2020

MobDev Projects

CHRONO TRACKER (ANDROID) WEATHER APP (IOS)

LUCA FAGGION

ANDROID – CHRONO TRACKER APP

1.STRUTTURA GENERALE

L'applicazione è stata strutturata seguendo il pattern di programmazione **MVC (Model -View - Controller)** utilizzato come modello base dal Framework UI di Android, sono stati cosi divisi i vari moduli dell'app in base alla loro funzione. Per la UI è stata utilizzata la libreria **Android Material** (https://material.io/develop/android/) per ottenere una UI moderna, responsiva e conforme alle specifiche del Material Design.

MODELS:

(fanno parte dei modelli tutte le classi utilizzate per lo storage dei dati e il database **SQLite**)

- **ActivityGeneral:** classe base per una attività contiene ID e Nome dell'attività
  - **ActivitySport**
  - **ActivitySportSpecialization:** variante di uno sport (attività)
- **ActivitySession:** descrive una sessione di allenamento di un utente comprende riferimenti al profilo dell'utente (atleta) all'attivata' svolta, tempo di inizio e tempo di fine e i giri (Laps) effettuati durante la sessione
- **Athlete:** descrive il profilo di un utente (atleta)
- **Lap:** descrive un periodo di tempo da uno specifico momento
- **Database SQLite**

HELPERS:

le classi helpers (o di servizio) sono utilizzate in tutta l'app nello specifico nei controllers per effetuare diverse operazioni, posso essere utilizzate per mettere in **comunicazione i Modelli con i Controllers** come nel caso della classe **Database**

- **Database class:** classe singleton che si occupa di gestire le query per la modifica/aggiornamento/recupero dei dati dal database analizzato più avanti nella relazione, tutte le query al database sono eseguite in modo asincrono utilizzando la classe **AsyncTask** ogni funzione prende in input i dati per realizzare la query e ritorna il risultato utilizzando il metodo di **callback** fornito alla funzione
- **FileIO:** classe utilizzata per scrivere e leggere file dal filesystem di Android.
- **Utils :** classe con funzioni generiche di utilità (ex: formatString, concatString, formatTime, etc...).

CONTROLLERS:

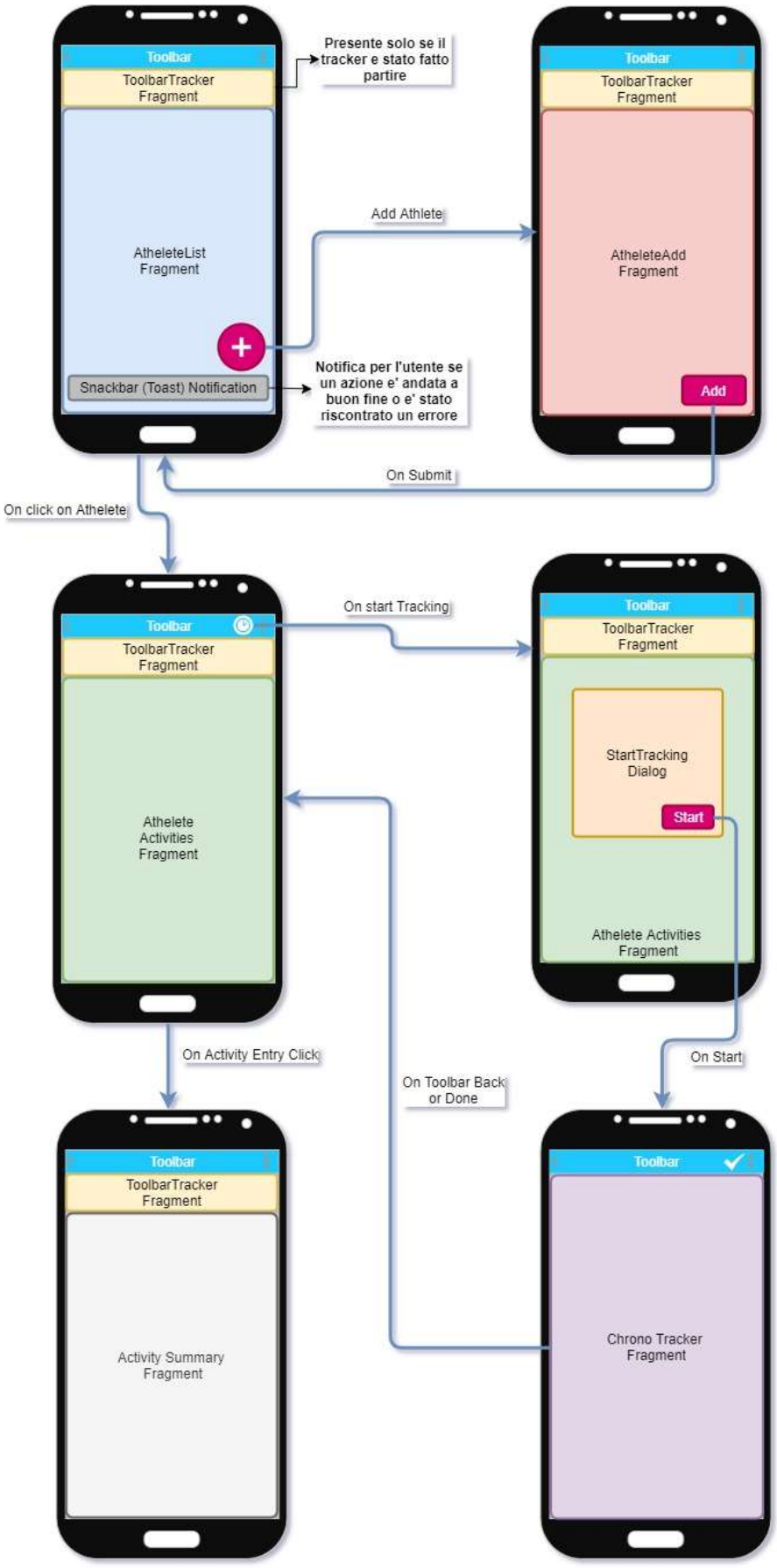
I controllers sono l'elemento centrale nel pattern MVC permettono di gestire l'input dell'utente e preparano e inviano alle view i dati che devono essere visualizzati, l'app e stata strutturata utilizzando **Single Activity multiple Fragments** per ottenere una UX fluida e responsiva al costo di una maggiorata resource allocation da parte dell'app, ciò significa che tutti i controllers sono dei **Fragments** con i propri lifecycles ed è presente una singola **Main Activity** che coordina l'intera app.

- **AthleteList:** fragment mostrato all'apertura dell'app mostra una lista dei profili di atleti creati dell'utente la lista è implementato utilizzando una **RecycleView** e tramite un **FloatingActionButton** permette di navigare alla creazione/aggiunta di profili
- **AthleteAdd:** fragment che permette la creazione di un nuovo profilo compilando un form, il form è validato quando l'utente preme il tasto (Add) aggiungi, se il form risulta invalido viene mostrato all'utente quale dei compi a dei dati non accettabili. La validazione del form consente, oltre che per

ragioni di sicurezza, di avere dati conformi alle specifiche richieste dell'applicazione e consentirne quindi un corretto funzionamento.

- **AthleteActivities:** fragment composto da una **CalendarView** e una RecycleView che mostra per ogni giorno selezionato la lista delle attività che sono state svolte.
- **ActivitySummary:** fragment che mostra il resoconto di una attività.
- **ChronoTracker:** fragment che consente di cronometrare una attività è composto da una RecycleView e un **Custom Layout** per il rendering del tracker, il fragment utilizza anche un **BoundService** per mantenere il tracking del tempo anche se l'applicazione viene posta in background.
- **ToolbarTracker:** questo fragment e mostrato solo se si esce dal **ChronoTracker Fragment** senza finire la sessione permette di gestire il cronometraggio senza trovarsi nel fragment rendendolo disponibile nell'itera app come una toolbar di facile accesso e quindi possibile utilizzare l'intera app, effettuare modifiche, aggiungere utenti, etc. avendo sempre la possibilità di ritornare e gestire l'attuale sessione.

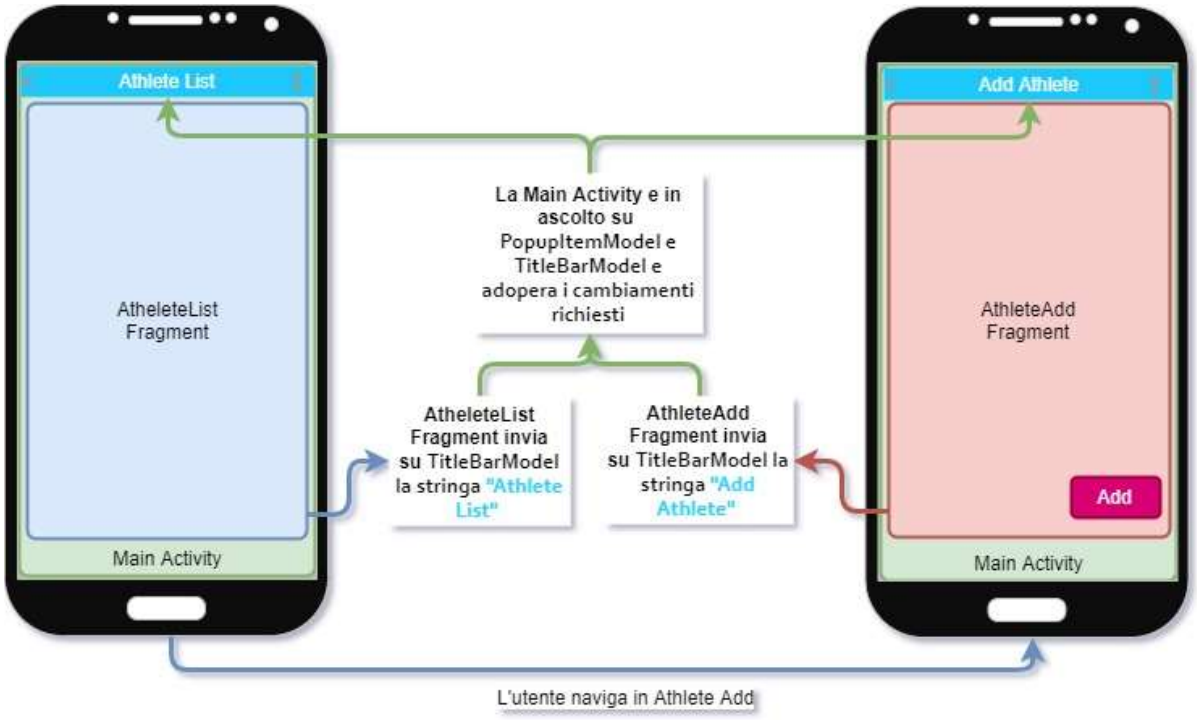
APP FLOW – SCHEMA DI NAVIGAZIONE DEI CONTROLLERS



COMUNICAZIONE TRA CONTROLLERS

Per la comunicazione avviene tramite **Observables (LiveData)** ogni fragment registra e invia i dati utilizzando differenti Observables per definire differenti flussi di dati derivanti dalle azioni dell'utente o anche dal recupero di dati dal Database

- **ActivitySessionModel:** viewmodel contenente observables per l'inizio e la fine delle sessioni di tracking.
- **AthleteModel:** viewmodel contenente observbles per dati relativi al profilo di un atleta (Aggiunta/ modifica, selezione)
- **PopupItemModel e TitleBarModel:** sono due view model utilizzate per semplificare il processo di modifica della struttura della toolbar. La toolbar è unica per l'intera app e non viene rimpiazzata quando si naviga da una schermata all'altra ma solo i fragment sono aggiunti e rimossi dallo stack durante la navigazione, queste due obsevables quindi sono utilizzate come punto di comunicazione da un fragment che ha necessità di modificare la toolbar e la Main Activity che adopera i cambiamenti richiesti.
  Esempio:

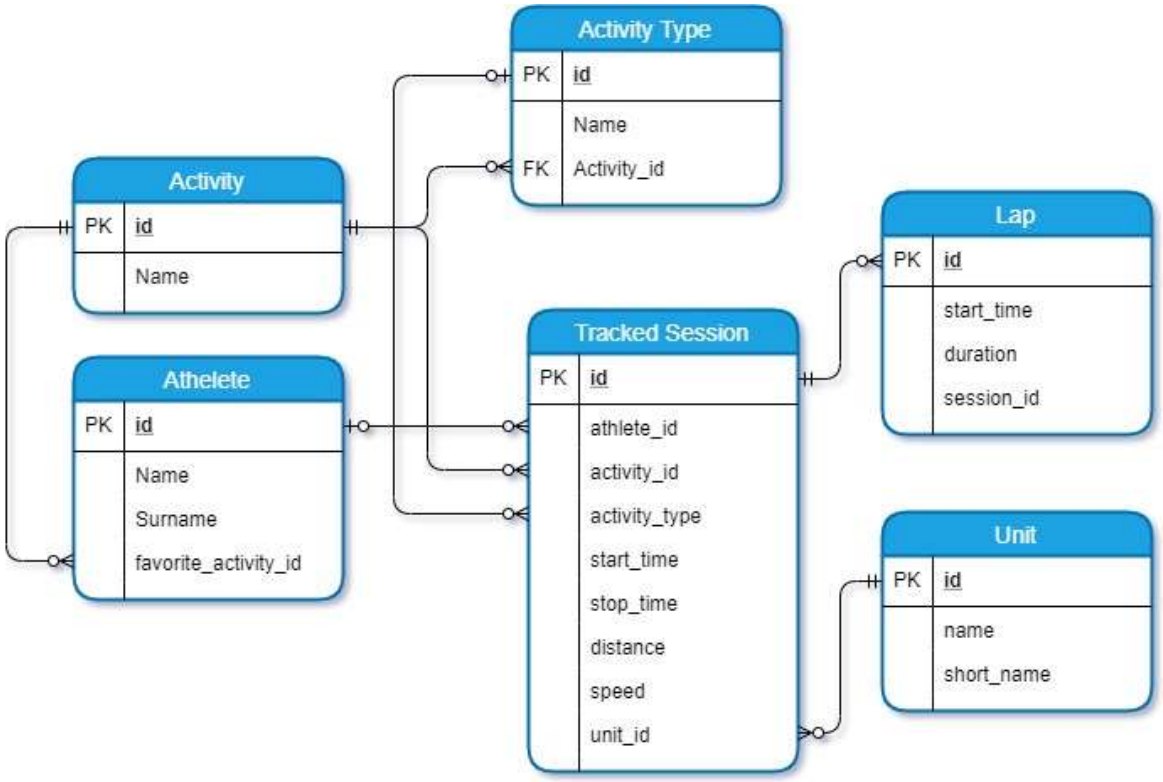


2.DATABASE

Per l'app è stato utilizzato il database relazionale **SQLite.** Sono state create tutte le classi riportate nel paragrafo **1.struttura generale.models** per una più facile manipolazione dei dati ricevuti dal database in questo paragrafo verrà tratta la struttura del database e l'implementazione della comunicazione con esso.

SCHEMA DATABASE

Lo schema del database è uno diagramma entità relazione che descrive le relazioni che intercorrono tra le varie **entità (tabelle)** del database. La classe **DatabaseHelper (che estende SQLiteOpenHelper)** si occupa della creazione del database e nelle future release dell'app di apportare i corretti aggiornamenti al database



IMPLEMENTAZIONE QUERY ASINCRONE

Per descrivere l'implementazione adottata per eseguire query in modo asincrono partiamo analizzando il risultato finale per destrutturarne e descriverne il funzionamento e le classi coinvolte.

```
public void getActivities(DatabaseResult result, DatabaseError error) {
    NoLeakAsyncTask<Void, Void, Cursor> task = new NoLeakAsyncTask<>(
            mContext,
            (Void... voids) -> {
                SQLiteDatabase db = dbHelper.getWritableDatabase();
                Cursor queryCursor = db.query(AppTables.ACTIVITY_TABLE.getName(),
                        new String[]{
                                AppTables.TABLE_ID_COL.getName(),
                                AppTables.ACTIVITY_TABLE_COL_0.getName()},
                        null,
                        null,
                        null,
                        null,
                        null);
                return queryCursor;
            },
            (Cursor cursor) -> {
                result.OnResult(cursor);
            },
            (Exception e) -> {
                error.OnError(e);
            }
    );
    task.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR);
}
```

Questa funzione della classe **Database Helper** è una funzione che esegue una query del tipo "SELECT columns FROM table" in modo asincrono da notare che la funzione non ha dati di ritorno (void), richiede come parametri di input due interfacce **DatabaseResult** e **DatabaseError** e il metodo fa l'uso della classe **`NoLeakAsyncTask`**. Cosi come la piattaforma Android fa largo utilizzo dei metodi di callback anche questa implementazione utilizza lo stesso pattern per garantire una esecuzione asincrona delle query al database.

QUERY CALLBACKS

Iniziamo quindi con analizzare i callback di input, tutti i metodi di callback di input sono eseguiti nel main thread ovvero quello della UI quindi possono essere utilizzati per eseguire **updates** alla **user interface.**

Sono presenti anche metodi eseguiti in background ma questi non sono mai esposti al di fuori della classe Database, ma sono utilizzati come metodi privati per eseguire le opportune operazioni in background all'interno della classe `NoLeakAsyncTask` ne vedremo l'utilizzo nel prossimo paragrafo.

Qui sotto sono riportate le interfacce che i metodi della classe helper Database richiedono come input, come si può notare sono annotate con `@FunctionalInterface` siccome sono interfacce con un singolo metodo astratto l'annotazione consente anche la possibilità di essere create con l'uso della lambda e il type checking durante la compilazione del codice.

```
//Interfaccia per callback in caso di errore
@FunctionalInterface
public interface DatabaseError {
    void OnError(SQLException exception);
}
//Interfaccia per callback dopo esecuzione di query di tipo INSERT/UPDATE
@FunctionalInterface
public interface DatabaseInsert {
    void OnInsert(long id);
}
//Interfaccia per callback dopo esecuzione di query di tipo SELECT
@FunctionalInterface
public interface DatabaseResult {
    void OnResult(Cursor cursor);
}
```

CLASSE NOLEAKASYNCTASK

La classe `NoLeakAsyncTask` e' una estensione della classe del framework android `AsynkTask`, consente di eseguire una task in background (in un thread differente dal principale) e al suo completamento o fallimento di chiamare metodi gli opportuni metodi di callback nel main thread ovvero quello della UI.

La classe è stata strutturata per semplificare le operazioni di query asincrone di un database ma essendo stata creata in modo da essere molto flessibile consente di gestire qualsiasi altra operazione che debba essere eseguita in background, in questa app il suo utilizzo e prettamente relativo all'esecuzione di query

Cosa distingue questa classe dalla sua classe base `AsynkTask`?

La differenza principale risiede nel fatto che questa classe detiene un `WeakReference<Activity>` ad una Activity, che è utilizzato nel momento in cui la task di background finisce l'esecuzione per stabile se e presente una activity oppure. se la task è sopravvissuta più a lungo della activity che la fatta partire. Nel secondo caso i metodi di callback **non vengono eseguiti** siccome non esiste una interfaccia grafica da aggiornare siccome l'activity e stata distrutta.

La classe mette a disposizione oltre a i metodi di callback eseguiti nel main thread anche callback che vengono eseguiti nel **background thread,** consentendo così una totale flessibilità per la sua configurazione.

```
1)public NoLeakAsyncTask(Activity context, BackgroundTask<I, R> task)

2)public NoLeakAsyncTask(Activity context, BackgroundTask<I, R> task, PostTask<R>
postTask)
```

```
3)public NoLeakAsyncTask(Activity context, BackgroundTask<I, R> task, PostTask<R>
postTask, ErrorTask errorTask)

4)public NoLeakAsyncTask(Activity context, BackgroundTask<I, R> task, PostTask<R>
postTask, ErrorTask errorTask, ThreadPostTask threadPostTask)

5)public NoLeakAsyncTask(Activity context, BackgroundTask<I, R> task, PostTask<R>
postTask, ErrorTask errorTask, ThreadErrorTask threadErrorTask, ThreadPostTask
threadPostTask)
```

Un utilizzo che mostra la flessibilità di questa classe e rappresentato dalla funzione **addSession** riportato qui sotto che fa uso delle database trans action per annullare un eventuale inserimento di una sessione.

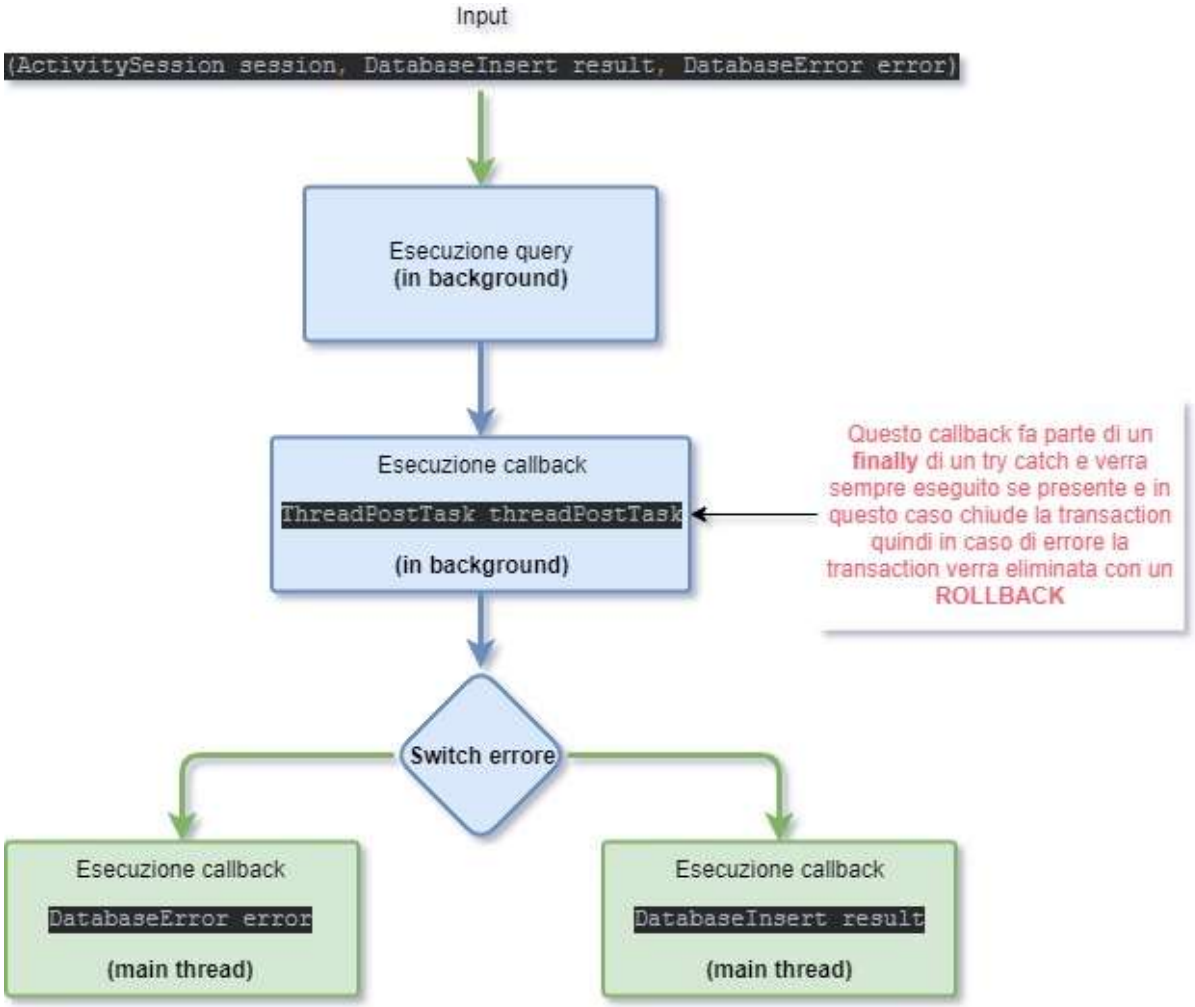
```
public void addSession(ActivitySession session, DatabaseInsert result,
DatabaseError error) {
    NoLeakAsyncTask<Void, Void, Long> task = new NoLeakAsyncTask<>(
            mContext,
            (Void... voids) -> {
                SQLiteDatabase db = dbHelper.getWritableDatabase();
                db.beginTransaction();
                ContentValues values = new ContentValues();
                values.put(AppTables.SESSION_TABLE_COL_0.getName(),
session.athlete);
                values.put(AppTables.SESSION_TABLE_COL_1.getName(),
session.activity);
                values.put(AppTables.SESSION_TABLE_COL_2.getName(),
session.activityType);
                values.put(AppTables.SESSION_TABLE_COL_3.getName(),
session.startTime);
                values.put(AppTables.SESSION_TABLE_COL_4.getName(),
session.stopTime);
                values.put(AppTables.SESSION_TABLE_COL_5.getName(),
session.distance);
                values.put(AppTables.SESSION_TABLE_COL_6.getName(), session.speed);
                long insertResult =
db.insertOrThrow(AppTables.SESSION_TABLE.getName(), null, values);

                for (int i = 0; i < session.laps.length; i++) {
                    ContentValues lap_values = new ContentValues();
                    lap_values.put(AppTables.LAP_TABLE_COL_0.getName(),
session.laps[i].fromStart);
                    lap_values.put(AppTables.LAP_TABLE_COL_1.getName(),
session.laps[i].duration);
                    lap_values.put(AppTables.LAP_TABLE_COL_2.getName(),
insertResult);
                    db.insertOrThrow(AppTables.LAP_TABLE.getName(), null,
lap_values);
                }

                db.setTransactionSuccessful();
                return insertResult;
            },
            (id) -> result.OnInsert(id),
            (e) -> error.OnError((SQLException) e),
            () -> {
                SQLiteDatabase db = dbHelper.getWritableDatabase();
                db.endTransaction();
            }
    );
    task.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR);
}
```

Questo utilizzo della classe `NoLeakAsyncTask` esegue il costruttore numero 4 riportato prima.

Quindi il flow di eventi può essere descritto nel seguente modo:

in background la classe apre il database e inizializza una **transaction** esegue il setup della **query** e prova ad eseguirla con la funzione `db.insertOrThrow` se tutto va per i meglio viene chiusa la transaction e poi eseguita la funzione di callback di input `DatabaseInsert result` con `result.OnInsert(id)` ovvero l'id dell'elemento appena inserito, altrimenti un errore viene generato e come si può vedere dal codice la funzione `setTransactionSuccessful` non viene mai chiamata quindi quando la transaction viene chiusa qualsiasi cambiamento nel database viene annullato siccome e eseguito un **ROLLBACK.**

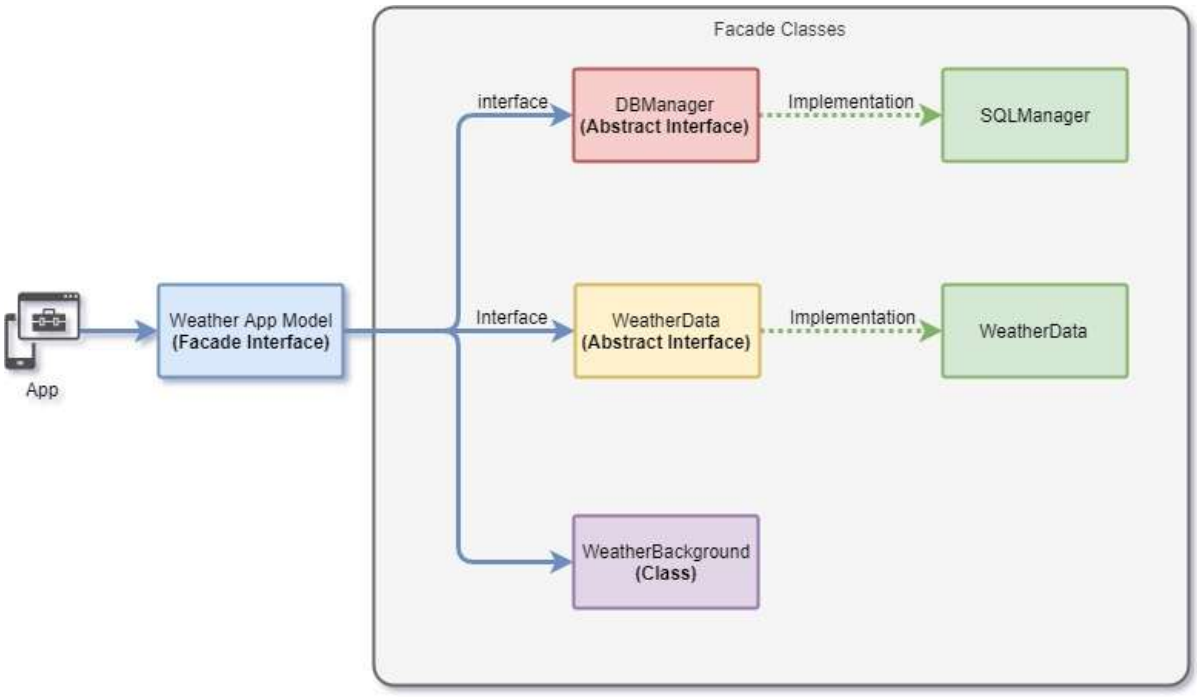
Il funzionamento schematizzato



IOS – WEATHER APP

1.STRUTTURA GENERALE

Come per la piattaforma Android, iOS è basata sul pattern di programmazione **MVC (Model View Controller),** l'app e stata quindi strutturata per adattarsi a questo pattern. Per i dati relativi al meteo sono state utilizzate le **API** di OpenWeatherMap (https://openweathermap.org/api) che provvedono nel fornire i dati sia per il meteo attuale che per previsioni meteo, con previsioni scandite ogni 3 ore per un periodo di 5 giorni. Tutte le query per il ricevimento dati sono fatte utilizzando il networking in modo **asincrono** cosi da garantire una user experience il più fluida possibile anche grazie all'uso di `Skeletal View` che consentono di all'utente di avere transizioni fluide durante il caricamento dei dati. L'app e stata anche strutturata in modo da risultare modulare utilizzando il pattern **Facade** per astrarre le varie implementazioni dei vari componenti chiave dell'app quali il database, la comunicazione con le API (**Networking**), e la gestione delle view in background, la classe WeatherAppModel (**Singleton class**) provvede a questo scopo ed è utilizzata nell'itera app. Per una efficiente gestione della memoria è stato utilizzato il sistema **ARC (Automatic Reference Counting)**



MODELS:

Le classi models sono legate sia allo storage dei dati sia alla rappresentazione in memoria dei dati ricevuti dalle API Meteo tramite networking.

- `CityWeather`**:** questa classe è la rappresentazione in memoria dei dati ricevuti tramite le API meteo si occupa di aggiornare i dati se richiesto e di trasmettere tramite, l'utilizzo del **protocollo** `WeatherModelDelegate,` i dati o eventuali errori durante l'aggiornamento dei dati meteo. La classe utilizza anche il database per recuperare le informazioni statiche, relative alla citta (come nome, posizione, stato, etc..), contenute nel database. **La classe si occupa quindi di aggiornare le informazioni meteo attuali e le previsioni, rappresentate relativamente dalle classi:**
    - **CurrentWeather**: contiene le informazioni meteo attuali

- ForecastWeather: contiene un array di classi **CurrentWeather** che contengono informazioni sul meteo ad intervalli di 3 ore ognuna

```
@interface CityWeather : NSObject

@property (nonatomic,strong)NSNumber * ID;
@property (nonatomic,strong)NSString* name;
@property (nonatomic,strong)NSString* country;
@property (nonatomic,strong)NSNumber* lon;
@property (nonatomic,strong)NSNumber* lat;
@property (nonatomic,strong)NSDate* last_updated;
@property Boolean hasData;
@property (nonatomic,strong)CurrentWeather* current;
@property (nonatomic,strong)ForecastWeather* forecast;

@property NSObject<WeatherModelDelegate>* delegate;

-(instancetype) initWithCityID:(NSNumber*)city_id;
-(instancetype) initWithCityID:(NSNumber*)city_id update:(Boolean)update;
-(void) performUpdate;

@end
```

- **WeatherHistoryEntry:** classe che rappresenta dati storici di uno specifico tempo e condizione meteo associati ad una città è utilizzata per rappresentare questi dati che provengono dal database

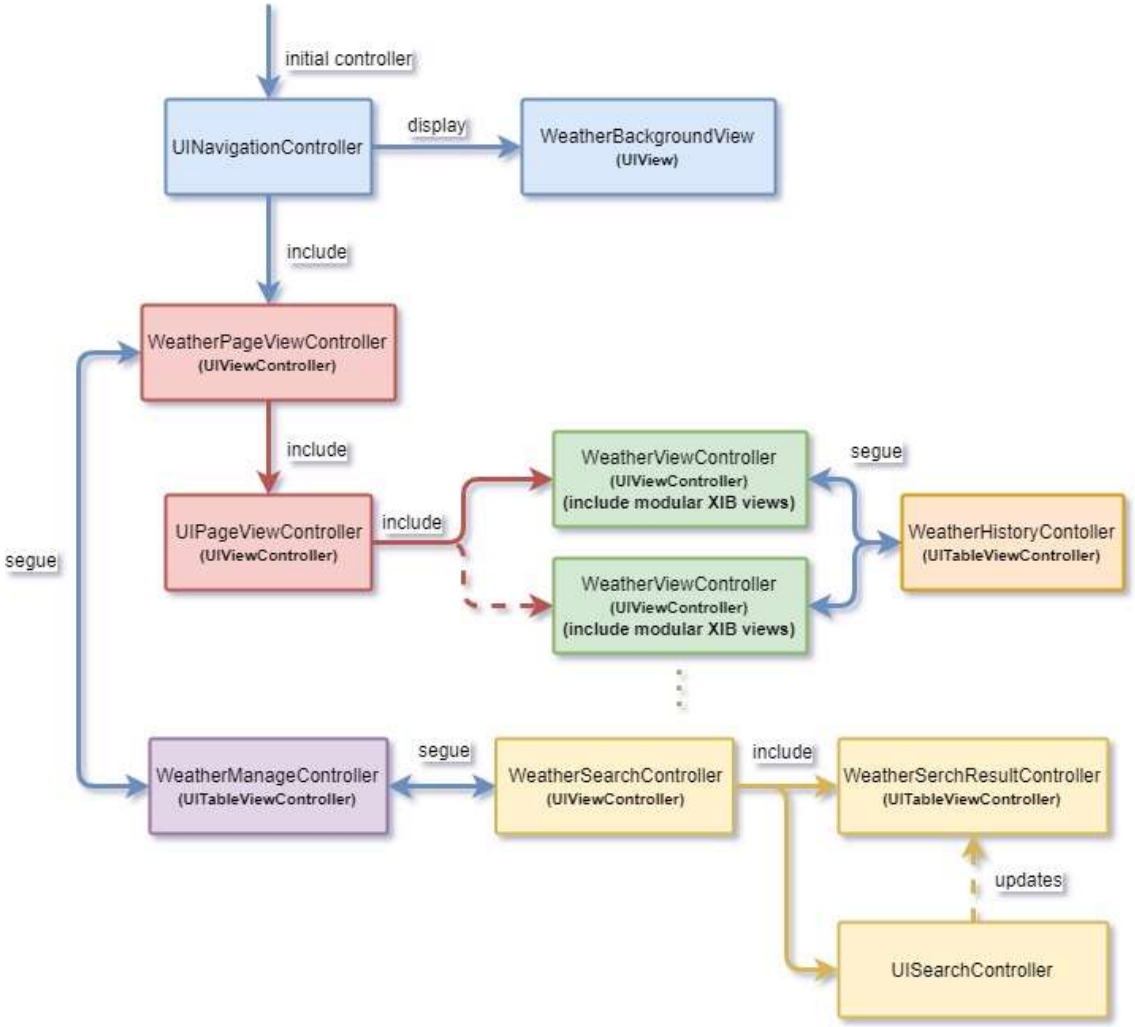
CONTROLLERS:

I controllers in iOS sono la parte centrale dell'applicazione che consente di gestire gli input dell'utente e di processare i dati provenienti dai models e provvedere a inviarli alle view per presentarli.In iOS ogni controller ha associato di base una view vuota che funge da contenitore per tutte le subview che il controller gestisce e o crea, si viene quindi a formare una struttura ad albero che rappresentano astrattamente le relazioni padre figlio di ogni view, e siccome un controller può essere padre di un altro controller (ovvero **Controllers di View Controllers**) le interfacce posso diventare molto complicate ma in questo modo e possibile un alta personalizzazione e flessibilità di esse.

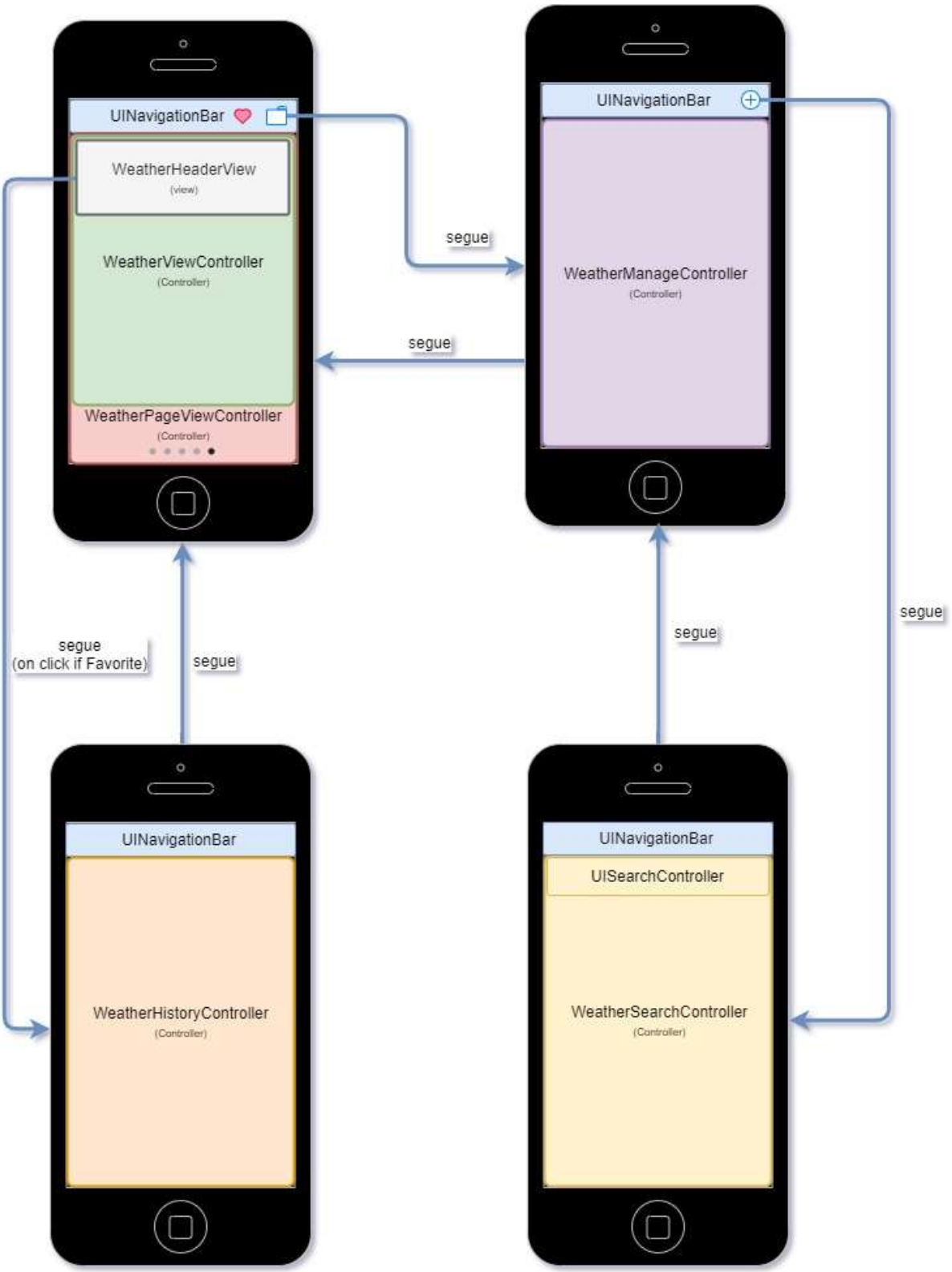
- **WeatherPageViewController:** (UIViewController) gestisce la principale schermata dell'app consente tramite un **UIPageViewController** di gestire con le gestures la navigazione tra i vari WeatherViewControllers che visualizzano ognuno i dati meteo relativi alle città scelte dall'utente.
- **WeatherViewController:** (UIViewController) gestisce molteplici view modulari (XIB) in base o meno alla presenza nei dati di certi eventi atmosferici e il componente principale dell'app. Notifica il WeatherBackgroundView di quale preset mostrare in base all'evento meteorologico attuale della citta visualizzata
- **WeatherManageContoller:** (UITableViewController) gestisce l'aggiunta e rimozione delle città, consente di salvare le città preferite su un file sul dispositivo
- **WeatherSearchController:** (UIViewController) gestisce la ricerca tramite due controller **UISearchController** per gestire l'input dell'utente e creare le query per il database e **WeatherSearchResultController** (UITableViewController) per mostrare i risultati del database, da

notare che le query eseguite sul database completamente asincrone per avere una UI responsiva e fluida.

VIEW HIERARCHY



APP FLOW


UINavigationBar
WeatherHeaderView
(view)
WeatherViewController
(Controller)
WeatherPageViewController
(Controller)
segue
segue
UINavigationBar
WeatherManageController
(Controller)
segue
segue
(on click if Favorite)
segue
segue
UINavigationBar
WeatherHistoryController
(Controller)
UINavigationBar
UISearchController
WeatherSearchController
(Controller)

COMUNICAZIONE TRA CONTROLLERS

La comunicazione tra controllers avviene tramite il passaggio di dati nella funzione

Utilizzata per recuperare un segue tramite il proprio ID e in base a quello e possibile identificare il prossimo controller che verrà visualizzato e quindi passagli i dati necessari. Nell' app è utilizzato questo metodo non solo per passare dati ma per registrare tramite l'utilizzo di protocolli anche delegati utilizzati per ottenere una connessione tra due view controllers come nel caso del passaggio dal **WeatherPageViewController** al **WeatherManageContoller**:

```
#pragma mark - Navigation
-(void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender{
   if([[segue identifier] isEqualToString:@"goToManage"]){
      WeatherManageController* manageController = (WeatherManageController*)[segue
destinationViewController];
      manageController.delegate = self;
   }
}
```

In questo caso il **WeatherPageViewController** viene registrato come delegato nel WeatherManageContoller per essere notificato di quando una citta viene aggiunta o rimossa.

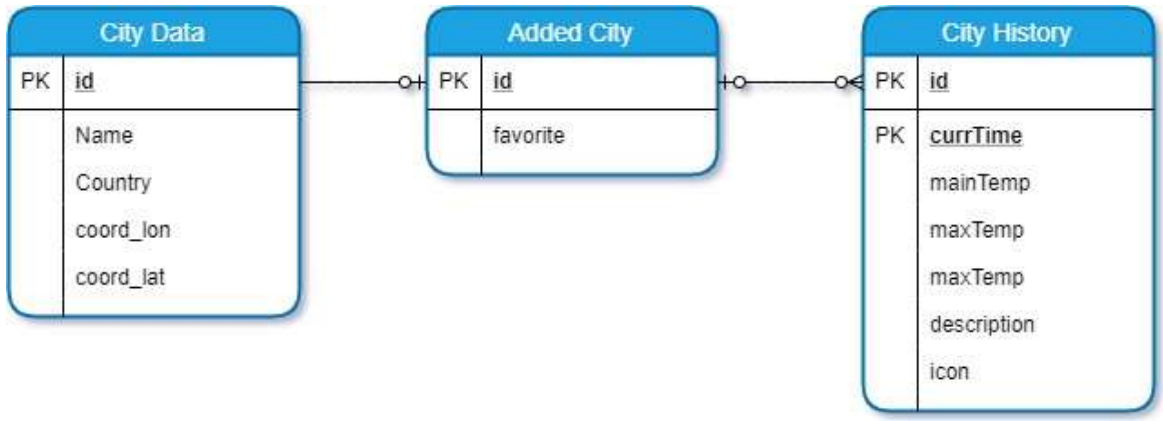
Questo è il protocollo al quale la classe **WeatherPageViewController** si conforma:

```
@protocol WeatherManageDelegate <NSObject>

-(void) onDeleteCityAtIndex:(NSInteger)index;
-(void) onAddCity:(CityWeather*)data;
-(NSArray*) getCities;
-(NSArray<CityWeather*>*) getCitiesWeather;
@end
```

2.DATABASE

Il database utilizzato e SQLite e la sua implementazione è descritta nella classe **SQLManager.** Il database per questa app e molto semplice e il suo scopo principale e tenere cache della lista della citta per poter eseguire query di ricerca in modo più efficace se si fosse utilizzato il **Networking tramite le API** per gestire le richieste di ricerca oltre che a una più lenta responsività dell'app si sarebbe certamente incorsi in un superamento delle richieste al minuto poste dalle API (60 cpm) siccome sarebbe stato l'utente durante la digitazione del nome della citta a creare di volta in volta richieste.

SCHEMA DATABASE



3.NETWORKING

La tipologia di app la rende pesantemente basata sull'utilizzo di API esterne con richiesta tramite **HTTP** (**HyperText Trasfer Protocol**) per il recupero dei dati metereologici. Nell'app la classe responsabile di gestire queste richieste e restituire i dati è la classe **WeatherData** e protocollo **WeatherModelDelegate** per la gestione asincrona degli eventi per la UI.

```
-(void) getCityCurrentWeatherbyId:(NSNumber*)city_id withSelector:(SEL)selector
ofObject:(id)object;{
   [[[NSURLSession sharedSession] dataTaskWithRequest:[self
setUpRequestAPI:@"weather?id=%@",city_id]
         completionHandler:^(NSData *data,NSURLResponse *response,NSError *error) {
            [self callCallback:selector ofObject:object withData:data];
         }] resume];
}
```

Per questa classe (a solo scopo di sperimentazione) sono stato usati i callback tramite **selectors,**callback tramite l'utilizzo dei **blocks** sono stati utilizzati nella classe **AnimatedBackground** e la categoria che la classe utilizza come input di dati **NSValue+AnimBackgroundData**, qui e mostrata la funzione helper che invoca il selector.

```
-(void) callCallback:(SEL)selector ofObject:(id)object withData:(NSData *)data{
   NSInvocation *inv = [NSInvocation invocationWithMethodSignature:[object
methodSignatureForSelector:selector]];
   [inv setSelector:selector];
   [inv setTarget:object];
   [inv setArgument:&data atIndex:2]; //gli argomenti 0 e 1 sono rispettivamente self e
_cmd, settati automaticamente da NSInvocation
   [inv invoke];
}
```

AGGIORNAMENTO ASINCRONO INTERFACCIA UTENTE

Ogni **WeatherViewController** detiene un riferimento alla classe **CityWeather** che viene utilizzata per ricevere gli aggiornamenti sia iniziati dall'utente sia quelli stabiliti dall'applicazione. Ogni **WeatherViewController** si registra come delegate della propria classe **CityWeather** in questo modo tramite l'implementazione del protocollo `WeatherModelDelegate` e possibile conosce quanto si e iniziato l'aggiornamento dei dati durante inizio dell'aggiornamento i corrente visualizzato **WeatherViewController** informa tutte le proprie views che un aggiornamento e stato iniziato e che quindi i dati che stanno attualmente mostrati sono probabilmente non attuali per creare una transizione non distruttiva e far capire all'utente che un aggiornamento sta venendo eseguito vengono mostrati al posto delle view le **SkeletalView** come mostrato nell'immagine qui accanto, ogni view gestisce le proprie skeletal views, che vengono tolte quando i dati sono nuovamente disponibili. La stessa cosa viene impiegata nella gestione della **WeatherBackgroundView** in questo caso la classe **AnimatedBackground** conensete di eseguire transizioni da un preset all'atro in modo non distruttivo. Alla fine dell'aggiornamento la classe **WeatherViewController** e nuovamente notificata che i dati sono disponibili oppure che e stato rilevato qualche errore e di conseguenza la classe informa tutte le sue views di aggiornarsi oppure di ritornare a una versione in memoria dei dai in caso di errore.