

RELAZIONE PROGETTO BASI DI DATI 2020

GRUPPO: ARCECITY



ArceCity Cinema

RELAZIONE DI
Casarotti Giulio,
Ferrari Simone,
Trolese Giulio.

UNIVERSITÀ CA' FOSCARI, VENEZIA

SOMMARIO

INTRODUZIONE	4
COME AVVIARE IL PROGETTO	4
SCHEMA DEL DATABASE	5
GUIDA APPLICAZIONE WEB	6
ERRORI GESTITI	8
CREAZIONE DEL DATABASE	9
File create_database.py	9
Creazione del database	9
Creazione tabelle	9
Definizione dei ruoli	10
Definizione dei trigger	11
Nella nostra applicazione abbiamo definito un TRIGGER con lo scopo di:	11
File fill_database.py	11
Riempimento database con dati di esempio	11
APPLICAZIONE PYTHON	12
Main Route - File app.py	12
Admin Route - File admin_route.py	12
User Route - File user_route.py	16
Query - File query.py	18
Engine - File engine.py	18
DESCRIZIONE QUERY	19
File query.py	19
generate_my_projection_dict()	19
generate_all_film_next_projection	19
generate_all_projection_film(id)	19
generate_all_projection	19
Query ad uso statistico: STAT_*	19
TEMPLATE HTML	20
Main Template	20
Admin Template	20
User Template	21
DOCUMENTAZIONE	21

RELAZIONE PROGETTO BASI DI DATI 2020

ARCECITY: Casarotti Giulio, Ferrari Simone, Trolese Giulio

INTRODUZIONE

L'**obiettivo** del nostro progetto è la **creazione di una applicazione web** che si interfacci con un Database. Il **tema** scelto è il **Cinema**.

Per la creazione della nostra applicazione web abbiamo utilizzato **Flask** con **SQLAlchemy**: una libreria di funzioni python utile ad interfacciarsi con un Database.

Abbiamo cercato di utilizzare il più possibile L'**Expression Language**, scrivendo le query in un "linguaggio generico" che poi vengono compilate in modo appropriato in base al DBMS sottostante su cui si lavora. Questo ci permette di essere **indipendenti dal DBMS** scelto.

Il **DBMS** scelto è **PostgreSQL**.

Nella seguente documentazione, salteremo alcune parti realizzative "scontate", inseriremo comunque tutti i **riferimenti ai file citati**, in modo tale da **poter visionare il codice sorgente** per intero.

COME AVVIARE IL PROGETTO

Per poter avviare il progetto in locale, bisogna seguire la seguente procedura:

Assumiamo che sul vostro pc sia già presente PostgreSQL e un terminale linux configurato correttamente (Python, Flask, SQLAlchemy, ...)

- **Avviate** il server **PostgreSQL** sul vostro pc
- Da **terminale**, recatevi nella **cartella** nel quale sono presenti i file del **progetto**.
- Nel file `create_database.py`, **modificate** la **linea 20** con il vostro **nome utente** e **password** della vostra **configurazione postgresQL**
- **Create** il **database**: `python3 create_database.py`
- Inserite alcuni **dati di esempio** (Facoltativo) nel database: `python3 fill_database.py`
- **Esportate** l'applicazione **Flask**: `export FLASK_APP=app.py;`
- Per sviluppare il progetto, abbiamo attivato le modalità **development** e **debug**:
 - `export FLASK_ENV=development;`
 - `export FLASK_DEBUG=1`
- **Avviate l'applicazione**: `flask run`
- Recatevi in un qualsiasi **browser** e **collegatevi** all'indirizzo: <http://127.0.0.1:5000/>
- **Se** vengono **apportate modifiche al codice** mentre l'applicazione flask è in esecuzione, potete utilizzare (Se la modalità DEBUG è attiva) la combinazione `shift+f5` per ricaricare la pagina con il ripristino della cache. In questo modo tutte le modifiche verranno applicate.

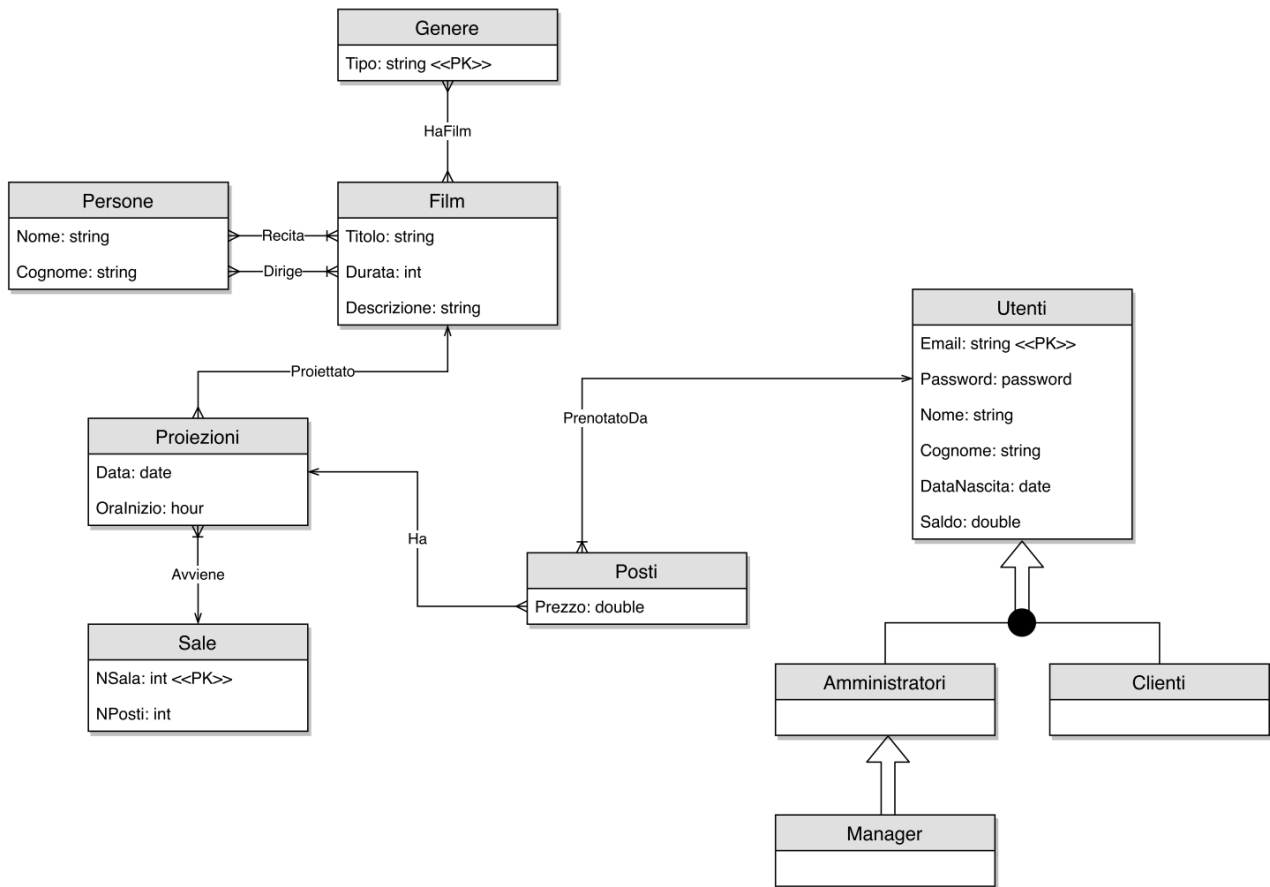
Controllate di aver salvato tutti i file modificati, altrimenti non noterete nessun cambiamento!

Una volta avviato il tutto potete loggare con le **seguenti credenziali**:

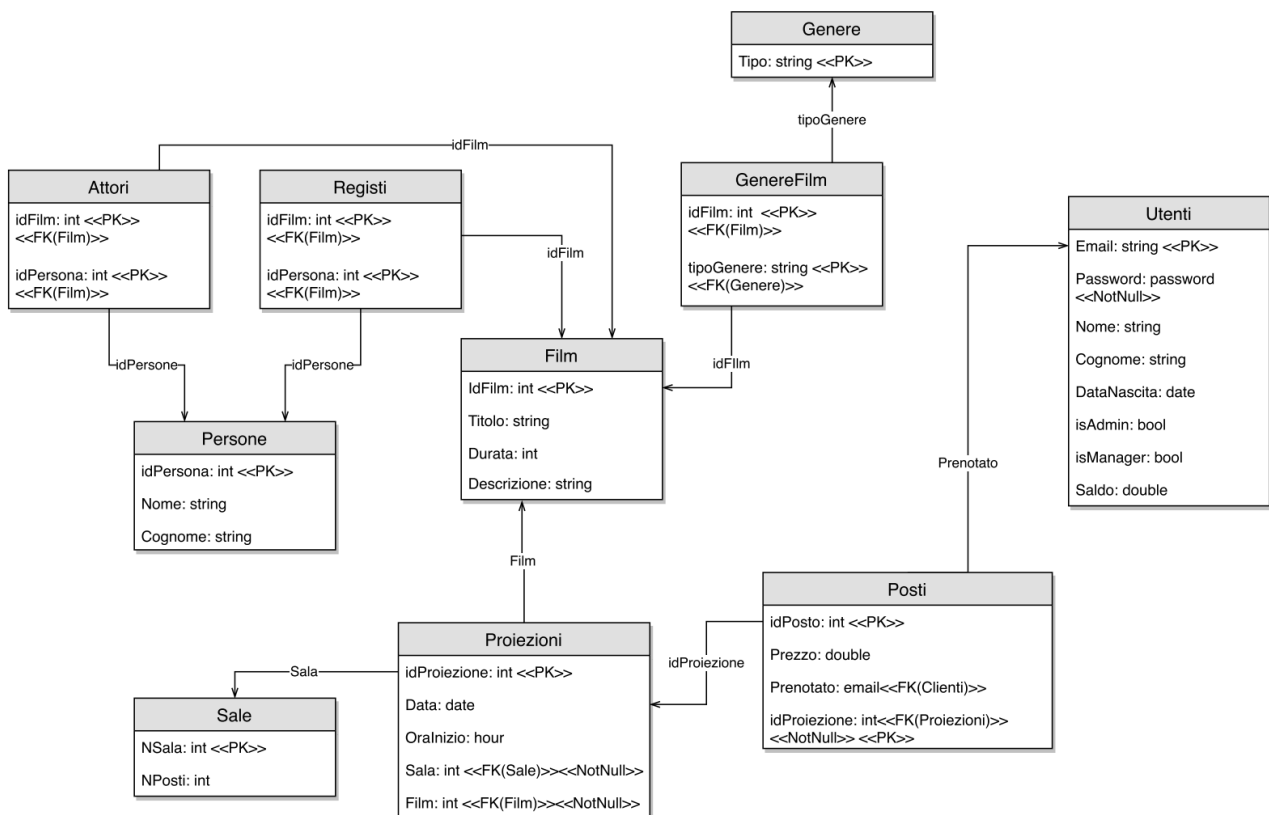
- | | | |
|---|----------------------------|------------------------|
| ● email : simoneferrari@gmail.com | Password : password | Ruolo : Admin |
| ● email : giulioasarotti@gmail.com | Password : password | Ruolo : Manager |
| ● email : giuliotrolese@gmail.com | Password : password | Ruolo : Cliente |
| ● Potete non loggare per navigare il sito come Anonimous (vedere ruoli) | | |

SCHEMA DEL DATABASE

Schema ad oggetti



Schema relazionale



GUIDA APPLICAZIONE WEB


Home

Dalla **home**, sono visibili le **prossime cinque proiezioni**.

Tramite il pulsante sulla barra superiore, si può accedere alla **lista completa dei film disponibili**, le **proprie prenotazioni** e la **dashboard del proprio account**.

Nella parte **superiore-destra** si può accedere al **login** e **logout**.

Vicino ai film visualizzati, è possibile **acquistare biglietti** per quella proiezione oppure **visionare altre date** previste per quel determinato film.



Titanic

195 minuti

Titanic è un film del 1997 co-montato, co-prodotto, scritto e diretto da James Cameron.

Prossima proiezione: 2020-08-30 16:00:00 Sala: 1

[Acquista](#)

[Altre date per questo film](#)

Registrazione e login

Se si è già in possesso di un account, in fase di **login** basterà **inserire** le proprie **credenziali**.

Altrimenti ci si può facilmente **iscrivere** mediante il **form** di **registrazione**.

Tutti i film

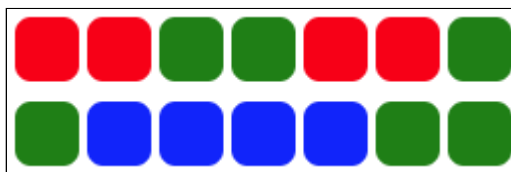
In questa pagina vengono **visualizzati tutti i film nel nostro catalogo**, con **annessa la prossima proiezione** più vicina.

Analogamente alla Home, **vicino ai film visualizzati** è possibile **prenotare** dei **posti** oppure **visionare tutte le date di proiezione previste**.

Acquisto biglietti

In fase di **acquisto biglietti**, l'utente deve semplicemente selezionare i posti che desidera acquistare.

I **posti liberi** sono indicati in **verde**, quelli **occupati** in **rosso**, e quegli che si sono **selezionati** sono indicati in **blu**.



Sul **fondo pagina**, è visualizzato il **costo previsto** per la prenotazione.

Se si desidera procedere, cliccare su **conferma acquisto**.

Importo previsto:
20.0€

[Conferma acquisto](#)

Se non si dispone di saldo sufficiente, verrete reindirizzati alla **pagina di ricarica saldo**.

Altre proiezioni programmate per un film

Da questa pagina, è possibile **vedere tutte le proiezioni previste per un determinato film**.

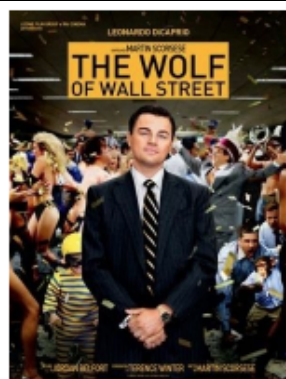
Basterà **scegliere quella che desiderata** e procedere con la prenotazione mediante il pulsante **acquista**.

Le mie prenotazioni

Se si desidera vedere il **riepilogo delle proprie prenotazioni**, questa è la pagina giusta.

Saranno subito disponibili informazioni riguardante tutte le proiezioni prenotate con il relativo numero di biglietti acquistato.

E' anche possibile **aggiungere altri posti** alla propria **prenotazione**.



The Wolf of Wall Street

180 minuti

The Wolf of Wall Street è un film del 2013 diretto e prodotto da Martin Scorsese.

Proiezione: 2020-08-30 21:00:00 Sala: 1

Numero di biglietti prenotati: 2

[Prenota altri posti](#)

Il mio account

In questa pagina sono disponibili le informazioni principali del proprio account.

E' possibile **vedere il proprio saldo** e **ricaricarlo**.

Inoltre, se si vuole **cambiare password**, questo è il luogo giusto.

Simone Ferrari

simone@gmail.com

Saldo: €10.0 [Ricarica Saldo](#)

[Modifica password](#)

Modifica password

Inserite la **vecchia password**, la **nuova password** e **confermate** la **nuova password**.

Se i dati inseriti sono corretti, avrete **cambiato** la **password** con estrema facilità!

Gestione sito web

Se si è **acceduto** mediante un **account** avente i privilegi di **amministratore**, nella **home** sarà presente un pulsante che vi porterà nella home della gestione del sito.

Sei un amministratore!

[Gestisci il sito](#)

Da qui, **sarà possibile**:

- Aggiungere / rimuovere film
- Aggiungere / rimuovere proiezioni
- Aggiungere persone (attori e/o registi)
- Aggiungere generi
- Aggiungere sale
- Aggiungere un profilo amministratore
- Ottenere varie statistiche

ERRORI GESTITI

Registrazione

- Registrazione con email già utilizzata da qualcuno
- Password e conferma password errate
- Campi del form html devono prendere valori corretti:
 - La mail deve essere nel formato nome@dominio.it/com/...
 - Nome e cognome non devono presentare all'interno caratteri speciali

Login

- Email errata / non presente nel database
- Password errata

Controllo accesso pagine per anonimo/cliente/admin

- Tutte le pagine che richiedono di essere loggati generano un errore se si tenta di accedervi senza avere i permessi necessari.

Modifica password

- Password vecchia deve corrispondere con quella realmente utilizzata
- La nuova password e la conferma della nuova password devono combaciare
- La vecchia password e quella nuova non possono essere uguali

Altre proiezioni per un film

- Controllo che si sia effettuata una scelta

Prenotazione posti

- Impossibilità di prenotare posti già occupati
- Prima di effettuare l'acquisto, bisogna avere un saldo sufficiente

Aggiunta film

- Inserimento multiplo dello stesso attore, regista e/o genere in un singolo film

Aggiungi genere

- Inserimento di un genere già presente nel database

Aggiunta amministratore

- Registrazione con email già utilizzata da qualcuno
- Password e conferma password errate
- Campi del form html devono prendere valori corretti:
 - La mail deve essere nel formato nome@dominio.it/com/...
 - Nome e cognome non devono presentare all'interno caratteri speciali

Altri errori e prevenzione SQL Injection

- Sono stati gestiti altri piccoli possibili errori che non avrebbero però corrotto l'integrità del database.
- Abbiamo protetto il database da possibili **SQL Injection** applicando dei controlli su tutti i campi a "scrittura libera", vietando simboli speciali quali ";" e "-"

CREAZIONE DEL DATABASE

File [create_database.py](#)

Questo script python viene eseguito solo una volta, in fase di creazione.

Creazione del database

Per la **creazione del database**, ci appoggiamo a **SQLAlchemy**: una libreria di Python pensata per interfacciarsi con i database SQL.

Ci appoggiamo il più possibile a funzioni python (**Expression Language**) per due motivi:

- Siamo indipendenti dal DBMS sul quale si lavora
- La portabilità è aumentata: ci è possibile ricreare il database su altri DBMS senza problemi

Una volta **importati i moduli** necessari...

```
from sqlalchemy import create_engine, MetaData, Table, Column, Integer, ...
from sqlalchemy_utils import database_exists, create_database, EmailType, ...
...
```

...possiamo **creare l'engine**

```
engine = create_engine("postgres+psycopg2://utente:pw@serversrv.ddns.net:2345/nomeDB")
```

dove:

[postgres](#): Driver per il DBMS che utilizziamo

[psycopg2](#): Dialecto utilizzato

[utente:password](#): nomeUtente : password

[serversrv.ddns.net:2345](#): IpServer : porta

[/nomeDB](#): nome database

Questo file viene eseguito esclusivamente per creare il database: di conseguenza, il seguente frammento di codice serve a controllare se il database esiste già, e, in caso affermativo, lo elimina e lo ricrea. Se invece non esiste, lo crea semplicemente.

Le seguenti funzioni appartengono alla libreria [sqlalchemy_utils](#).

```
if database_exists(engine.url):
    drop_database(engine.url)
if not database_exists(engine.url):
    create_database(engine.url)
```

Creazione tabelle

Prima di creare le tabelle, dobbiamo **definire un oggetto** che avrà la funzione di **contenere al suo interno tutte le relazioni che verranno create**.

```
metadata = MetaData() #oggetto su cui vengono salvate tutte le tabelle
```

Creiamo ora una **tabella**: [utenti](#)

```
utenti = Table('utenti', metadata,
               Column('nome', String(255)),
               Column('cognome', String(255)),
               Column('data_nascita', Date),
               ...
               )
```

I tipi utilizzati sono descritti nella documentazione: [Data Types](#)

E facciamo la stessa cosa per le tabelle: `film`, `genere`, `persone`, `sale`.

Adesso che abbiamo creato le tabelle prive di **Foreign Key**, possiamo creare le relazioni che invece né presentano all'interno: `genere_film`, `proiezioni`, `posti`, `attori`, `registi`

Vediamo un esempio: tabella `Proiezioni`

```
proiezioni = Table('proiezioni', metadata,
    Column('id_proiezione', Integer, primary_key = True),
    Column('data', Date),
    Column('ora_inizio', Time),
    Column('sala', Integer, ForeignKey("sale.n_sala"), nullable = False),
    Column('film', Integer, ForeignKey("film.id_film"), nullable = False)
)
```

Sarebbe stato possibile anche creare subito tutte le tabelle (senza definire le foreign key) e, successivamente, attraverso `alter table`, assegnare le foreign key.

Una volta definite tutte le tabelle, le andiamo a **creare nell'engine**

```
metadata.create_all(engine)
```

Definizione dei ruoli

I ruoli sono stati definiti nel file `create_database.py`.

La nostra gestione dei permessi avviene nel seguente modo:

- **Creiamo quattro utenti** nel database:
 - Admin
 - Manager
 - Cliente
 - Anonim
- **Creiamo 4 ruoli** a cui assegniamo il rispettivo utente e i rispettivi permessi
 - Superusers
 - Tutti i privilegi su tutte le tabelle
 - Managers
 - SELECT, UPDATE, INSERT su tutte le tabelle
 - Clienti
 - SELECT su tutte le tabelle
 - UPDATE su utenti
 - INSERT su posti
 - Anonymous
 - SELECT su tutte le tabelle
 - INSERT su utenti
- **Utilizziamo 4 engine diversi in fase di collegamento al database**
 - Eseguiamo le query utilizzando l'engine "più basso" avente tutti i permessi necessari per la query desiderata.
- Abbiamo **due flag boolean** (`is_admin`, `is_manager`) **all'interno della tabella utenti** per indicare se un utente è admin o manager

Definizione dei trigger

Nella nostra applicazione abbiamo **definito un TRIGGER** con lo scopo di:

Rimborsare tutti i clienti, che hanno acquistato dei biglietti per una certa proiezione, nel caso che tale proiezione (non ancora avvenuta) venga cancellata.

Tale trigger è contenuto nel file [create_database.py](#) e lo descriviamo qui in breve.

```
CREATE TRIGGER refund
BEFORE DELETE ON proiezioni
FOR EACH ROW
WHEN (OLD.data > current_date OR
      (OLD.data = current_date AND OLD.ora_inizio > current_time))
EXECUTE PROCEDURE refund()
```

Creiamo il trigger chiamato “[refund](#)” che, **prima che avvenga una cancellazione** dalla tabella **proiezioni**, per tutte le righe in cui la **data** della proiezione da cancellare è **successiva alla data odierna** (oppure è uguale, ma l’ora è successiva all’ora attuale), andiamo ad **eseguire la procedura [refund\(\)](#)** che si occupa di **rimborsare** ai clienti i **biglietti acquistati** per tale prenotazione.

File [fill_database.py](#)

Tramite questo file, è possibile aggiungere dati di esempio al database.

Riempimento database con dati di esempio

Abbiamo inserito:

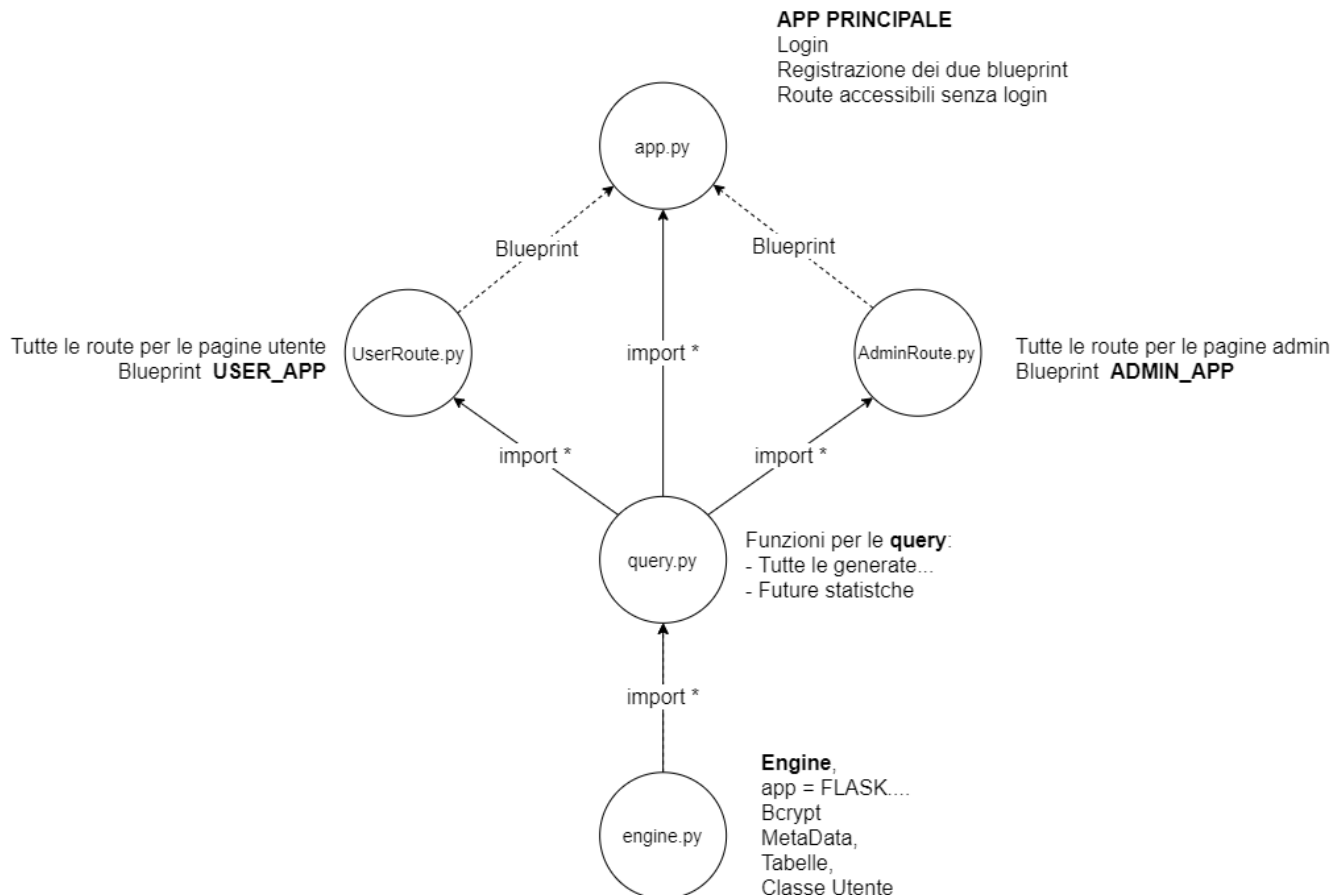
- 8 utenti (6 Clienti, 1 Admin, un manager)
- 15 Film
- 12 generi
- 25 persone (Attori e registi)
- 3 Sale
- 32 relazioni genere-film
- 60 proiezioni, 4 al giorno, dal 30/08 al 13/09
- 30 posti già prenotati
- 25 relazioni attori-film
- 20 relazioni registi-film

APPLICAZIONE PYTHON

La nostra applicazione è **suddivisa nei seguenti file**:

- App.py
- Admin_route.py
- User_route.py
- Query.py
- Engine.py

E i file sono **importati come nel seguente schema**:



Main Route - File [app.py](#)

Questo script python è la **vera e propria applicazione, il "main"**.

Importiamo il file query e i due blueprint.

importiamo i **blueprint** per admin_app e user_app.

```
app.register_blueprint(user_app)
app.register_blueprint(admin_app)
```

Route: /

Questa è la route della home, la pagina iniziale.

```
@app.route('/')

```

Con la funzione [generate_all_film_next_projection\(\)](#) creiamo una lista di dizionari contenenti tutti i film con la relativa prossima proiezione. Nella home **visualizziamo i prossimi cinque film in sala**.

Route: /login

Definiamo la **funzione** `load_user(user_email)` che ha il compito di restituire l'utente associato alla mail presa come parametro.

Inoltre, definiamo **come** devono **comportarsi** le **Route** che **richiedono** il **login** (`@LoginRequired`) quando un **utente non è loggato**: `unauthorized()`

E' finalmente arrivato il momento di **renderizzare** la **pagina html** del login e di gestirne il **form**.

La **funzione** `login()` si occupa di questo.

Route: /logout

Per **uscire dal proprio** profilo, basta richiamare la funzione di flask_login `logout_user()`

Route: /registrazione

Per poter visionare la nostra applicazione web, non è necessario iscriversi (si è definiti utenti **anonimi**).

La registrazione, invece, diventa **obbligatoria** in molte sezioni, e ci classifica come **clienti**. (Vedi [ruoli](#))

Definiamo la **route** per la pagina di sign_up

```
@app.route('/registrazione', methods=['GET', 'POST'])
```

Successivamente, se il metodo è **POST**, andiamo a **leggere** i dati dal **form di registrazione**

```
nome = request.form["nome"]
...
psw = request.form["psw"]
conferma = request.form["conferma_password"]
```

Dato che la **password** è inserita come una normalissima stringa, andiamo a **criptarla** con il **Bcrypt**

```
hashed_psw = bcrypt.generate_password_hash(psw).decode('utf-8')
```

Documentazione per il Bcrypt: [Flask Bcrypt](#)

Controlliamo ora **che le due password inserite coincidano**.

Se **non coincidono**, renderizziamo nuovamente la pagina di inserimento dei dati presentando all'utente un **errore**.

```
if(psw != conferma):
    return render_template('registrazione.html', errore=True)
```

Altrimenti procediamo con **l'inserimento dei dati nel database**.

```
else:
    ins = clienti.insert() # predo la insert
    values = { # dizionario per i valori da inserire
        'nome': nome,
        ...
    }
    conn = anonim_engine.connect() # apro connessione con l'engine
    conn.execute(ins, values) # eseguo l'inserimento dei valori
    return redirect(url_for(login)) # indirizzo alla pagina login
```

Se invece il **metodo** è **GET**, semplicemente **renderizzo** il **template** della pagina di **registrazione**

```
else:
    return render_template('registrazione.html', errore=False)
```

Route: /tutti_i_film

Tramite questa route, **tutti** possono **visualizzare tutti i film presenti nel nostro cinema**.

E' anche possibile **filtrarli** mediante la **scelta** del **genere**.

```
@app.route('/tutti_i_film')
```

Utilizziamo la funzione `generate_all_film_next_projection()` che ci genera la lista di tutti i film con associata la prossima proiezione in programma.

Route: /altre_date_film

Tramite questa route si possono **visualizzare tutte le proiezioni future** per un **determinato film**.

```
@app.route('/altre_date_film/<id_film>', methods=['GET', 'POST'])
```

Utilizziamo la funzione `generate_all_projection_film(id_film)` che genera tutte le proiezioni future relative al film identificato da `id_film`

Admin Route - File `admin_route.py`

Tutte le seguenti Route richiedono il login: sono pagine dedicate agli **amministratori** ([ruoli](#))

Importiamo il file query e creiamo il **Blueprint admin_app**

```
admin_app = Blueprint(admin_app, __name__)
```

Route: /home_gestione_sito

In questa Route sono **raccolti tutti i collegamenti alle varie pagine** utili ad un **amministratore / manager** per gestire il sito.

```
@app.route('/home_gestione_sito')
```

Route: /aggiungi_persona

Nel nostro database abbiamo bisogno di salvare dati su delle **persone** (generiche).

Esse poi saranno associate ai film come **attori** o **registi**.

Spiegheremo in **modo completo** la procedura di **acquisizione dati** da un **form html**, successivamente salteremo i passaggi meno importanti.

Da qui, quindi, un **amministratore / manager** può aggiungere una persona al database.

Definiamo la **route** per la pagina di aggiunta persone

```
@app.route('/aggiungi_persona', methods=['GET', 'POST'])
```

E **gestiamo la differenza tra metodo GET e POST**

```
def registrazione():
    if request.method == 'POST':
```

Successivamente, se il metodo è **POST**, andiamo a **leggere** i dati dal **form**

```
    nome = request.form["nome"]
    cognome = request.form["cognome"]
```

ed **andiamo ad inserirli** nella tabella persone del nostro database.

```
    ins = persone.insert()
    values = {
        'nome': nome,
        'cognome': cognome,
    }
    conn = admin_engine.connect()
    conn.execute(ins, values)
```

```

if request.form["Submit"] == "Film": #due diversi bottoni di submit
    return redirect(url_for('aggiungi_film'))
else:
    return redirect(url_for('aggiungi_persona'))

```

Route: aggiungi_film

Tramite questa route, un **amministratore / manager** può **aggiungere film al database**.

Definiamo la **route** per la pagina di aggiunta film

```
@app.route('/aggiungi_film', methods=['GET', 'POST'])
```

Dato che ad un film vogliamo associare degli **attori/registi e dei generi**, dobbiamo prima **generare un dizionario** contenente tutte le persone del database e un dizionario contenente i generi: questo perché vogliamo **evitare** che, in fase di aggiunta di un film, venga **inserito** un attore/regista/genere **non presente nel database**.

Quindi, se il **metodo** è **GET**, **generiamo i dizionari** con la seguenti funzioni (Vedi [sezione query](#))

```

dict_p = generate_persons_dict()
dict_g = generate_genres_dict()

```

Possiamo ora **renderizzare il template**, passando come parametri i dizionari appena calcolati.

```

return render_template('aggiungi_film.html',
    persone_dict = json.dumps(dict_p), generi_dict = json.dumps(dict_g))

```

Sotto metodo **POST**, invece, andiamo a **gestire il form**.

La questione qui è un po' più complicata, in quanto:

- Quando aggiungiamo un film, **dobbiamo inserire** nelle tabelle **attori**, **registi** e **genere_film** il **collegamento** al **film appena aggiunto**: questo avviene attraverso l'**id** del film.
Noi sappiamo l'id del film che stiamo inserendo? No, per conoscerlo, dobbiamo prima inserire il film nel database.
Quindi, **per assegnare correttamente la relazione** film-attore, film-regista e film-genere, è necessario:

- Inserire il film
- Con una query prendere l'ID dell'ultimo film inserito
- Assegnare l'ID agli attori, registi e generi

Dobbiamo però **assicurarci** che, **tra l'inserimento** del film e la **successiva query** per estrapolare l'id, **non vi siano altri utenti che inseriscano altri film**: l'id estrapolato altrimenti sarebbe errato.

ABBIAMO BISOGNO DI UNA TRANSAZIONE.

- Non sappiamo con precisione il numero** di attori, registi e generi che un film avrà: abbiamo bisogno di un **form dinamico**.

Prima di tutto, prendiamo i **valori** di cui siamo **sicuri** e **aggiungiamo il film**:

```

titolo = request.form["titolo"]
durata = request.form["durata"]
descrizione = request.form["descrizione"]
ins = film.insert() # prendo la insert
values = { # dizionario per i valori
    ...
}

```

Ora, possiamo **avviare** la **transazione** per **estrapolare l'id** del film appena inserito

```
with admin_engine.connect().execution_options(isolation_level="SERIALIZABLE")
as conn:
    trans = conn.begin()
    try:
        conn.execute(ins, values)
        sel = select([func.max(film.c.id_film).label('latest_film')])
        result = conn.execute(sel)
        id_film = result.fetchone()['latest_film']
        trans.commit()
    except:
        trans.rollback()
    finally:
        conn.close()
```

Una volta che abbiamo ottenuto l'id corretto, dobbiamo andare ad **aggiungere attori, registi, generi**.

```
for elem in request.form: #non so quanti siano...
    # se è un attore
    if "attori" in str(elem):
        id_attore = request.form[str(elem)]
        ins_attori = attori.insert()

        attori_values = {
            "id_film": id_film,
            "id_persona": id_attore
        }
        # aggiungo i dati alla tabella attori
        conn.execute(ins_attori, attori_values)
```

E faccio lo stesso per registi e generi.

Abbiamo così aggiunto un film al database, con i relativi attori, registi e generi.

Route: /aggiungi_admin

Mediante questa Route, un **amministratore** può **registrare nel database un nuovo** utente, che, però, sarà un nuovo **amministratore**.

Abbiamo scelto di creare un nuovo profilo per un amministratore a prescindere che lui sia già registrato o meno come cliente: **vogliamo che un amministratore abbia un profilo diverso** per amministrare il sito/database rispetto al profilo con cui acquista biglietti.

Route: /riepilogo_sale

Tramite questa route un **amministratore / manager** può **visionare il riepilogo** delle nostre **sale** con **possibilità di aggiungerne**.

```
@app.route('/riepilogo_sale', methods=['GET', 'POST'])
```

Per semplicità, tutte le sale hanno **150 posti**.

Richiediamo il login: vedi sezione [ruoli](#)

Come al solito, abbiamo il metodo **GET** e **POST**.

- **GET:** renderizza il template passando un dizionario contenente le sale
- **POST:** gestisce il form di creazione sala

Route: /aggiungi_proiezione

Con questa route un **amministratore / manager** può **aggiungere le proiezioni per un determinato film**.

```
@app.route('/aggiungi_proiezione', methods=['GET', 'POST'])
```

Richiediamo il login: vedi sezione [ruoli](#)

Il metodo POST gestisce il form, prendendo i **dati** anche questa volta in modo **dinamico**: per lo stesso film, posso inserire più proiezioni alla volta.

⚠ **Il controllo per evitare conflitti di sala e orario non sono stati fatti**: assumiamo che sia responsabilità di un amministratore inserire i film in modo tale da non creare conflitti.

Route: /aggiungi_genere

Da questa route un **amministratore / manager** può aggiungere un **nuovo genere nel database**.

```
@app.route('/aggiungi_proiezione', methods=['GET', 'POST'])
```

Richiediamo il login: vedi sezione [ruoli](#)

Route: rimuovi_film

Tramite questa route, un **amministratore** può cancellare un film dal database.

L'eliminazione del film, causa l'eliminazione a cascata di tutte le tuple che riferiscono a tale film dalle tabelle [proiezioni](#), [attori](#), [registi](#), [genere_film](#).

```
@app.route('/rimuovi_film', methods=['GET', 'POST'])
```

Route: rimuovi_proiezione

Tramite questa route, un **amministratore** può cancellare una proiezione dal database.

```
@app.route('/rimuovi_proiezione', methods=['GET', 'POST'])
```

Se la **proiezione** era **prevista** per una **data futura** a quella odierna, allora bisogna **rimborsare** ai clienti i **biglietti** acquistati per quella proiezione.

Per fare questo, abbiamo creato un **TRIGGER**

Il **trigger** è definito nel file [create_database.py](#) ed è spiegato qui sopra, nella [sezione apposita](#)

User Route - File [user_route.py](#)

In questo file sono contenute **tutte le route per gli utenti**.

Importiamo il file query.

Creiamo il **Blueprint user_app**

```
user_app = Blueprint('user_app', __name__)
```

Route: /dashboard_account

Questa route permette ad un **cliente** di **visualizzare la pagina profilo di un utente: nome, cognome, mail e saldo** corrente e, se l'utente desidera, può **ricaricare il suo conto e/o modificare la sua password**.

Nella realtà, la ricarica avverrebbe con un pagamento, qui per semplicità faremo un semplice form inserendo la cifra da ricaricare.

Richiediamo il login: dobbiamo essere registrati per poter visionare il nostro profilo. ([ruoli](#))

```
@login_required
@app.route('/dashboard_account')
```


Route: /cambia_password

Tramite questa route, un **cliente** può cambiare la propria password.

Richiediamo il login: dobbiamo essere registrati per poter cambiare la nostra password. ([ruoli](#))

```
@login_required
@app.route('/cambia_password')
```

Route: /ricarica_saldo

Tramite questa route, un **cliente** può ricaricare il proprio saldo.

Richiediamo il login: dobbiamo essere registrati per poter ricaricare il nostro saldo. ([ruoli](#))

```
@login_required
@app.route('/ricarica_saldo')
```

Route: /prenota_biglietto

Tramite questa route, un **cliente** può prenotare uno o più biglietto/i.

Richiediamo il login: dobbiamo essere registrati per poter comprare biglietti. ([ruoli](#))

```
@login_required
@app.route('/prenota_biglietto/<id_pr>')
```

Anche qui, abbiamo bisogno di una **TRANSAZIONE**: vogliamo assolutamente **evitare** che più utenti **prenotino contemporaneamente** lo **stesso posto** per la **stessa proiezione**.

Grazie ad una **transazione**, garantiamo il corretto funzionamento della prenotazione.

Route: /le_mie_prenotazioni

In questa route, un **cliente** può **visualizzare** le sue **prenotazioni**.

```
@login_required
@app.route('/prenota_biglietto/<id_pr>')
```

Query - File [query.py](#)

Questo script python contiene tutte le funzioni ausiliarie, **tra cui tutte le query eseguite nel nostro progetto**.

Da questo file, importiamo il file sottostante: engine.py

Engine - File [engine.py](#)

Questo file contiene le configurazioni iniziali dell'applicazione: engine, metaData, Tabelle, Moduli importati

Introduzione e configuraione app

Dopo aver **importato tutte le librerie** necessarie

```
from flask import ...
from flask_bcrypt import ...
from sqlalchemy import ...
from sqlalchemy.sql import ...
from flask_login import ...
```

Inizializziamo l'applicazione e il **Bcrypt** (Spiegato successivamente) ed

```
app = Flask(__name__)  
bcrypt = Bcrypt(app)
```

E inizializziamo il **login manager**, settando anche la **secret_key** (Come consigliato dalla documentazione di [Flask_Login](#))

```
app.secret_key = b'f^iz\x05~\x1b\xaat\x07\x00\xb4Lf7\xa0'  
login_manager = LoginManager()  
login_manager.init_app(app)
```

Apriamo quattro engine collegandoci al database come: **Anonimo**, **cliente**, **manager** e **admin** e recuperiamo il MetaData e tutte le tabelle interne.

Vedi [ruoli](#) per riferimento alla **scelta degli engine**.

```
anonim_engine = create_engine("...")  
clienti_engine = create_engine("...")  
admin_engine = create_engine("...")  
manager_engine = create_engine("...")  
meta = MetaData(engine)  
meta.reflect()
```

Estraiamo dal `meta.Tables` le tabelle del database

```
utenti = meta.tables['utenti']  
posti = meta.tables['posti']  
proiezioni = meta.tables['proiezioni']  
...
```

Creiamo la **classe Python** che rappresenta un [Utente](#)

```
class Utente(UserMixin):  
    def __init__(self, email, password): #costruttore  
        self.email = email  
        self.password = password  
        self.is_admin = is_admin
```

e definiamo al suo interno un **metodo** che **ritorna l'id** (in questo caso, la mail, che è PrimaryKey)

```
def get_id(self): #metodo che restituisce l'id (in questo caso, la email)  
    return self.email
```

DESCRIZIONE QUERY

File `query.py`

Alcune sono state scritte in **Expression Language**, ma, per comodità e questioni di tempo, la maggior parte sono state scritte in **TextualSQL** utilizzando la sintassi del **DBMS PostgreSQL**

`generate_my_projection_dict()`

Query scritta in **Expression language**, corrisponde alla seguente query scritta in PostgreSQL:

```
SELECT f.titolo, pr.data, pr.ora_inizio, pr.sala, COUNT(*) as NumBiglietti
FROM posti po
JOIN proiezioni pr ON po.id_proiezione = pr.id_proiezione
JOIN film f ON pr.film = f.id_film
WHERE po.prenotato = current_user.email
AND pr.data >= current_date
AND pr.ora_inizio >= current_time
GROUP BY pr.id_proiezione
```

Essa si occupa di interrogare il database in modo tale da restituire tutte le prenotazioni eseguite dall'utente corrente.

`generate_all_film_next_projection`

Questa query, scritta in **TextualSQL**, si occupa di generare una lista contenente la proiezione più recente per ogni film

Questa richiesta ci torna utile sia nella home, che nella pagina "tutti i film"

`generate_all_projection_film(id)`

La seguente query ha lo scopo di estrapolare tutte le prossime proiezioni in programma per un determinato film, identificato dall'id preso in input dalla funzione python.

`generate_all_projection`

Simile alla precedente, la seguente query estrapola la lista di tutte le prossime proiezioni in programma.

Query ad uso statistico: `STAT_*`

Tutte le seguenti query sono espresse in **TextualSQL**.

Per **raccogliere alcune statistiche**, abbiamo eseguito le seguenti query.

(NB: vengono descritte solo quelle meno banali)

`STAT_numero_di_film`

Una semplice count sulla tabella film.

`STAT_film_con_piu_proiezioni`

Creiamo una view visualizzando il numero di proiezioni per ogni film.

Successivamente, estraiamo il film che ha il numero massimo di proiezioni.

`STAT_film_con_piu_posti_prenotati_totali`

Attuiamo lo stesso procedimento della query precedente, massimizzando il numero di posti totali.

`STAT_film_con_piu_incassi_totali`

Ancora una volta, creiamo una view per poi estrarre l'incasso massimo

`STAT_genere_con_piu_posti_prenotati`

Abbiamo bisogno di 4 tabelle: proiezioni, posti, film, genere_film.

Creiamo una view eseguendo 3 join e selezionando il genere, il relativo numero di posti totale e il totale di incassi.

Successivamente, da questa view estraiamo il massimo per numero di posti.

STAT_genere_con_più_incassi

Dalla view precedente, estraiamo il massimo per totale di incassi

STAT_numero_di_proiezioni_totale

Una semplice count sulla tabella proiezioni

STAT_proiezione_piu_prenotata

Dopo aver creato una view, viene massimizzato il numero di posti.

STAT_proiezione_con_incasso_piu_alto

Dalla view precedente, viene massimizzato l'incasso

STAT_orario_piu_prenotato

Creando una view contenente orario - numero di posti, è facile poi estrarre l'orario in cui ci sono state più prenotazioni.

STAT_attore_in_piu_film

Dalla view attore-numero_film estraiamo il massimo

STAT_regista_di_piu_film

Dalla view regista-numero_film estraiamo il massimo

STAT_numero_clienti_iscritti

Una semplice count dalla tabella utenti, escludendo admin e manager dal conteggio.

STAT_numero_posti_medio_prenotati_da_un_cliente

Creata la view per email-numero_prenotazioni, andiamo a calcolare la media per il numero di prenotazioni.

STAT_eta_media_utenti

Creiamo una view per email-età, dove l'età viene calcolata mediante la funzione

`(current_date - data_nascita) / 365.25`

La divisione per 365.25 ha lo scopo di trasformare il risultato in "anni", altrimenti il valore sarebbe espresso in giorni

Da questa view, poi, andiamo a calcolare la media di età.

STAT_numero_proiezioni_per_sala

Per ogni sala, andiamo a contare il numero di proiezioni eseguendo una semplice join, una count e raggruppando per numero sala.

STAT_durata_media_dei_film_per_sala

Creiamo una view che, per ogni sala, visualizzi il numero di film proiettati e la durata totale di tutti i film proiettati in quella sala.

Da questa view, andiamo a calcolare la media eseguendo, per ogni sala, la divisione `durata/num_film`

TEMPLATE HTML

Main Template

File [main_template.html](#)

File [home.html](#)

File [login.html](#)

Admin Template

File [home_gestione_sito.html](#)

File [aggiungi_admin.html](#)

File [aggiungi_film.html](#)

Questa pagina si appoggia ad un **File JavaScript**, [aggiungi_film.js](#) contenuto nella directory `static`

File [aggiungi_genere.html](#)

File [aggiungi_persona.html](#)

File [aggiungi_proiezione.html](#)

Questa pagina si appoggia ad un **File JavaScript**, [aggiungi_proiezione.js](#) contenuto nella directory `static`

File [riepilogo_sale.html](#)

File [rimuovi_film.html](#)

File [rimuovi_proiezioni.html](#)

Questa pagina si appoggia ad un **File JavaScript**, [rimuovi_proiezione.js](#) contenuto nella directory `static`

User Template

File [tutti_i_film.html](#)

File [cambia_password.html](#)

File [dashboard_account.html](#)

File [le_mie_prenotazioni.html](#)

File [prenota_biglietto.html](#)

Questa pagina si appoggia ad un **File JavaScript**, [prenota_biglietto.js](#) contenuto nella directory `static`

File [ricarica_saldo.html](#)

File [registrazione.html](#)

File [tutti_i_film.html](#)

DOCUMENTAZIONE

- [Flask](#)
- [Flask Login](#)
- [Python](#)
- [SQLAlchemy](#)
- [HTML](#)
- [Jinja](#)
- [JavaScript](#)
- [CSS](#)
- [PostgreSQL 12](#)