ha una chiave esterna verso Movie e una verso Room. Una chiave primaria interessante
- **Room**: modella le sale del cinema. Sembra ragionevole assumere che il nome della sala, la identifichi, per cui 'name' è chiave primaria.

Altre scelte di design

- Abbiamo deciso di modellare i tipi enumerati (Genre, Nation, PaymentCircuit) come relazioni, in quanto questa scelta dava innumerevoli vantaggi, prima di tutto permette di avere i valori memorizzati nel database in modo centralizzato, in modo che possono essere espansi o rinominati facilmente (senza dover poi cambiare parti di codice) e possono essere richiesti in qualunque momento.

Note

- Tutti gli attributi hanno il vincolo NOT NULL

Query interessanti

Le seguenti query possono essere di particolare interesse in quanto non sono state usate per completezza del progetto ma per fornire statistiche utili agli operatori del cinema

```sql
-- mostra i generi più comuni tra i film proiettati dal cinema in ordine non crescente
(decrescente)
-- conta il numero di film per ogni genere
SELECT "Movie".genre, count(*) AS genre_count
FROM "Movie"
GROUP BY "Movie".genre
ORDER BY genre_count DESC

-- mostra i generi di film più popolari tra i clienti del cinema in ordine non crescente
(decrescente)
-- conta numero di biglietti venduti per genere
SELECT "Movie".genre, count(*) AS genre_count
FROM "Movie", "Projection", "Purchase", "Ticket"
WHERE "Movie".id = "Projection".movie
  AND "Projection".id = "Purchase".projection
  AND "Purchase".id = "Ticket".purchase
GROUP BY "Movie".genre
ORDER BY genre_count DESC

-- mostra i generi di film del cinema più redditizi in ordine non crescente (decrescente)
-- calcola entrate per ogni genere
SELECT "Movie".genre, sum("Purchase".total) AS genre_profit
FROM "Movie", "Purchase", "Projection"
WHERE "Movie".id = "Projection".movie AND "Projection".id = "Purchase".projection
GROUP BY "Movie".genre
ORDER BY genre_profit DESC

-- mostra gli attori più popolari tra i clienti del cinema in ordine non crescente
(decrescente)
-- i 25 attori i cui film hanno venduto più biglietti
SELECT "CastMember".name, "CastMember".surname, count(*) AS ticket_count
FROM "CastMember", "ActorMovie", "Movie", "Projection", "Purchase", "Ticket"
WHERE "CastMember".id = "ActorMovie".actor
  AND "ActorMovie".movie = "Movie".id
  AND "Movie".id = "Projection".movie
  AND "Projection".id = "Purchase".projection
  AND "Purchase".id = "Ticket".purchase
GROUP BY "CastMember".id
ORDER BY ticket_count DESC
LIMIT 25
```

```sql
-- mostra i registi più popolari tra i clienti del cinema in ordine non crescente
(decrescente)
-- i 25 registi i cui film hanno venduto più biglietti
SELECT "CastMember".name, "CastMember".surname, count(*) AS ticket_count
FROM "CastMember", "Movie", "Projection", "Purchase", "Ticket"
WHERE "CastMember".id = "Movie".director
 AND "Movie".id = "Projection".movie
 AND "Projection".id = "Purchase".projection
 AND "Purchase".id = "Ticket".purchase
GROUP BY "CastMember".id
ORDER BY ticket_count DESC
LIMIT 25
```

La seguente merita un'analisi più approfondita

```sql
SELECT "Movie".genre,
       count(*) AS ticket_count
       CASE
         WHEN (
             EXTRACT(year FROM CURRENT_DATE) - EXTRACT(year FROM "User".birthdate) >= 0
             AND EXTRACT(year FROM CURRENT_DATE) - EXTRACT(year FROM "User".birthdate) <= 19
         ) THEN '0-19'

         WHEN (
           EXTRACT(year FROM CURRENT_DATE) - EXTRACT(year FROM "User".birthdate) >= 20
           AND EXTRACT(year FROM CURRENT_DATE) - EXTRACT(year FROM "User".birthdate) <= 29
         ) THEN '20-29'
       ...
         ELSE '100+'
       END AS age_group
FROM "Movie", "User", "Projection", "Purchase", "Ticket", "PaymentMethod"
WHERE "Movie".id = "Projection".movie
 AND "Projection".id = "Purchase".projection
 AND "Purchase".id = "Ticket".purchase
 AND "Purchase"."paymentMethod" = "PaymentMethod".id
 AND "PaymentMethod"."user" = "User".username
GROUP BY age_group, "Movie".genre
ORDER BY ticket_count DESC
```

Da questa query si può estrarre la stessa analisi vista da 2 prospettive diverse:

1. Si può determinare, per ogni genere, la classifica delle fasce d'età che lo seguono di più.
2. Si può determinare, per ogni fascia di età, la classifica dei generi più visti.

La query utilizza un costrutto non comune, ovvero il costrutto CASE, che in questo caso non non fa altro che, per ogni utente, estrarre l'età (tramite `EXTRACT(year FROM CURRENT_DATE) - EXTRACT(year FROM "User".birthdate))e`

assegnare all’utente, sotto l’attributo age_group, una delle stringhe che vengono dopo i THEN, a seconda di quale condizione risulta vera per il determinato utente.
La query prende tutti i film, per ogni film le sue proiezioni, per ogni proiezione gli acquisti relativi, e per ogni acquisto l’elenco dei biglietti venduti; prende inoltre tutti gli utenti che hanno acquistato queste proiezioni passando per i metodi di pagamento.
Al termine raggruppa le tuple selezionate per fascia di età e genere, ottenendo così per ogni fascia di età e per ogni genere il conteggio dei biglietti venduti.

Scelte progettuali

Scelta DBMS

Abbiamo deciso di utilizzare come DBMS PostgreSQL, per vari motivi, tra cui:

- era il DBMS consigliato per il modulo 1 del corso di Basi di Dati
- è un progetto open source
- è tra i DBMS che si attiene maggiormente allo standard SQL
- ha un'ottima documentazione online

Vincoli

- User.registrationDate <= CURRENT_DATE
- User.registrationDate > User.birthDate
- PaymentMethod.cardNumber -> 16 cifre
- Ticket.row <= Room.numberOfRow AND Ticket.column <= Room.numberOfColumn
- La tripla (Ticket.row, Ticket.column, Ticket.Purchase.projection) è unique
- Purchase.purchaseDateTime <= Projection.dateTime AND Purchase.purchaseDateTime <= CURRENT_DATE_TIME
- Projection.dateTime >= Movie.releaseDate
- Tutti i numeri >= 0
- a.dateTime<b.dateTime AND a.room=b.room -> a.dateTime+a.length<b.dateTIme (per ogni coppia di proiezioni distinte a e b) (Non deve esserci overlapping tra proiezioni nella stessa stanza)

Check

Esempio check:

```
"numberOfColumns" INTEGER NOT NULL
                                CONSTRAINT positive_number_of_cols
                                CHECK ("numberOfColumns" > 0),
```

Trigger

Esempio trigger:

```
-- verifica che data di acquisto sia antecedente alla data di proiezione
CREATE TRIGGER purchase_datetime_before_projection_datetime
BEFORE INSERT ON "Purchase"
 FOR EACH ROW
 WHEN (NEW.purchaseDatetime >  (SELECT Projection.datetime
```

```
                                FROM Projection
                                WHERE Projection.id=NEW.projection))
EXECUTE RAISE_APPLICATION_ERROR( -20001, 'old projections cannot be purchased');
```

Una descrizione più approfondita dei trigger può essere trovato al path ***db/sql/trigger.sql*** del progetto

Transazioni

Nell'implementazione del progetto abbiamo avuto necessità di utilizzare un'unica transazione, ovvero quella per la finalizzazione dell'acquisto dei biglietti. La transazione in questione è composta principalmente dalle seguenti query:

1. *Selezione dei biglietti venduti per la proiezione di interesse*.
    a. Questa query serve a verificare se i posti che l'utente ha selezionato sono disponibili.
2. *Inserimento dell'acquisto.*
    a. Se sono disponibili i posti allora viene eseguita la query che si occupa di inserire l'acquisto nella tabella Purchase.
3. *Inserimento dei biglietti.*
    a. Infine, viene eseguita la query che inserisce i biglietti selezionati.

Le query 2 e 3 devono essere eseguite solo se i biglietti sono disponibili, quindi dipendono dal risultato della query 1. E' necessario assicurarsi che se un'altro utente tenta di acquistare gli stessi biglietti contemporaneamente, la concorrenza venga risolta in modo corretto, ovvero bloccando uno dei 2. Infatti non deve mai accadere che un biglietto venga venduto a 2 utenti.
Inoltre le query 2 e 3 modificano lo stato della base di dati, non è accettabile che una delle 2 venga portata a termine e l'altra no, perché ciò porterebbe la base di dati in uno stato inconsistente. E' necessario quindi assicurarsi che o vengano portate a termine entrambe o vengano annullate entrambe.
Queste 2 situazioni sono proprio il caso d'uso delle transazioni. Per questa transazione abbiamo utilizzato il livello di isolamento SERIALIZABLE, l'unico che ci dava garanzie sul corretto funzionamento della transazione.

Indici

- User (username, email)
    - l'email non è né PK né FK, tuttavia è unique (e quindi un attributo fortemente discriminativo) ed è usata al login di ogni utente!
- PaymentMethod (id, user)

- Ticket (purchase)
- Purchase (id, projection, paymentMethod)
- Projection (id, movie)
- Movie (id, genre)
    - anche se genre non è un attributo discriminativo come per esempio title, questo attributo è usato in molte query, in particolare per i GROUP BY, pertanto un indice su di esso può portare a un significativo incremento delle performance
- CastMember (id)

La maggior parte degli attributi su cui abbiamo usato un indice sono PK o FK molto usati nelle query, spesso per operazioni di JOIN.
Nota: le tabelle Room, PaymentCircuit, Genre e Nation sono molto piccole e poco usate dal momento che sono state create principalmente con lo scopo di funzionare come un tipo enum.
Per queste ragioni non abbiamo ritenuto opportuno assegnarvi un indice.

Informazioni utili per apprezzare il progetto

- Validazione input utente sia lato client per una maggiore responsività, sia lato server per motivi di sicurezza.
- Utilizzo assiduo di bindparam nelle query quando si ricevono parametri dall'esterno (anche indirettamente).
- Suddivisione della web app in flask.Blueprint, ovvero suddivisione in moduli della web app. Ciò ha permesso di avere una migliore organizzazione del codice e delle route.
- Database inizializzato con svariati dati per una miglior esperienza di test e usabilità.
- In caso di tentativo di accesso di un utente non loggato ad una pagina in cui è richiesto il login, l'utente viene reindirizzato alla view di login e, subito dopo essersi loggato, viene rimandato alla pagina a cui stava tentando di accedere, il tutto a formare una migliore user experience.
- Utilizzo di logger per loggare l'avvenimento di alcuni eventi più importanti.
- Meccanismo di autorizzazione per l'accesso alle route per soli operatori (esempio dashboard amministratori) implementata tramite decoratore @operator_required, in modo da fornire un metodo consistente (simile a quello di Flask-Login (@operator_required)) per l'autorizzazione all'accesso alle varie pagine.
- Form di selezione posti molto intuitivo e
- Utilizzo del costrutto with per effettuare le query, in modo da non doversi preoccupare della chiusura della connessione al termine di ogni sessione di query (e certezza che verrà chiusa).