Scuola universitaria professionale della Svizzera italiana
Dipartimento tecnologie innovative

SUPSI

Corso di Laurea in Ingegneria Informatica

Anno accademico 2020/2021

Sviluppo di applicazioni "Mobile"
Wheater Mobile Application

DTI.M02074

Studenti: Luca Pasini
Sebastian William Nee

Relatori: Michel De Vittori
Samuel Poretti

Data: 30 aprile 2021

Indice

0.1 Project Requirements

The project requires the development of an Android application using Android Studio that is capable of checking weather forecast based on location.

The application must have two main views:

- A main view, shown at the start of the application, containing a list of locations added by the user, as well as their current location;
- A view that shows detailed information about a location, such as current temperature, minimum temperature, and maximum temperature;

The application must have the following functionalities:

- The ability for a user to insert a new location to the list of locations;
- The ability for a user to open a detailed location page by clicking on that location on the list;
- The ability to show the users current location on the list of places, obtained using GPS;
- Background monitoring of current location temperature to check if temperature becomes critical[1].
- Saving new locations locally, with data persistence for other app sessions.

The locations and their temperatures are loaded form the *OpenWeatherMap* API[2]

[1]A temperature is critical if it is within a pre-defined range
[2]https://openweathermap.org/

1 Implementation

In the following chapter the various techniques used to implement the project requirements will be explained.

1.1 Locations List and Detail Page

The main page of the application is composed of just one element: A *RecyclerView* that holds a list of all user defined locations.

In order to be able to make the *RecyclerView* work we need to implement also a *ViewHolder*, responsible of "holding" a single element of the user locations list, and an *Adapter*, responsible of "adapting" the list to each *ViewHolder* and it's fields.

In this case, the *LocationsViewHolder* has two fields: the data represented (class Location) and the *TextView* used to display the text for each entry of the *RecyclerView*. Its function is to bind values of the view to values of fields from the model (i.e. the location it stores).

On the other hand, the *LocationAdapter* holds the full list of location being represented, plus a location outside this list, used to hold the current location (retrieved from GPS coordinates[1]).

The method *onBindViewHolder* of the *LocationsAdapter* is called every time the list updates, for each item that gets updated, receiving the positon in the *RecyclerView* of that item and the *ViewHolder* that should represent it. This method is implemented in a way that makes the current location always be at the first position in the *RecyclerView* and all the other user defined locations right after that.

Additionally, in order to make more clear to the user that the first location is the one taken from GPS, the *ViewHolder* of each location receives a boolean variable upon binding, indicating if the location is the aformentioned, so that it can be represented with a label in front of it.

1.2 UserLocationsHolder with cached list and database persistence

In order to have only one place to hold location added by the user, the solution uses a singleton class: *UserLocationsHolder*. The objective of this class is to provide an easy-to-use source of all the locations to represent. In fact, it holds a list of all the locations added by the user, a current location field, and a reference to the *LocationsDatabase*.

[1] see section 1.5

Since every operation that interacts with the database has to be done on a separate thread, this class holds a cache of locations that is filled with all the locations stored in the DB upon initialization. After that, every time client classes need to get a location, the *LocationsHolder* won't have the need of requesting any data to the database, instead, it will return the reference of the location stored in the database.

On the other hand, this method is not achievable when inserting a new Location since it's the database that generates locations Ids, hence, methods that require thread completion receive a callback function that gets called when the operation is completed. For example, when the method *insertLocation* is called, a callback function that receives as input a location is passed to this method, and client classes can specify what to do with the location resulting from this operation.

1.3 API Connection and Requests

Connection to the *OpenWeatherMap* API[2] is handled by *Retrofit 2*[3], an HTTP client for Android and Java. Retrofit turns an HTTP API into a Java interface. The interface can make a synchronous or asynchronous HTTP request to a remote webserver, passed as an argument. Retrofit can convert the JSON returned from the API call into a callable Java Object. This allows the information to be passed and parsed easily to different sections of the code.

All of this is handled in the *WeatherFetcher* class, which contains two functions:

- *fetchWeather* fetches the weather info from *OpenWeatherMap* and returns a callable class that represents the JSON that the API call returned. It accepts one consumer as argument which defines what to do when the API call returns a result.
- *checkCity* checks if a city actually exists in the *OpenWeatherMap* database. It accepts two consumers as arguments which define what to do if the API call fails or returns a correct result.

1.4 Adding a Location

When the button to click a new location is clicked, a Dialog Box is created, which asks the user to insert a location. When a location is inserted, the *onDialogResult* function is called, which checks if the city inserted actually exists in the *OpenWeatherMap* API. To check if the city exists, the *checkCity* function in the *WeatherFetcher* class mentioned above is called. If the call fails, meaning the city does not exist, a Toast is created notifying user that the city is not found.

1.5 Retrieving the current position with SmartLocationLibrary

In order to retrieve coordinates from android localization services, this solution uses a library named SmartLocationsLibrary[4]. This library offer an easy way to configure location monitoring or location requests for Android.

In the explained solution, the class *SmartLocationController*, which is a singleton, represent a simple interface between client classes and the library. It offers three main methods:

- *startMonitoring* as the name states, is used to initiate a continuous location detection that runs a callback passed as parameter upon a location change;
- *requestLocation* offers a way to get the current location only one time, without setting up a listener.

[2]https://openweathermap.org/
[3]https://square.github.io/retrofit/
[4]https://github.com/mrmans0n/smart-location-lib

- *getLastLocation* is a method that return the last location returned by the listener configured with the aforementioned method, without any need of interaction with the location library.

1.6 Background process and notifications

Background Processes in Android are handled by extending the class *Worker* and overriding the method *doWork*.

In this implementation, the class BackgroundWorker uses the instance of *SmartLocatonController*[5] to retrieve the last known location, then it will use APIs to check for the current temperature. If it's value is outside of the safe range it will send a notification using Androids Class *NotificationManager* combined with the *NotificationCompat.Builder*, which is in charge of building a notification object.

This background worker is launched conditionally in the MainActivity only if there is network to make an API request, once in range of 15 minutes. To start the worker the solution uses a *PeriodicWorkRequest*, configured with constraints and built upon the worker, then passed into the *WorkManager* singleton instance of Android.

[5] see section 1.5