

# Design e Implementazione di un Database Relazionale con MySQL e Django

Alessandro Scodavolpe

January 2025

## 1 Introduzione

Nel modello relazionale il dato è rappresentato come una tupla che segue le regole dell'algebra relazionale e che viene salvata in memoria secondaria per la persistenza, mentre un linguaggio orientato ad oggetti rappresenta un dato o una collezione di dati come una struttura organizzata chiamata Classe o Oggetto, insieme a metodi associati alla classe/oggetto, la quale è gestita tramite Heap in memoria principale e non è possibile conservarne lo stato in maniera persistente come avviene con i DB. Inoltre il concetto di Ereditarietà non è presente nei Database Relazionali.

Un ORM (Object Relational Mappe) è un concetto di programmazione che prevede un layer di traduzione tra un sistema che utilizza un paradigma di programmazione orientata ad oggetti e un Relational Database Management System (RDBMS), consentendo un'integrazione tra queste due componenti. In questa maniera è possibile interagire con un RDBMS usando un linguaggio di programmazione orientato ad oggetti, poichè la logica di traduzione tra i due sistemi è gestita dal ORM.

Questa soluzione non è perfetta e presenta dei problemi come vedremo, a causa del fondamentale disallineamento tra le due rappresentazioni. I vantaggi principali nell'uso di un ORM per implementare e mantenere un Database Relazionale sono i seguenti:

1. Eseguire transazioni e interrogazioni di base su di un Database secondo una sintassi orientata ad oggetti e costrutti tipici di questo paradigma.
2. L'interazione con il Database viene effettuata nello stesso ambiente in cui si interagisce col resto del codice dell'applicazione che sfrutta tale Database, aumentando la velocità di sviluppo.
3. Gli ORM sono generalmente più sicuri perchè hanno funzioni di sicurezza di base per prevenire attacchi al Database (come una SQL Injection)

Tuttavia ci sono anche degli svantaggi:

1. La sintassi e le funzioni dell'ORM possono non essere intuitive e vanno studiate
2. Alcune Feature dei RDBMS come i Trigger non sono sempre supportate (come vedremo in seguito)

Per questo progetto si è fatto uso del ORM di Django per la costruzione e gestione di un database MySQL. Django è un python framework molto utilizzato per lo sviluppo di applicazioni web e con il suo ORM consente di mappare specifiche classi Python (chiamate "Models") a tabelle di un RDBMS supportato come MySQL. Sono presenti anche altre feature molto utili per lo sviluppo web come un web-server e un sito admin pre-configurato per amministrare il database tramite un interfaccia grafica intuitiva.

## 2 Database Design

### 2.1 Specifica dei Requisiti

Il database modella un servizio internet per lo streaming di serie TV, tale servizio permette agli utenti registrati di vedere episodi disponibili di serie TV, recensirli con un voto e un commento, seguire artisti e studi che producono le serie, creare playlist personalizzate di episodi e marciare come preferita una serie. Ogni entità modellata nel database possiederà un numero ID per identificarla univocamente.

Ogni Utente Registrato possiede email, password, nome utente, data di nascita e data di registrazione al servizio. Le date di registrazione e di nascita non sono alterabili una volta che un utente si registra. Email password e nome utente dovranno essere unici per ogni utente, inoltre le password dovranno essere di almeno 8 caratteri. Le Recensioni che un utente può lasciare riguarderanno ciascuna un singolo episodio, avranno un voto che potrà essere positivo o negativo e un commento opzionale.

Un utente potrà lasciare recensioni solo per episodi che ha visto.

Gli utenti possono quindi guardare Episodi delle serie disponibili, ogni Episodio ha un numero progressivo che indica l'ordine di visione degli episodi, un titolo, il numero della stagione di cui fa parte, la durata in minuti, la data di messa in onda, l'Artista che l'ha diretto, lo Studio che lo ha prodotto, la Serie a cui appartiene. Per ogni stagione vi potrà essere un singolo episodio che fungerà da finale. Ogni episodio avrà un rating calcolato secondo le seguenti regole:

Se vi sono almeno 10 voti, il rating viene calcolato come il rapporto tra numero di voti positivi e numero totale di voti moltiplicato per 10. In caso non vi siano voti sufficienti il rating non viene calcolato. Gli Artisti sono memorizzati nel database e possiedono nome e cognome.

Gli Studio hanno un nome unico, un indirizzo della loro sede principale e devono avere un Artista a capo dello studio.

I Franchise di cui fanno parte le Serie hanno un nome unico per ciascuno e il nome del detentore dei diritti per quel franchise.

Le Serie hanno un nome, un genere, un franchise di cui fanno parte, una classificazione per l'età consigliata di visione dei rispettivi episodi (Per tutti, da 14 anni in su o da 18 anni in su). Ciò significa che alcuni utenti potrebbero non poter vedere episodi di una determinata serie se non rientrano nei criteri di età stabiliti dalla sua classificazione.

Una playlist creata da un utente avrà degli episodi salvati ( 0 o più), un nome e una data di creazione.

Il sistema inoltre tiene traccia della cronologia degli episodi guardati per ciascun utente, memorizzando la data di visione di ogni episodio e l'utente che l'ha guardato. Le date dovranno essere coerenti tra entità (ad esempio gli utenti non possono vedere un episodio o inserirlo in una playlist prima che venga rilasciato), non ci possono essere date che superano quella odierna tranne che per episodi che devono ancora essere rilasciati.

### **2.1.1 Business Rules**

Le business rules individuate nella specifica dei requisiti sono le seguenti:

- Email, password e nome degli Utenti saranno diverse per ogni utente.
- Le password degli utenti devono avere una lunghezza minima di 8 caratteri.
- Le date di nascita e registrazione degli Utenti non possono essere cambiate.
- Se un Utente ha meno di 14 anni di età o 18 anni di età non potrà vedere serie classificate come rivolte ad un pubblico con età maggiore di 14 anni o di 18 anni rispettivamente.
- Un Utente potrà lasciare recensioni solo per Episodi che ha visto e solo una per Episodio.
- Ogni Episodio appartenente alla stessa stagione e Serie dovrà avere una numerazione unica.
- Solo per gli Episodi è ammesso che la data superi quella del giorno corrente.
- Non ci può essere più di un Episodio finale per ogni stagione di una Serie.

## **2.2 Progettazione Concettuale**

Di seguito viene illustrato lo schema E-R (Entità - Relazione) ricavato dalla specifica progettuale.

## **2.3 Progettazione Logica**

Traduzione dello schema E-R mostrato nella sezione precedente in elementi del modello Relazionale. Nella traduzione verso lo schema logico, gli identificatori

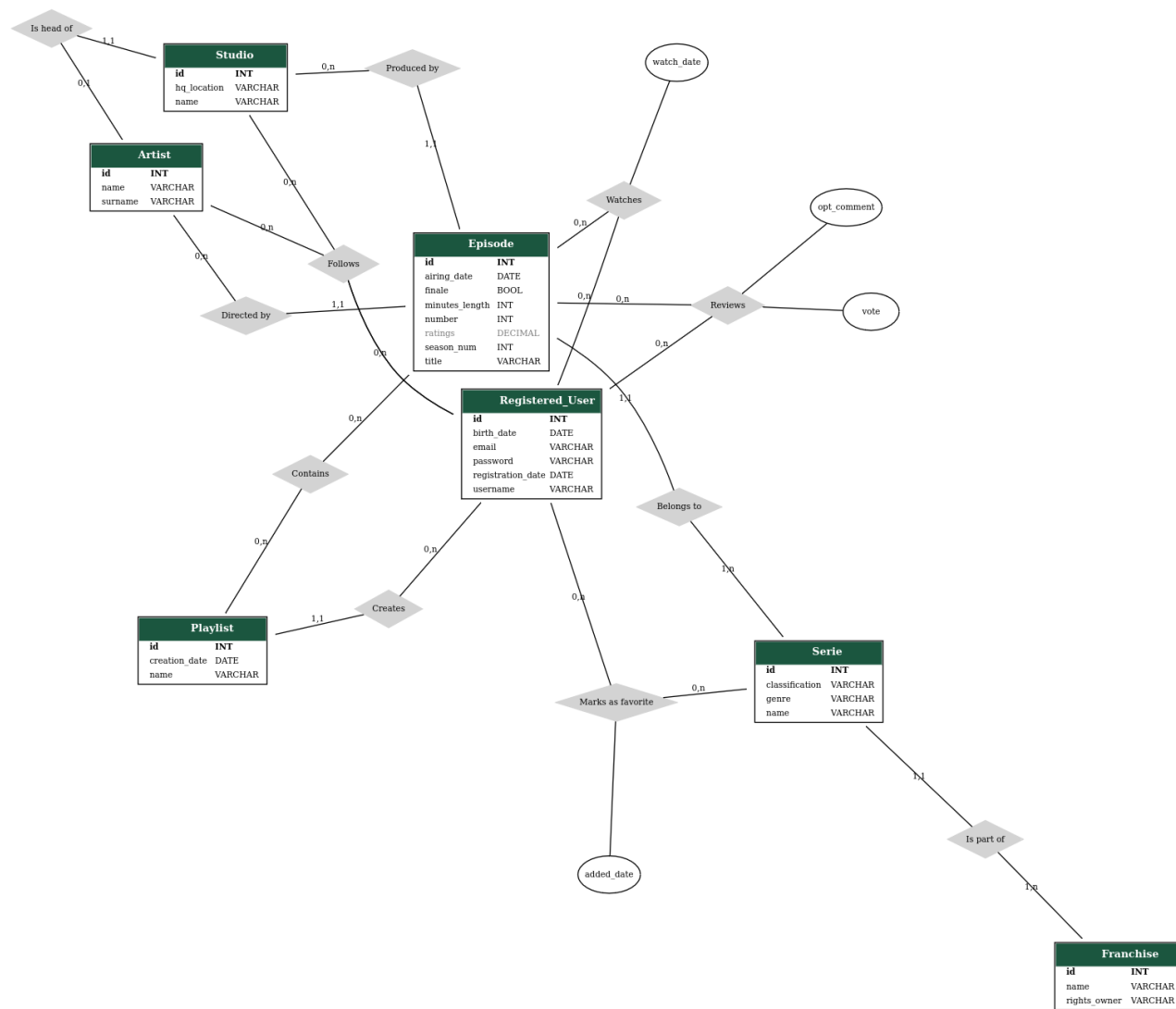


Figure 1: Schema Entità - Relazione del progetto

delle entità diventano le chiavi primarie (Primary Key) per le tabelle corrispondenti a quelle entità, gli identificatori esterni di un'entità rappresentano i vincoli di integrità referenziale di quell'entità e sono inseriti nella tabella corrispondente come chiavi esterne (Foreign Key)

Ogni entità dello schema è stata trasformata in una tabella contenente gli attributi indicati nello schema, ogni relazione è stata tradotta con le seguenti regole:

Traduzione della relazione Uno a Uno (Artist Studio) accorpando la relazione nell'entità Studio che ha cardinalità 1,1 con l'identificatore di Artist

Traduzione delle Relazioni Molti a Molti con Entità dedicata che ha come vincoli di integrità gli identificatori di entrambe le entità (tutte le relazioni del tipo Registered User Episode, Registered User Artist, Registered User Studio).

Traduzione delle Relazioni Uno a Molti con accorpamento della relazione nell'Entità con cardinalità massima più alta, tramite vincolo di integrità dell'altra Entità (Playlist Episode, Playlist Registered User, Episode Artist, Episode Studio).  
Elenco delle Tabelle ottenuto nella traduzione in schema logico:

- Registered User(user id, email, password, username, birth date, registration date)
- Artist(artist id, name, surname)
- Studio(studio id, artist id, studio name, hq location)
- Franchise(franchise id, franchise name, rights owner)
- Playlist(playlist id, user id, name, creation date)
- Episode(episode id, series id, studio id, artist id, number, season number, title, length, airing date, season finale, rating)
- Series(series id, franchise id, series name, genre, classification)
- User Follows Artist (artist id, user id)
- User Follows Studio (studio id, user id)
- User Reviews Episode (user id, episode id, vote, comment)
- User Favorite Series (user id, series id, added date)
- User Watches Episode(user id, episode id, watch date)
- Playlist Contains Episode (playlist id, episode id)

### 3 Implementazione del Database

Partendo dallo schema logico, si passa alla fase di implementazione effettiva del database. Questa fase può essere affrontata interagendo direttamente con il DBMS e scrivendo il codice SQL DDL (Data Definition Language) necessario per la creazione delle tabelle dello schema logico con gli attributi specificati con il tipo di dominio adatto (ad esempio DATE per un attributo che riguarda una data), le chiavi primarie ed esterne per il rispetto dei vincoli di integrità referenziale delle tabelle.

Utilizzando un ORM invece, è possibile scrivere del codice che utilizza sintassi e costrutti del paradigma ad oggetti e sarà l'ORM a gestire l'interazione con il database per la creazione delle tabelle con i relativi attributi e vincoli d'integrità, potendo così applicare i vantaggi del riuso di codice, della maggior sicurezza e della facilità con cui è possibile alterare il database cambiando poche linee di codice in caso di errore o cambiamenti di esigenze.

### 3.1 Implementazione tabelle con Django Models

Il framework Django include un componente ORM che utilizza delle speciali classi chiamate Django Models, su cui vengono mappate le corrispondenti tabelle del RDBMS con cui si vuole interagire.

Possiamo definire sottoclassi di Models, le quali rappresentano una Tabella nel DB e le istanze di ciascuna sottoclasse di Model rappresentano una riga di quella Tabella. I campi (fields) dei ciascun Model definiscono le colonne della Tabella corrispondente nel DB ed è possibile definire campi come valori enumerati, chiavi esterne che indicano una relazione con un altro Model, utilizzare gli argomenti opzionali per indicare valori unici e molto altro.

E' anche presente una funzionalità per la gestione delle relazioni Molti-a-Molti, che consente di specificarne una nei campi dei Models che si intende mettere in relazione, utilizzando argomenti opzionali per regolare come funziona tale relazione; l'ORM creerà automaticamente una tabella di Join per i due models, similmente al metodo di traduzione delle Relazioni usato per la Progettazione Logica. Per memorizzare dati extra nella tabella di Join è possibile specificare un model che fungerà da tabella per la relazione, mediante l'argomento opzionale "through". Si è scelto di sfruttare questa funzionalità per le relazioni Molti-a-Molti che richiedono informazioni aggiuntive come nel caso di Registered User Watches Episode.

Django tiene traccia del cambiamento dei Models e delle tabelle del DB per un determinato progetto (chiamato django app) con il concetto di **migration**. Per sincronizzare il DB con il contenuto dei models e dei migration file si utilizza il modulo manage.py e il comando migrate, che si occupa di connettersi al DBMS ed emettere il comando SQL adatto. E' anche possibile avere in output il codice SQL generato tramite il comando **sqlmigrate**.

Django fornisce anche un webserver per lo sviluppo e una admin page che consente di interagire tramite web GUI (Graphical User Interface) con il DBMS del progetto per effettuare operazioni CRUD (Create, Read, Update, Delete) sul database.

### 3.2 Query tramite Django ORM

In Django grazie alle API fornite per l'interazione col DB è possibile effettuare query su di esso utilizzando i riferimenti ai Models.

```

class Serie(models.Model):
    class GenreType(models.TextChoices):
        MUSICAL = 'MUS', 'Musical'
        THRILLER = 'THR', 'Thriller'
        COMEDY = 'COM', 'Comedy'
        ACTION = 'ACT', 'Action'
        DRAMA = 'DRM', 'Drama'
        HORROR = 'HOR', 'Horror'
        DOCUMENTARY = 'DOC', 'Documentary'
        NOT_SPECIFIED = '-', 'Not specified'

    class Classification(models.TextChoices):
        EVERYONE = 'T', 'For Everyone'
        NOT_UNDER_14 = '14+', 'Not for under 14'
        NOT_UNDER_18 = '18+', 'Mature content'

    name = models.CharField(max_length=50, unique=True)
    franchise = models.ForeignKey(Franchise, related_name="series", on_delete=models.CASCADE)
    genre = models.CharField(
        max_length=3,
        choices=GenreType.choices,
        default=GenreType.NOT_SPECIFIED
    )
    classification = models.CharField(
        max_length=3,
        choices=Classification.choices,
        default=Classification.EVERYONE
    )

    def __str__(self):
        return f'{self.name}'

class Episode(models.Model):
    number = models.PositiveIntegerField()
    title = models.CharField(max_length=50)
    season_num = models.PositiveIntegerField()
    serie = models.ForeignKey(Serie, related_name="episodes", on_delete=models.CASCADE)
    studio = models.ForeignKey(Studio, related_name="ep_produced", on_delete=models.SET_NULL, null=True) #
    director = models.ForeignKey(Artist, related_name="ep_directed", on_delete=models.SET_NULL, null=True) #
    minutes_length = models.PositiveIntegerField()
    airing_date = models.DateField()
    finale = models.BooleanField()
    ratings = models.DecimalField(max_digits=3, decimal_places=1, null=True, blank=True, default='NULL')

    def __str__(self):
        return f"E{self.number} S{self.season_num} {self.serie.name}"

```

Figure 2: Definizione delle Classi Models, si noti il campo ForeignKey; accedendo ad esso da un istanza di un Model (ad es. Serie) si ottiene il riferimento all'istanza della classe Model in relazione con la prima (ed es. Franchise). A livello di database questo campo corrisponde ad un vincolo d'integrità referenziale.

```

$ python manage.py sqlmigrate streamv 0001
System check identified some issues:

WARNINGS:
streamv.Studio.studio_head: (fields.E402) Setting unique=True on a ForeignKey has the same effect as using a OneToOneField.
HINT: ForeignKey(unique=True) is usually better served by a OneToOneField.

--
1. Create model Artist
CREATE TABLE "streamv_artist" ("id" bigint AUTO INCREMENT NOT NULL PRIMARY KEY, "name" varchar(25) NOT NULL, "surname" varchar(25) NOT NULL, "birth_date" date NOT NULL);

2. Create model Episode
CREATE TABLE "streamv_episode" ("id" bigint AUTO INCREMENT NOT NULL PRIMARY KEY, "number" integer UNSIGNED NOT NULL CHECK ("number" <= 8), "title" varchar(50) NOT NULL, "season_num" integer UNSIGNED NOT NULL CHECK ("season_num" <= 8), "minutes_length" integer UNSIGNED NOT NULL CHECK ("minutes_length" <= 60), "airing_date" date NOT NULL, "franchise" bigint NOT NULL, "ratings" numeric(3, 1) NOT NULL, "finale" boolean NOT NULL);

3. Create model Franchise
CREATE TABLE "streamv_franchise" ("id" bigint AUTO INCREMENT NOT NULL PRIMARY KEY, "name" varchar(50) NOT NULL UNIQUE, "rights_owner" varchar(50) NOT NULL, "creation_date" date NOT NULL);

4. Create model RegisteredUser
CREATE TABLE "streamv_registered_user" ("id" bigint AUTO INCREMENT NOT NULL PRIMARY KEY, "email" varchar(50) NOT NULL UNIQUE, "password" varchar(128) NOT NULL UNIQUE, "username" varchar(50) NOT NULL UNIQUE, "birth_date" date NOT NULL);

5. Create model Serie
CREATE TABLE "streamv_serie" ("id" bigint AUTO INCREMENT NOT NULL PRIMARY KEY, "name" varchar(50) NOT NULL UNIQUE, "genre" varchar(3) NOT NULL, "classification" varchar(3) NOT NULL, "franchise_id" bigint NOT NULL);

```

Figure 3: Parte dell'output generato dall'opzione sqlmigrate per la prima migration del progetto in cui vengono create le tabelle corrispondenti ai model definiti nel file models.py

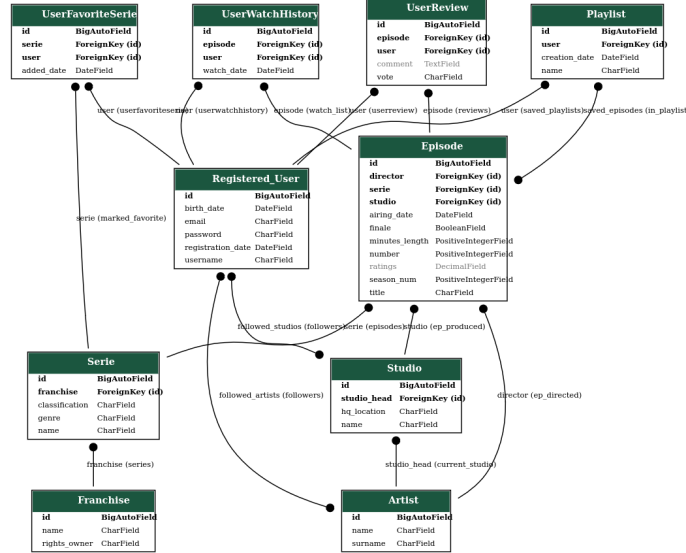


Figure 4: Schema dei classi Models implementate nel file models.py: le linee indicano le relazioni descritte dai Vincoli d'integrità referenziale di ciascun modello e i punti neri indicano una cardinalità multipla a un capo della relazione. Le etichette sulle relazioni indicano il nome del campo ForeignKey a cui si può accedere da entrambi i capi della relazione (tra parentesi il nome per accedere dal senso opposto). Notare che non sono presenti Model per le relazioni Molti a Molti che non richiedevano informazioni aggiuntive, in quanto le tabelle relative sono create direttamente nel database da Django

```

ALTER TABLE 'streamtv_registered_user' ADD CONSTRAINT 'password_min_length' CHECK (CHAR_LENGTH('password') >= 8);
... Create constraint unique_episode_per_season_per_serie on model episode
ALTER TABLE 'streamtv_episode' ADD CONSTRAINT 'unique_episode_per_season_per_serie' UNIQUE ('serie_id', 'season_num', 'number');
ALTER TABLE 'streamtv_episode' ADD CONSTRAINT 'streamtv_episode_director_id_fk_streamtv_artist_id' FOREIGN KEY ('director_id') REFERENCES 'streamtv_artist' ('id');
ALTER TABLE 'streamtv_serie' ADD CONSTRAINT 'streamtv_serie_franchise_id_fk_streamtv_franchise_id' FOREIGN KEY ('franchise_id') REFERENCES 'streamtv_franchise' ('id');
ALTER TABLE 'streamtv_userwatchhistory' ADD CONSTRAINT 'streamtv_userwatchhistory_episode_id_fk_streamtv_episode_id' FOREIGN KEY ('episode_id') REFERENCES 'streamtv_episode' ('id');
ALTER TABLE 'streamtv_userwatchhistory' ADD CONSTRAINT 'streamtv_userwatchhistory_user_id_fk_streamtv_registered_user_id' FOREIGN KEY ('user_id') REFERENCES 'streamtv_registered_user' ('id');
ALTER TABLE 'streamtv_userreview' ADD CONSTRAINT 'streamtv_userreview_episode_id_fk_streamtv_episode_id' FOREIGN KEY ('episode_id') REFERENCES 'streamtv_episode' ('id');
ALTER TABLE 'streamtv_userreview' ADD CONSTRAINT 'streamtv_userreview_user_id_fk_streamtv_registered_user_id' FOREIGN KEY ('user_id') REFERENCES 'streamtv_registered_user' ('id');
ALTER TABLE 'streamtv_userfavoriteSerie' ADD CONSTRAINT 'streamtv_userfavoriteSerie_serie_id_fk_streamtv_serie_id' FOREIGN KEY ('serie_id') REFERENCES 'streamtv_serie' ('id');
ALTER TABLE 'streamtv_userfavoriteSerie' ADD CONSTRAINT 'streamtv_userfavoriteSerie_user_id_fk_streamtv_registered_user_id' FOREIGN KEY ('user_id') REFERENCES 'streamtv_registered_user' ('id');
ALTER TABLE 'streamtv_studio' ADD CONSTRAINT 'streamtv_studio_head_id_fk_streamtv_artist_id' FOREIGN KEY ('studio_head_id') REFERENCES 'streamtv_artist' ('id');
ALTER TABLE 'streamtv_registered_user_followed_artists' ADD CONSTRAINT 'streamtv_registered_user_registered_user_id_fk_streamtv_registered_user_id' FOREIGN KEY ('registered_user_id') REFERENCES 'streamtv_registered_user' ('id');
ALTER TABLE 'streamtv_registered_user_followed_artists' ADD CONSTRAINT 'streamtv_registered_user_registered_user_id_fk_streamtv_registered_user_id' FOREIGN KEY ('registered_user_id') REFERENCES 'streamtv_registered_user' ('id');
ALTER TABLE 'streamtv_registered_user_followed_studios' ADD CONSTRAINT 'streamtv_registered_user_registered_user_id_fk_streamtv_registered_user_id' FOREIGN KEY ('registered_user_id') REFERENCES 'streamtv_registered_user' ('id');
ALTER TABLE 'streamtv_registered_user_followed_studios' ADD CONSTRAINT 'streamtv_registered_user_registered_user_id_fk_streamtv_registered_user_id' FOREIGN KEY ('registered_user_id') REFERENCES 'streamtv_registered_user' ('id');
ALTER TABLE 'streamtv_registered_user_followed_studios' ADD CONSTRAINT 'streamtv_registered_user_registered_user_id_fk_streamtv_registered_user_id' FOREIGN KEY ('registered_user_id') REFERENCES 'streamtv_registered_user' ('id');
ALTER TABLE 'streamtv_playlist' ADD CONSTRAINT 'streamtv_playlist_saved_playlist_id_fk_streamtv_playlist_id' FOREIGN KEY ('playlist_id') REFERENCES 'streamtv_playlist' ('id');
ALTER TABLE 'streamtv_playlist_saved_episodes' ADD CONSTRAINT 'streamtv_playlist_saved_episodes_episode_id_fk_streamtv_episode_id' FOREIGN KEY ('episode_id') REFERENCES 'streamtv_episode' ('id');
ALTER TABLE 'streamtv_playlist_saved_episodes' ADD CONSTRAINT 'streamtv_playlist_saved_episodes_episode_id_fk_streamtv_episode_id' FOREIGN KEY ('episode_id') REFERENCES 'streamtv_episode' ('id');

```

Figure 5: Parte finale dell'output dell'opzione sqlmigrate, dopo aver creato le tabelle vengono inseriti i vincoli unique e d'integrità referenziale tra le tabelle.



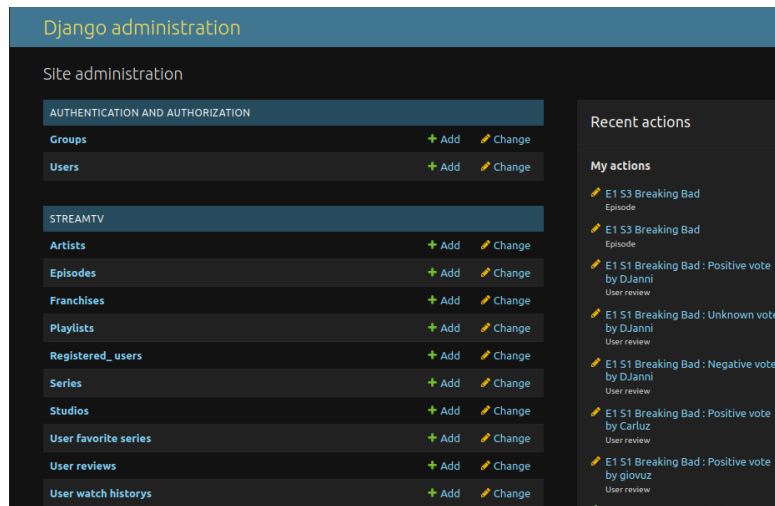


Figure 6: Una volta loggato come admin, un utente è indirizzato alla pagina admin index contenente informazioni sull'attività recente e con link alle pagine per accesso alle interfacce (incluse da Django) di gestione delle singole Model Class, specificate nel file admin.py

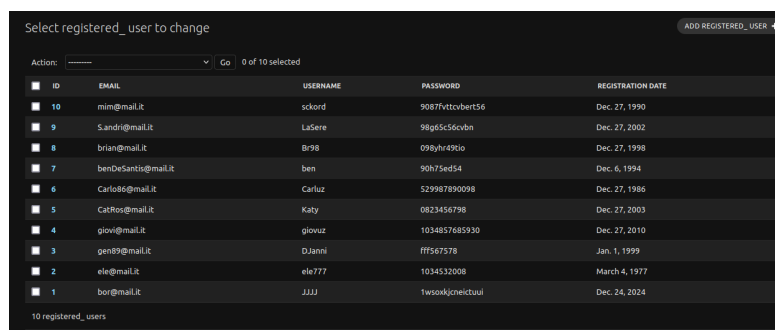
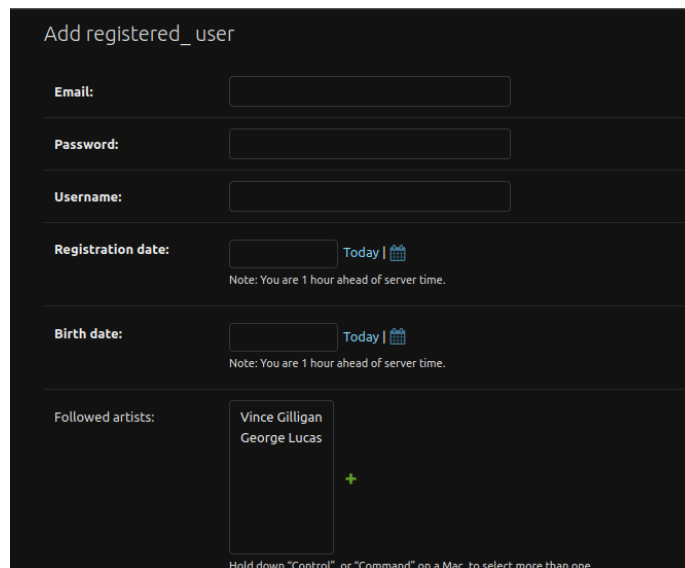


Figure 7: Pagina contenente le entry della tabella RegisteredUser, è possibile aggiungere, modificare o eliminare entry.





Add registered\_user

Email:

Password:


Username:

Registration date:  Today |   
Note: You are 1 hour ahead of server time.

Birth date:  Today |   
Note: You are 1 hour ahead of server time.

Followed artists: 

Vince Gilligan  
George Lucas



Hold down "Control", or "Command" on a Mac, to select more than one.

Figure 8: Un esempio di interfaccia auto-generata per aggiungere un istanza del Django Model `RegisteredUser` con form adatti per ciascun tipo di dato da inserire, questi form vengono poi validati clickando sul bottone "Save" e se superano la validazione del server, Django interagirà con il DBMS per sincronizzare le informazioni sulla tabella. A questo livello vengono invocati i metodi `clean()` per la pulizia dei dati dei form, vengono poi validate le informazioni immesse rispetto ai constraint e infine viene invocato `save()` per la sincronizzazione con il database.

Come illustrato precedentemente ciascuna sottoclasse di Models rappresenta una tabella del DB e le istanze di quel Model rappresentano una riga della tabella corrispondente. Quando istanziamo un Model l'ORM genera un SQL statement del tipo INSERT sulla Tabella mappata da quel Model. Per istanziare un modello si utilizza il suo metodo *save()*

Quando effettuiamo dei cambiamenti su un istanza del modello, ad esempio alteriamo il valore di uno dei campi presenti o cambiamo il riferimento ad un altro modello in un campo ForeignKey, per sincronizzare la riga corrispondente nel DB si utilizza nuovamente il metodo *save()*; contestualmente Django effettua un SQL statement di tipo UPDATE su quella riga.

Per recuperare le istanze di un modello, Django utilizza i metodi di un'interfaccia *Manager* detta *objects* presente per ciascuna sottoclasse di Models e accessibile direttamente dal riferimento alla sottoclasse con *NomeClasse.objects*. Il Manager per ciascuna classe contiene metodi che restituiscono **QuerySet**; si tratta di insiemi di oggetti corrispondenti alle righe del DB. Da un QuerySet si possono poi applicare filtri per ottenere specifiche istanze di un Model. Dal punto di vista del Database, un metodo che restituisce un QuerySet non corrisponde a una query nel DB.

Le query vengono effettuate dall'ORM solo in alcuni casi come nel caso di un iterazione sul QuerySet, in tal caso Django effettua uno statement SELECT e i filtri applicati ad un QuerySet corrispondono alle clausole WHERE.

### 3.3 Implementazione Business Rules

Alcune Business Rules richieste dalla specifica sono state implementate come vincoli utilizzando le clausole CHECK Constraints o UNIQUE Constraint fornite dal linguaggio SQL.

Queste clausole consentono di fornire una condizione che deve risultare vera per ogni entry della tabella su cui sono presenti. Ad esempio per la regola di lunghezza minima delle password si è ricorso ad un CHECK Constraint sulla tabella RegisteredUser.

Django supporta CHECK e UNIQUE Constraints con i metodi della classe Model omonimi, a cui bisogna fornire il vincolo di CHECK tramite espressione Booleana o tramite costrutto Q(), che rappresenta un'espressione SQL. Questi vincoli si possono impostare tramite la classe interna a Models chiamata *Meta* con cui è possibile specificare varie opzioni tramite campi specifici.

Tuttavia non tutti i Vincoli sono stati implementati tramite Constraints, poichè richiedevano Query e confronti tra valori di Tabelle diverse, funzione per cui sono adatti i Trigger SQL.

Un'alternativa all'uso dei Trigger per imporre dei vincoli è l'utilizzo del metodo *save()* o di *clean()* della Classe Model in Django, di cui si può fare overriding nelle sottoclassi per effettuare controlli personalizzati sull'istanza del modello prima che avvenga l'interazione col DB (per il metodo *save()*) o prima

del submit di un form della pagina di admin (nel caso del metodo *clean()* ). Pur sfruttando questa possibilità, implementandola nei modelli del progetto, questo controllo può essere sorpassato interagendo direttamente con il DB. Grazie all'utilizzo dei Trigger invece ci sarà sempre un controllo sull'integrità dei dati quando si interagisce con le tabelle.

I Trigger SQL sono costruiti con cui è possibile eseguire comandi e procedure SQL in maniera automatica su una certa Tabella in seguito ad eventi nel database come l'inserimento di una nuova riga o l'update di una riga già presente.

Sfortunatamente L'ORM di Django non supporta i Trigger, pertanto sono stati scritti direttamente in linguaggio SQL e applicati al database tramite uno script. Questo problema rappresenta uno degli svantaggi principali nell'uso di un ORM ma si può ovviare semplicemente interagendo direttamente con il DB.

I Trigger utilizzati controllano che ad ogni inserimento/update i dati siano coerenti con il. Per la riusabilità del codice SQL sono state utilizzate anche le Procedure SQL, le quali vengono chiamate all'interno del Trigger.

Ad esempio la procedura `VerifyAcceptableDate` accetta due dati come parametri in ingresso e controlla che tra la prima data fornita non sia maggiore della seconda e in caso affermativo viene sollevato un errore e ritornato un SQL STATE. Questa procedura è invocata ogniqualvolta viene tentato l'inserimento o l'update su di una tabella che include una data tra le colonne (tranne nel caso della tabella per Episode) così da impedire che venga violata l'integrità dei dati per quella tabella. Elenco completo di Procedure inserite nel DB:

- **VerifyAcceptableDate** confronta due date fornite come parametri in ingresso, se la prima supera la seconda viene sollevato un Errore.
- **VerifyUserWatchedEpisode** controlla che nella tabella `UserWatchHistory` sia presente una riga che associa un episodio ad un utente, in caso contrario solleva un Errore.
- **DeleteUserReview** per evitare problemi di integrità in cui utenti risultano aver recensito episodi che non hanno visto, questa procedura elimina righe in `UserReview` se l'utente non risulta aver visto l'episodio recensito.
- **EnforceContentRestriction** verifica che un utente abbia un'età adatta per vedere l'Episodio di una Serie (ovvero che una riga che associa quell'Episodio a quell'Utente sia inseribile in `UserWatchHistory`).

Scopo dei Trigger implementati è quello di:

- Impedire che un Utente rilasci una recensione ad un Episodio senza averlo visto (senza una riga che associa un `RegisteredUser` ad un Episode nella tabella `UserWatchHistory`)
- Impedire che un Utente possa vedere un episodio di una serie classificata come "14+" o "18+" se la sua età è inferiore a quanto indicato

```
class Meta:
    constraints = [
        models.CheckConstraint(
            check=Q(password_length_gte=8),
            name="password_min_length")
    ]
```

Figure 9: Classe Meta interna a RegisteredUser in cui è specificato il CHECK constraint per la lunghezza delle password.

```
| Table | Create Table |
+-----+-----+
| streamtv_registered_user | CREATE TABLE `streamtv_registered_user` (
  `id` bigint NOT NULL AUTO INCREMENT,
  `email` varchar(50) NOT NULL,
  `password` varchar(128) NOT NULL,
  `username` varchar(50) NOT NULL,
  `registration_date` date NOT NULL,
  `birth_date` date NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `email` (`email`),
  UNIQUE KEY `password` (`password`),
  UNIQUE KEY `username` (`username`),
  CONSTRAINT `password_min_length` CHECK ((char_length(`password`) >= 8))
) ENGINE=InnoDB AUTO INCREMENT=11 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci |
```

Figure 10: Output del comando SHOW CREATE TABLE per Registered User su MySQL client, il constraint per la tabella è stato applicato automaticamente da Django dopo la relativa migration

- Impedire che più Episodi di una Serie con stesso numero di stagione siano Finali di quella stagione
- Impedire che una data violi le business rules o non sia coerente con date correlate (es. la data di nascita di un utente deve essere inferiore alla data di registrazione, un utente non può aver visto un episodio che non è ancora stato rilasciato).  
Per tale motivo è stato scelto di impedire la modifica delle date relative a date di iscrizione, date di nascita di un Utente, date di uscita di un Episodio.
- Calcolare il rating di un Episodio secondo la business rule relativa e aggiornarlo in caso di cambiamento nei dati.

```

verifyUserWatchedEpisode = """CREATE PROCEDURE `verifyUserWatchedEpisode` (
    IN new_user_id INT,
    IN new_episode_id INT
)
BEGIN
    DECLARE watchedEpisode INT;
    SELECT COUNT(episode_id)
    INTO watchedEpisode
    FROM streamtv_userwatchhistory
    WHERE user_id = new_user_id
    AND episode_id = new_episode_id;
    IF watchedEpisode < 1
    THEN SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Users can review only watched episodes';
    END IF;
END"""
crt_procedures.append(verifyUserWatchedEpisode)

deleteUserReview = """CREATE PROCEDURE `deleteUserReview` (
    IN old_user_id INT,
    IN old_episode_id INT
)
BEGIN
    DECLARE watchedEpisode INT;
    SELECT COUNT(episode_id)
    INTO watchedEpisode
    FROM streamtv_userwatchhistory
    WHERE user_id = old_user_id
    AND episode_id = old_episode_id;
    IF watchedEpisode < 1
    THEN DELETE FROM streamtv_userreview
    WHERE user_id = old_user_id
    AND episode_id = old_episode_id;
    END IF;
END"""
crt_procedures.append(deleteUserReview)

```

Figure 11: Parte dello script file per la creazione di procedure e trigger sul DB: Definizione delle procedure VerifyAcceptableDate e DeleteUserReview.

```

before_playlist_episode_date_insert = """CREATE TRIGGER before_playlist_episode_date_insert
BEFORE INSERT
ON streamtv_playlist_saved_episodes
FOR EACH ROW
BEGIN
    DECLARE episodeDate DATE;
    SELECT airing_date
    INTO episodeDate
    FROM streamtv_episode
    WHERE id = new_episode_id;
    CALL verifyAcceptableDate(episodeDate, CURDATE());
END"""
crt_triggers.append(before_playlist_episode_date_insert)

before_registered_user_date_insert = """CREATE TRIGGER before_registered_user_date_insert
BEFORE INSERT
ON streamtv_registered_user
FOR EACH ROW
BEGIN
    CALL verifyAcceptableDate(new.birth_date, CURDATE());
    CALL verifyAcceptableDate(new.registration_date, CURDATE());
    CALL verifyAcceptableDate(new.birth_date, new.registration_date);
END"""
crt_triggers.append(before_registered_user_date_insert)

before_registered_user_date_update = """CREATE TRIGGER before_registered_user_date_update
BEFORE UPDATE
ON streamtv_registered_user
FOR EACH ROW
BEGIN
    IF (old.birth_date <> new.birth_date
    OR old.registration_date <> new.registration_date)
    THEN SIGNAL SQLSTATE '48000'
        SET MESSAGE_TEXT = 'user birth date and registration date cannot be changed';
    END IF;
END"""
crt_triggers.append(before_registered_user_date_update)

```

Figure 12: Parte dello script file per la creazione di procedure e trigger sul DB: Definizione dei trigger che invocano la procedura di verifica delle date per le tabelle Playlist e Registered User

## 4 Testing del Database

### 4.1 Testing su Django Admin

Di seguito sono illustrati degli esempi di testing dei vincoli tramite la Admin Page:

The screenshot shows a web form titled "Add user watch history". At the top, a red-bordered box contains the text "Please correct the errors below." Below this, there are three input fields, each with a red border and a red error message above it: "This field is required." The fields are: "User:" (a dropdown menu), "Episode:" (a dropdown menu), and "Watch date:" (a date input field). To the right of the "Watch date:" field, there is a "Today" button and a calendar icon. Below the date field, a note says "Note: You are 1 hour ahead of server time." At the bottom of the form, there are three buttons: "SAVE", "Save and add another", and "Save and continue editing".

Figure 13: Tentativo di salvare un modello con valori dei campi non specificati

The screenshot shows the same "Add user watch history" form. The "User:" dropdown is now set to "utente 1:JJJJ" and the "Episode:" dropdown is set to "E2 S1 Better Call Saul". A red error message above the "User:" field reads: "Invalid user: user not old enough to see episodes of this serie". The "Watch date:" field is now set to "2025-02-10". The "Today" button and calendar icon are still present. The note "Note: You are 1 hour ahead of server time." is also visible. The same three buttons ("SAVE", "Save and add another", "Save and continue editing") are at the bottom.

Figure 14: Inserimento di un Utente e di un Episodio non compatibili

## 4.2 Testing con UnitTest

Django supporta la scrittura ed esecuzione di UnitTest, un modulo python che consente di organizzare test in Classi.

Durante l'esecuzione di un test, Django costruisce un database di testing su cui svolgerà le operazioni previste dai test per i modelli. Sono stati implementati due file di testing, eseguiti entrambi con il comando *manage.py test*, nel primo vengono testati i Models del file models.py mentre nel secondo vengono testati Trigger e Procedure del database. Per poter testare Trigger e Procedure implementati è stato necessario importare lo script per crearli nel database di test ed

The screenshot shows a web form titled "Change user review" for the episode "E2 S1 Better Call Saul". The form contains the following elements:

- Title:** E2 S1 Better Call Saul : Positive vote by Carluz
- Error Message:** A red-bordered box with the text "Please correct the error below."
- Vote:** A dropdown menu currently set to "Positive vote".
- User:** A dropdown menu showing "utente 6: Carluz". Above this dropdown is a red error message: "Only users that watched an episode can review it". To the right of the dropdown are icons for edit (pencil), add (plus), and view (eye).
- Episode:** A dropdown menu showing "E2 S1 Better Call Saul". To its right are the same edit, add, and view icons.
- Comment:** A large, empty text area.
- Buttons:** At the bottom, there are three buttons: "SAVE", "Save and add another", and "Save and continue editing".

Figure 15: Inserimento di una recensione di un Episodio non visto da parte di un Utente

eseguirlo nel metodo *setUp()* che viene invocato prima di ogni metodo di test.



Figure 16: Inserimento di una seconda recensione con stessa coppia Utente - Episodio

```
mysql> INSERT INTO streamtv_registered_user VALUES (123456, "sdf", "dfgjrdifn48", "curnvru", "2025-01-01", "2026-01-01");
ERROR 1644 (45000): specified date is not acceptable
mysql> INSERT INTO streamtv_registered_user VALUES (123456, "sdf", "dfgjrdifn48", "curnvru", "2028-01-01", "2026-01-01");
ERROR 1644 (45000): specified date is not acceptable
mysql> INSERT INTO streamtv_registered_user VALUES (123456, "sdf", "dfgjrdifn48", "curnvru", "1999-01-01", "2026-01-01");
ERROR 1644 (45000): specified date is not acceptable
mysql> INSERT INTO streamtv_registered_user VALUES (123456, "sdf", "dfgjrdifn48", "curnvru", "2000-01-01", "2026-01-01");
ERROR 1644 (45000): specified date is not acceptable
mysql> INSERT INTO streamtv_registered_user VALUES (123456, "sdf", "dfgjrdifn48", "curnvru", "2024-01-01", "2000-01-01");
Query OK, 1 row affected (0.01 sec)

mysql>
```

Figure 17: Esempio di utilizzo dei Trigger durante l'inserimento di valori di date non ammessi nella Tabella Registered User

```
from django.test import TransactionTestCase, TestCase
from django.core.exceptions import ValidationError
from django.db import IntegrityError, OperationalError
from models import *
from datetime import datetime, date
# Create your tests here.

class RegUserModelTest(TransactionTestCase):

    def setUp(self):
        self.user = RegisteredUser(email="ale95@gmail.com", password="asdghjkl", username="ale95", registration_date=date(1995, 3, 19), birth_date=date(1985, 3, 1))

    def test_clean_method(self):
        self.user.clean() # the date is acceptable
        self.user.registration_date = date(2000, 3, 19)
        self.assertEqual(self.user.registration_date, date(2000, 3, 19))
        with self.assertRaises(ValidationError):
            self.user.clean() # the date is a future date (not acceptable)

    def test_str_method(self):
        self.assertEqual(str(self.user), f"utente ({self.user.id}: ale95")
```

Figure 18: Esempio di codice utilizzato per testare il modello Registered User

