TV Series App – Relazione

Introduzione

"TV Series App" è un'applicazione mobile sviluppata in **Flutter** che consente la gestione completa e personalizzata delle proprie serie TV preferite. La scelta di Flutter permette di sviluppare un'**app cross-platform**, funzionante sia su Android sia su iOS, sfruttando un'unica codebase. La persistenza locale dei dati è garantita tramite **SQLite**, implementato con il **plugin `sqflite`**, che assicura la conservazione offline e un accesso rapido ai dati.

L'app è strutturata con un'architettura modulare e separazione netta tra UI, logica di business e gestione dati, favorendo manutenibilità e scalabilità. La cartella `assets/immagini` contiene risorse grafiche locali, ottimizzate per un caricamento rapido senza dipendere da connessioni di rete.

main.dart – Avvio dell'applicazione

Il file **main.dart** costituisce il punto d'ingresso dell'intera applicazione Flutter. La funzione **main()** invoca **runApp()**, che monta il widget radice TVSeriesApp e avvia il rendering dell'interfaccia.

Struttura e scelte progettuali

```
void main() {
  runApp(const TVSeriesApp());
}
```

- La chiamata a runApp() ha il compito di inizializzare l'interfaccia grafica dell'app, montando il widget TVSeriesApp, che è una sottoclasse di StatelessWidget. La scelta di un widget senza stato (StatelessWidget) è motivata dal fatto che la configurazione iniziale dell'app è statica e non prevede aggiornamenti dinamici in questa fase.

```
class TVSeriesApp extends StatelessWidget {
  const TVSeriesApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      title: 'TV Series App',
      ...
    );
  }
}
```

Configurazione del `MaterialApp`

Il widget `MaterialApp` fornisce l'impalcatura visiva di base dell'app e definisce le impostazioni globali di stile, tema, e comportamento.

- **`debugShowCheckedModeBanner: false`**: rimuove il banner di debug in alto a destra, per offrire un aspetto più professionale e rifinito.
- **`title: 'TV Series App'`**: definisce il titolo dell'app, utilizzato ad esempio per le impostazioni del task manager nei dispositivi Android.
- **`home: const HomeScreen()`**: imposta la `HomeScreen` come schermata iniziale dell'app, da cui parte la navigazione.

Personalizzazione del tema

```
theme: ThemeData(
  scaffoldBackgroundColor: Colors.black,
  primarySwatch: Colors.red,
  fontFamily: 'Roboto',
  textTheme: const TextTheme(
    bodyLarge: TextStyle(color: Colors.white),
    bodyMedium: TextStyle(color: Colors.white),
    titleLarge: TextStyle(
      color: Colors.white,
      fontSize: 24,
      fontWeight: FontWeight.bold,
    ),
  ),
),
```

Il tema è stato definito tramite `ThemeData` e include:

- **ScaffoldBackgroundColor**: impostato su `Colors.black`, per adottare un tema scuro coerente con tutte le schermate dell'app.
- **PrimarySwatch**: impostato su `Colors.red`, coerente con gli elementi interattivi dell'interfaccia (come il `FloatingActionButton` e le icone principali).
- **fontFamily: 'Roboto'**: imposta il font principale dell'app, moderno e leggibile, in linea con le linee guida Material Design.

- **textTheme**: definisce lo stile del testo principale, usando un colore bianco per garantire visibilità sullo sfondo scuro. La dimensione e il peso del titolo (`titleLarge`) sono regolati per dare maggiore enfasi visiva alle intestazioni.

home_screen-Schermata principale

La `HomeScreen` è il punto di ingresso dell'applicazione **TV Series App**. È una schermata dinamica e multifunzionale che consente all'utente di accedere, esplorare e filtrare tutte le serie TV salvate nel database locale. Integra anche funzioni per la navigazione tra diverse sezioni (home, catalogo, preferiti, ricerca) grazie a una `BottomNavigationBar`.

Dettagli tecnici e funzioni implementate:

Caricamento dati

- I dati delle serie vengono recuperati tramite il metodo asincrono **`DatabaseHelper().getAllSeries()`** in **`initState()`**.
- Le serie sono suddivise in liste separate in base allo **stato**, **preferenze** e per la sezione di **evidenza**.
- È utilizzato **`setState`** per aggiornare lo stato della UI una volta ricevuti i dati dal database.

Sezioni dinamiche della home

Ogni serie viene assegnata dinamicamente a una specifica sezione della schermata principale in base al proprio stato:

- Se lo stato è **"in corso"**, la serie verrà visualizzata nella sezione **Continua a guardare**.
- Se lo stato è **"non iniziata"**, verrà mostrata tra i **Consigliati per te**.
- Se lo stato è **"completata"**, sarà inserita nella sezione **Top da rivedere**.

Questa logica garantisce una categorizzazione automatica coerente e personalizzata, migliorando l'esperienza utente e la navigabilità dei contenuti.

1. **Serie in evidenza**
   - Visualizzata in alto con un grande banner (immagine copertina + titolo).
   - Selezione casuale del catalogo completo.
   - Tappabile: apre `DetailScreen`.
2. **Continua a guardare**
   - Visualizzate in `ListView` orizzontale con card personalizzate.
3. **Consigliati per te**
   - Layout orizzontale, leggermente più grande.
4. **Top da rivedere**
   - Mostrate in formato card orizzontale.

Catalogo

- Accessibile tramite il secondo tab della barra di navigazione.
- Mostra due sezioni separate:
  - **Catalogo per Genere**
  - **Catalogo per Piattaforma**
- Ogni categoria è raggruppata e rappresentata con una **`GridView.builder`**, ottimizzata per il caricamento efficiente delle card.

Preferiti

- Accessibile dal terzo tab della **`BottomNavigationBar`**.
- Mostra le serie marcate come preferite **(`isFavorite == 1`)**.
- Layout verticale con anteprima immagine, titolo e piattaforma.
- Navigazione a `DetailScreen` al tap.

Ricerca avanzata

- Accessibile dal quarto tab.
- Comprende:
  - **`TextField`** per ricerca testuale (titolo, genere, piattaforma).
  - **`DropdownButton`** per filtri su **stato** e **numero stagioni**.
  - I filtri sono combinabili.

- I risultati sono mostrati con **`ListView.builder`**.
- Se nessun risultato viene trovato, compare un messaggio testuale.

Navigazione

- **Navigazione a `DetailScreen`**: ogni card/list tile è tappabile e apre una schermata dettagliata tramite `Navigator.push`.
- Alla chiusura di `DetailScreen`, la schermata viene aggiornata automaticamente con `_caricaSerie()`.

FloatingActionButton personalizzato

- Invece del tipico **`FloatingActionButton`**, il pulsante per aggiungere nuove serie è stato integrato nella **`AppBar`** tramite l'icona `Icons.add`.
- Premendolo, si apre **`AggiungiSerieScreen`**, e al ritorno i dati sono aggiornati tramite **`then((_) => _caricaSerie())`**.

Gestione dello stato e UI responsive

- Tutte le schermate sono costruite dinamicamente in base al valore di **`_paginaCorrente`**.
- I contenuti si adattano al numero di serie salvate, evitando rendering inutili (ritorno di **`SizedBox.shrink()`** in caso di liste vuote).
- Uso di **`SingleChildScrollView`** e **`Expanded`** per garantire un layout adattivo, anche in presenza di contenuti lunghi.

Aspetto grafico

- Background globale nero (`Colors.black`) con testi bianchi o grigi per leggibilità.
- Titoli in rosso coerenti con il branding dell'app.
- Le immagini sono caricate da asset locali tramite `Image.asset`, con path ottenuto da `serie.image`.

Detail_Screen–Schermata di Dettaglio Serie

La schermata `DetailScreen` consente la **visualizzazione approfondita** di una singola serie TV selezionata, mostrando tutte le informazioni principali e fornendo funzionalità per **modifica**, **eliminazione**, e **gestione dello stato utente** (episodi visti e preferiti).

Caratteristiche Tecniche:
Ricezione Parametri

Il costruttore riceve un oggetto `Serie` tramite la navigazione. Questo approccio consente di evitare ulteriori query al database e garantisce **prestazioni ottimizzate** e un'esperienza utente più fluida.

```
final Serie serie;
const DetailScreen({super.key, required this.serie});
```

Layout e Interfaccia Utente

Il layout si basa su una **colonna verticale** (`ListView`) che include:

- **Immagine della serie** tramite `Image.asset`.
- **Titolo**, **trama** (`plot`), **genere**, **piattaforma**, **stato**, **stagioni**, **numero episodi**.
- **Messaggio informativo** sulla disponibilità di una nuova stagione (`dataProssimoEpisodio`).
- **Slider** per modificare il numero di episodi visti, sincronizzato con il database.
- Pulsanti per aggiungere un episodio, gestire lo stato di preferito e eliminare la serie.

Esempio:

```
Slider(
  value: _episodiVisti,
  max: serie.episodes.toDouble(),
  onChanged: (value) {
    setState(() {
      _episodiVisti = value;
    });
    _salvaNelDatabase(_episodiVisti.toInt());
  },
);
```

Azioni Utente:
Modifica Serie

Tramite un **`IconButton`** con `Icons.edit`, l'utente può accedere alla schermata `ModificaSerieScreen`. L'oggetto `Serie` viene passato come parametro.

```
IconButton(
  icon: const Icon(Icons.edit),
  onPressed: _modificaSerie,
);
```

Eliminazione Serie

Tramite un **`IconButton`** con `Icons.delete` (all'interno di un `ElevatedButton`), viene mostrato un **`AlertDialog`** di conferma. In caso di conferma, la serie viene eliminata dal database e si ritorna alla schermata `HomeScreen`.

```
final db = await DatabaseHelper().database;
await db.delete('series', where: 'title = ?', whereArgs:
[widget.serie.title]);
```

Gestione Stato UI

- Il valore degli episodi visti è sincronizzato con il database attraverso la funzione `_salvaNelDatabase`.
- Lo stato di "preferito" è gestito con la funzione `_togglePreferito`, che aggiorna il valore nel database SQLite.
- Eventuali modifiche/eliminazioni causano il ritorno alla `HomeScreen`, consentendo l'**aggiornamento automatico** della lista, grazie all'uso di `Navigator.pop(context)` o `Navigator.pop(context, true)`.

Altre Funzionalità

- **Preferiti**: pulsante toggle per aggiungere o rimuovere dai preferiti, con icona dinamica (`Icons.favorite` / `Icons.favorite_border`).
- **Aggiunta rapida episodio visto**: pulsante che incrementa `_episodiVisti` di 1 e aggiorna il database.

aggiungi_serie_screen–Inserimento Nuova Serie

Schermata dedicata all'aggiunta di una nuova serie TV all'interno dell'applicazione.

Caratteristiche tecniche:
Form e validazione

- Tutti i campi sono implementati con `TextFormField` o `DropdownButtonFormField`, ciascuno dotato di validatore per garantire:
  - Compilazione obbligatoria.
  - Coerenza logica dei dati (es. episodi visti ≤ episodi totali).
- Il form è gestito tramite `GlobalKey<FormState>` per abilitare la validazione completa prima del salvataggio.

Selezione opzioni tramite dropdown

- `Genere`, `Piattaforma` e `Stato` sono selezionabili tramite menu a tendina con valori predefiniti.
- Questo approccio previene errori di battitura e mantiene la coerenza dei dati nel database.

Salvataggio e inserimento

- Alla pressione del pulsante **"Salva"**:
  - Se il form è valido, viene creato un oggetto `Serie` con i dati raccolti.
  - Viene chiamato `DatabaseHelper.insertSerie(serie)` per l'inserimento nel database.
  - L'utente viene riportato alla `HomeScreen`, che aggiorna la lista.

Validazione avanzata

- Se lo stato è `"completata"`, gli episodi visti sono impostati automaticamente uguali a quelli totali.
- Se lo stato è `"in corso"`, viene mostrato il campo aggiuntivo per gli episodi visti e si verifica che non superino il totale.
- In caso di errore, viene visualizzato un `SnackBar` con un messaggio d'allerta.

modifica_serie_screen–Modifica Serie Esistente

Questa schermata consente all'utente di aggiornare i dati di una serie TV già salvata nel database. L'interfaccia è composta da un form con campi precompilati, validazione integrata e logica condizionale basata sullo stato della serie.

Dettagli implementativi

Precompilazione automatica dei campi
Tutti i campi del form sono associati a `TextEditingController` inizializzati con i

valori correnti della serie passata come parametro (`Serie`). I `DropdownButtonFormField` vengono inizializzati con il valore corrente dell'attributo corrispondente (genere, piattaforma, stato), così da facilitare l'editing evitando un reinserimento completo dei dati.

Gestione dinamica dei campi

Il campo "Episodi visti" è visibile solo se lo stato della serie è impostato su "in corso". Se l'utente seleziona "completata", il campo scompare e il numero di episodi visti viene automaticamente impostato uguale al totale degli episodi. Questa logica è implementata nel listener associato al `Dropdown` dello stato.

Validazione

La validazione è gestita a due livelli:

- **A livello di interfaccia** tramite `Form` e relativi validator, che impediscono l'invio se i campi sono vuoti o non coerenti.
- **A livello di logica applicativa**, per esempio impedendo che il numero di episodi visti superi quello totale. In tal caso viene mostrato uno `SnackBar` rosso con messaggio d'errore e il salvataggio viene interrotto.

Aggiornamento del database

Alla pressione del pulsante "Salva modifiche", se tutti i controlli sono superati, viene creato un nuovo oggetto `Serie` con gli aggiornamenti. Viene poi chiamato:

```
dart
Copia codice
await db.update('series', serieAggiornata.toMap(), where: 'id = ?',
whereArgs: [widget.serie.id]);
```

L'operazione SQL `UPDATE` modifica la riga corrispondente nel database. Dopo il completamento, la schermata viene chiusa con `Navigator.pop`, ritornando a `DetailScreen` o `HomeScreen` con i dati aggiornati.

Interfaccia utente

L'UI mantiene uno stile coerente con il resto dell'app: sfondo nero, testi bianchi, accenti rossi per evidenziare i bordi attivi o i pulsanti principali. La struttura della schermata è gestita tramite un `Form` racchiuso in uno `Scaffold` con `AppBar`, ed è completamente scrollabile grazie all'uso di `ListView`.

Database-Gestione con SQLite

Il cuore della persistenza dei dati nella *TV Series App* è rappresentato da un database locale SQLite, progettato per conservare tutte le informazioni relative alle serie TV in modo strutturato, efficiente e persistente.

Struttura tecnica

La gestione del database è incapsulata all'interno della classe `DatabaseHelper`, implementata seguendo il **pattern Singleton**, che garantisce un'unica istanza condivisa del database per l'intera applicazione, prevenendo conflitti e duplicazioni di accesso.

Creazione della tabella

Nel metodo `onCreate`, viene generata la tabella `series` con i seguenti campi:

```
CREATE TABLE series (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  title TEXT NOT NULL,
  plot TEXT,
  genre TEXT,
  platform TEXT,
  image TEXT,
  status TEXT,
  seasons INTEGER,
  episodes INTEGER,
  episodesWatched INTEGER DEFAULT 0,
  isFavorite INTEGER DEFAULT 0,
  dataProssimoEpisodio TEXT
)
```

Questa struttura supporta tutte le funzionalità dell'app, comprese le informazioni testuali e numeriche, lo stato di avanzamento, i preferiti e la data del prossimo episodio.

Dati pre-inseriti

Durante la creazione del database vengono automaticamente inserite **12 serie TV** di esempio, ognuna completa di titolo, trama, genere, piattaforma, immagine, stato, stagioni, episodi, episodi visti, preferiti e data del prossimo episodio (ove applicabile). Questo consente di offrire contenuti dimostrativi già alla prima apertura dell'app.

Operazioni CRUD

La classe `DatabaseHelper` espone diversi metodi per la gestione dei dati:

- **Inserimento (`insertSerie(Serie s)`):** Converte l'oggetto `Serie` in una mappa (`toMap()`) e la inserisce nel database con `db.insert('series', map)`.
- **Recupero (`getAllSeries()`):** Estrae tutte le righe dalla tabella, convertendo ogni mappa in un oggetto `Serie` con `fromMap`.

- **Aggiornamento episodi visti (`aggiornaEpisodiVisti`)** e **preferiti (`aggiornaPreferiti`)**:
  Utilizzano `db.update` per modificare i campi `episodesWatched` o `isFavorite` in base al titolo della serie.

- **Recupero preferiti (`getPreferiti()`):** Ritorna una lista filtrata di tutte le serie contrassegnate come preferite (`isFavorite = 1`).

Logica di conversione dati

Il modello `Serie` implementa due metodi fondamentali per l'integrazione con SQLite:

- `toMap()`: converte un'istanza in una mappa (`Map<String, dynamic>`) compatibile con le operazioni del database.
- `fromMap(Map<String, dynamic>)`: crea un oggetto `Serie` a partire dai dati memorizzati nel database.

Questa astrazione permette una gestione flessibile dei dati, semplificando l'interazione tra l'applicazione e il database sottostante.

Vantaggi dell’approccio adottato

- **Modularità:** la logica del database è isolata all’interno di una classe dedicata.
- **Espandibilità:** l’aggiunta di nuovi campi o tabelle è agevolata da una struttura già scalabile.
- **Efficienza:** le query dinamiche, la gestione centralizzata e l’uso del singleton ottimizzano prestazioni e memoria.

Conclusione

"TV Series App" integra un sistema solido per la gestione offline di serie TV, utilizzando Flutter per la **UI reattiva** e **SQLite** per una gestione dati efficiente e persistente. L’architettura modulare e il rispetto delle best practice Flutter rendono l’app estendibile, manutenibile e pronta a future integrazioni (es. sincronizzazione cloud, notifiche).

La combinazione di interfacce user-friendly con filtri avanzati, immagini gestite localmente, validazioni robuste e una gestione dinamica dei dati consente un’esperienza d’uso fluida e professionale, pienamente aderente ai requisiti progettuali.