**STRUTTURA PROGETTO**  
Cartella backend   
  
~~Models  
Controllers  
Middleware  
Routes~~  
Passport  
~~Utils  
Config~~

~~.env~~.gitignore  
package.json  
package-lock.json

Cartella frontend

app.js  
index.html  
style.css

**Un flusso end-to-end (dalla pagina alla tabella)**

1. Il FE fa login → riceve access + refresh.
2. Il FE chiama GET /api/movies?title=matrix con Authorization: Bearer <access>.
3. verifyToken valida il token, il controller esegue la query Sequelize su movies e risponde.
4. Se l’access scade, il FE chiama POST /auth/refresh-token con il refresh, ottiene un nuovo access, e ripete la richiesta fallita.
5. Se l’utente fa logout, il refresh viene revocato (non potrà più generare access).

 Ogni router:

1. Riceve la richiesta.
2. Fa passare la richiesta dentro i **middleware** (es. verifyToken, requireRole).
3. Chiama il **controller** giusto (la funzione che fa “il lavoro”).

 I controller poi parlano con:

* **Sequelize** (modelli User, Movie, RefreshToken) → DB locale.
* **TMDB** (con fetch) → dati esterni.
* **JWT/bcrypt** → sicurezza.

**Config:** contiene il file di configurazione del db, le cui variabili in chiaro sono dentro .env  
  
**Sequelize** è un **ORM** (Object-Relational Mapper) per Node.js: ti permette di lavorare con il database (MySQL nel tuo caso) usando **oggetti e metodi JS** invece di scrivere SQL a mano. Con Sequelize:

* **Definisci modelli** (User, Movie, …) che mappano tabelle e colonne.
* **Esegui query** con metodi (es. findAll, create, update) invece di SELECT/INSERT/UPDATE.
* **Gestisci relazioni** (1–N, N–N) con hasMany, belongsTo, ecc.
* **Configuri la connessione** al DB una volta sola e poi la riusi ovunque.

**// Configurazione della connessione al Database**

export const sequelize = new Sequelize(

process.env.DB\_NAME, // nome DB

process.env.DB\_USER, // utente

process.env.DB\_PASSWORD, // password

{

host: process.env.DB\_HOST,

port: process.env.DB\_PORT || 3306, // porta

dialect: process.env.DB\_DIALECT || 'mysql',

logging: process.env.LOG\_LEVEL === 'debug' ? console.log : false,

}

);

**Nota:** logging: se LOG\_LEVEL=debug, Sequelize stampa **tutte le query SQL** su console; altrimenti è silenzioso.

**dotenv** carica il contenuto del file **.env** dentro process.env all’avvio dell’app.  
Questo ti permette di **tenere le configurazioni fuori dal codice** (porte, credenziali, secret, token, ecc.) e di **non committare** in Git valori sensibili. Cambi ambiente? Cambi file .env (o variabili del sistema) senza toccare il codice.

**Come si incastrano assieme**

1. dotenv.config() legge le variabili dal tuo .env.
2. Con quei valori, **costruisci l’istanza Sequelize** (new Sequelize(...)).
3. Altrove (modelli/rotte) **importi sequelize** per definire tabelle e fare query.
4. All’avvio, sequelize.authenticate() verifica la connessione e sequelize.sync() sincronizza lo schema.

Un **JWT** (JSON Web Token) è un **token firmato** che rappresenta in modo compatto alcune informazioni (le *claim*) sull’utente o sulla sessione e che il client presenta ad ogni richiesta;  
il server lo verifica e decide l’accesso senza consultare una sessione (autenticazione stateless).

**Cos’è un JWT**

È una stringa base64 composta da **3 parti**:  
header.payload.signature

* **Header**: algoritmo e tipo, es. { "alg": "HS256", "typ": "JWT" }
* **Payload** (*claim*): dati come sub (id utente), exp (scadenza), roles, ecc.
* **Signature**: firma crittografica del header.payload con un **segreto** (HS256) o **chiave privata** (RS256/ECDSA). Serve a garantire **integrità** e **autenticità** (non è cifrato).

Importante: firmato ≠ cifrato. Chiunque possa leggere il token vede il payload; per segreti/sensibili usa cifratura esterna o non metterli nel token.

**Come si usa in un’API**

* Alla login generi un **access token** a vita breve (es. 10–30 min) e, spesso, un **refresh token** a vita più lunga (giorni).
* Il client invia l’access token in Authorization: Bearer <token> (oppure in cookie HttpOnly).
* Il server **verifica** la firma e le claim ad ogni richiesta. Non c’è sessione server-side: è *stateless*.

**Cosa fa jwtHerlper della cartella Utils**

* **generateAccessToken(user)**  
  Crea un **access token** (vita breve) con payload { id, role }.  
  Serve per autenticare ogni richiesta alle rotte protette.  
  Scadenza: JWT\_EXPIRES\_IN (es. 1h).
* **generateRefreshToken(user)**  
  Crea un **refresh token** (vita più lunga) con payload { id } minimo.  
  Serve solo per chiedere un nuovo access token quando scade.  
  Scadenza: REFRESH\_TOKEN\_EXPIRES\_IN (es. 7d).

Entrambi usano jwt.sign(...) con la chiave process.env.JWT\_SECRET.

**Come interagisce col resto**

* Viene usato nei **controller auth**:
  + loginUser: genera accessToken + refreshToken; salva il refresh nel DB.
  + refreshUserToken: dopo aver verificato il refresh, genera **nuovo access**.
* Il **middleware verifyToken** verificherà l’**access token** sulle rotte protette.

**middleware di autorizzazione**. Sono “posti di blocco” che si mettono **prima** dei controller nelle route.

**Cosa fa verifyToken (in pratica)**

* **Legge** l’header Authorization: Bearer <token>.
* **Verifica** il JWT con jwt.verify(token, JWT\_SECRET).
* **Carica** l’utente dal DB con User.findByPk(decoded.id).
* Se tutto ok, mette l’utente in req.user e chiama next().  
  Da quel momento i controller possono fidarsi di req.user.

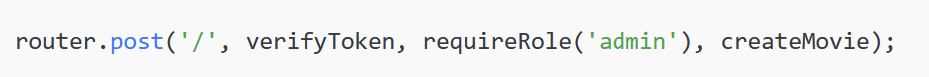
**Dove si usa:** nelle route protette, es:



**Perché è utile:** evita di riscrivere la logica di controllo in ogni controller; garantisce che chi arriva lì sia autenticato.  
  
  
**Cosa fa requireRole(role)**

* **Controlla** che req.user esista (quindi deve venire **dopo** verifyToken).
* **Confronta** req.user.role con il ruolo richiesto.
* Se non combacia → 403 Accesso negato.

**Dove si usa:** per proteggere operazioni admin:

  
  
**Come interagiscono con il resto**

* Le **route** montano i middleware: verifyToken → (opzionale) requireRole → **controller**.
* I **controller** leggono req.user settato da verifyToken (es. getUserProfile).
* Con i token emessi da utils/jwtHelper.js, il **client** chiama le route protette; se l’access scade, usa /auth/refresh-token e riprova.

**Model** contiene User, Movie e RefreshToken  
  
**Sequelize** ti permette di dichiarare schema, vincoli, relazioni e logica (hook) in JS, poi sincronizzare verso il DB.

**bcrypt** garantisce che le password (e idealmente i refresh token) siano salvati **hashati**, non in chiaro.

**RefreshToken come model** è fondamentale se vuoi revoca granulare, rotazione, audit e multi-device: trattalo come **risorsa del dominio**, non come semplice stringa effimera.

**Cos’è Sequelize (qui, in pratica)**

* È l’**ORM** che mappa tabelle ↔️ oggetti JS.
* Con sequelize.define('Nome', { campi }, { opzioni }) descrivi **schema**, **vincoli**, **validazioni**, **hook**, **relazioni**.
* Poi sequelize.sync() crea/aggiorna le tabelle in base a queste definizioni.
* Vantaggi: niente SQL “a mano” per CRUD, relazioni esplicite, validazioni, hook (come l’hash della password) centralizzati.

**bcrypt**: serve a **hashare** la password prima di salvarla.

* genSalt(10) genera il sale; hash(plain, salt) produce l’hash.
* In beforeCreate e beforeUpdate garantisci che **mai** finisca in DB una password in chiaro.

È la **regola di integrità referenziale** sul **vincolo di chiave esterna**:  
onDelete: 'CASCADE' dice al database che **quando la riga “padre” viene eliminata**, tutte le righe “figlie” che la referenziano devono essere **eliminate automaticamente**.

Nel tuo caso:

* tabella **users** = padre
* tabella **refresh\_tokens** = figlia (ha userId come foreign key)
* se cancelli un utente, **tutti i suoi refresh token** vengono rimossi dal DB in automatico.

**Perché serve**

* Evita **record orfani** (token che puntano a un utente inesistente).
* Mantiene la **coerenza** dei dati senza dover scrivere delete manuali in cascata nel codice.

**Alternative comuni**

* SET NULL: se il padre viene cancellato, il FK nel figlio diventa NULL.
* RESTRICT / NO ACTION: **impediscono** la cancellazione se esistono figli.

Gli **hook** (o *lifecycle hooks*) in Sequelize sono **funzioni che si agganciano a momenti specifici del ciclo di vita** di un model/record: prima o dopo creare, aggiornare, validare, distruggere, ecc.   
Ti permettono di **eseguire logica automatica** senza doverla riscrivere in ogni controller.  
  


**Significa:**

* **beforeCreate**: prima che un nuovo utente venga inserito nel DB, l’hook genera un *salt* e **hasha la password**. Così non finirà mai in chiaro nel DB.
* **beforeUpdate**: prima di un update, se il campo password è cambiato (user.changed('password')), la **ri-hasha**.

**Perché sono utili**

* **Centralizzazione**: eviti di dimenticarti l’hash in un controller; la regola vive nel model.
* **Coerenza**: stessa logica ovunque (create/update/bulk).
* **Sicurezza**: impediscono salvataggi accidentali in chiaro.

**Tipi di hook comuni**

* **Validazione**: beforeValidate, afterValidate
* **Creazione**: beforeCreate, afterCreate
* **Aggiornamento**: beforeUpdate, afterUpdate
* **Salvataggio generico**: beforeSave, afterSave (coprono sia create che update)
* **Cancellazione**: beforeDestroy, afterDestroy
* **Bulk**: beforeBulkCreate, beforeBulkUpdate, …
* **Globali** (su sequelize): si applicano a tutti i modelli.

**Controllers**  
**Auth controllers**

**loginUser**

* **Cosa fa:** prende username e password, verifica l’utente, confronta la password con **bcrypt**, genera **accessToken** e **refreshToken**, salva il refresh nel DB.
* **Perché funziona:**
  + bcrypt.compare() verifica l’hash salvato.
  + generateAccessToken/RefreshToken creano i JWT.
  + RefreshToken.create(...) permette logout/revoca in futuro.
* **Attenzione:** salva i refresh **hashati** (come le password) per sicurezza; aggiungi rate-limit sul login.

**registerUser**

* **Cosa fa:** controlla che username/email non esistano, crea l’utente.
* **Perché funziona:**
  + L’hash password avviene negli **hook** del model User (prima del salvataggio).
  + Op.or cerca clash su username **o** email.
* **Attenzione:** normalizza email (toLowerCase), valida meglio la password.

**logoutUser**

* **Cosa fa:** prende un refreshToken e lo **revoca** (set revoked: true).
* **Perché funziona:**
  + Al prossimo uso, il refresh “non valido” viene rifiutato.
* **Attenzione:** se ruoti i refresh, qui puoi revocare **tutta la famiglia**; se salvi token hashati, fai compare per trovarlo.

**refreshUserToken**

* **Cosa fa:** verifica che il refresh esista e non sia revocato, verifica firma/scadenza JWT, crea un **nuovo access token**.
* **Perché funziona:**
  + jwt.verify garantisce che il refresh non sia scaduto/manomesso.
  + Recupera l’utente e rigenera l’access.
* **Attenzione:**
  + Implementa **rotazione** del refresh: quando lo usi, emetti un **nuovo refresh** e invalida il vecchio.
  + Verifica anche iss/aud nel verify.
  + Se token in DB è **hash**, confronta con bcrypt.compare.

**getUserProfile**

* **Cosa fa:** ritorna i dati dell’utente autenticato (preso da req.user impostato da un middleware che verifica l’access token).
* **Attenzione:** non restituire mai il campo password.

**TMDB controller**

**searchMovies**

* **Cosa fa:** chiama la **TMDB API** per cercare film (/search/movie), poi per ogni film carica i **dettagli con credits** per trovare **regista**, **generi**, filtra per genre/director/year, e mappa i dati puliti per il frontend.
* **Perché funziona:**
  + fetch fa due chiamate: ricerca → dettagli con append\_to\_response=credits.
* **Attenzione:**
  + Aggiungi **paginazione** (TMDB ha page).
  + Gestisci errori di rete/status code TMDB.
  + Metti **cache** o limiter per non sforare i rate limit.

**Movie controllers (DB locale)**

**getMovies**

* **Cosa fa:** filtra su title/genre/director/year con LIKE.
* **Perché funziona:**
  + Costruisce dinamicamente where usando Op.like.
* **Attenzione:**
  + Case-insensitive: su MySQL spesso LIKE è già case-insensitive; su altri DB usa LOWER(col) LIKE LOWER(:q).
  + Aggiungi **paginazione** (limit/offset) e **ordinamento**.

**getMovieById**

* **Cosa fa:** trova per PK e ritorna 404 se non esiste.

**createMovie**

* **Cosa fa:** controlla tmdb\_id e title, evita duplicati su tmdb\_id, salva il film.
* **Attenzione:**
  + Valida tipi (es. runtime numero), sanitizza input, e considera un **indice UNIQUE** su tmdb\_id.

**updateMovie**

* **Cosa fa:** carica il film, fa update con il body, ritorna il record aggiornato.
* **Attenzione:**
  + Convalida i campi aggiornabili (whitelist) per evitare modifiche indesiderate.

**deleteMovie**

* **Cosa fa:** elimina il film se esiste.

**Librerie usate (brevissimo)**

* **bcrypt**: confronto password (compare) con hash salvato (gli hash vengono creati negli **hook** del model User).
* **jsonwebtoken (jwt)**: crea e verifica **access/refresh token**.
* **sequelize + Op**: costruzione query in JS; Op.like, findOne, findAll, findByPk, create, update, destroy.
* **node-fetch**: chiama API TMDB.

**Passport.js** è una libreria che si occupa del login