WEATHER APPLICATION REPORT

660037119

Design Overview

In terms of the overall project, the idea was to make a main home activity that displays in detail the current weather. These details would represent the closest forecast information available to the current actual time, since the data is provided in 3 hour intervals. There will also be a visual list on this screen representing brief information about the weather for the next 4 days. Upon clicking one of these list elements, a detailed view will be produced of the weather overview for the selected day. This home screen also includes the functionality to select your location. Weather data is also stored in a persistent database allowing information to be viewed without an internet connection. By implementing all of these, I have met the provided specification. I will outline in detail each component mentioned in further detail within this report, as well as any additional features I have also included, with justification.

Home Screen

Firstly I will cover in more detail the high level design of the project. The home screen as previously discussed, shows the most up to date detailed weather information available from the 'open weather map' API, respective to the current time. For example, if the current time was 20:30, the detailed information for 21:00 would be shown. The home screen itself is an activity acting as a parent to many fragments. The current detailed view for today is one fragment, and the future forecast information is also composed of fragments. Each fragment has associated with it a java class for individual logic and at least one xml file to specify the layout. Using this fragmented approach allows easy manipulation of segmented logic sections in the view. This is vital for adapting to circumstances such as screen rotation.

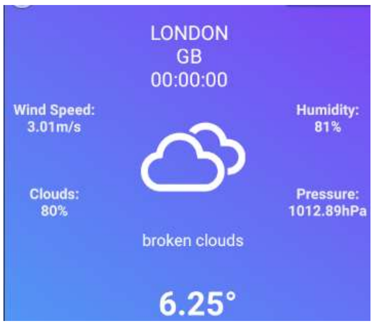


Fig 1.1

Fig 1.1 represents the fragment for displaying today's current detailed weather forecast. The city, country and forecast time are displayed at the top, followed by an icon graphically describing the current weather situation with it's textual description directly beneath it. The detailed information surrounding this includes temperature Wind Speed, Humidity, Pressure and Cloud Coverage. The implemented class associated with this fragment contains the logic needed to render this information from the database and populate the view.

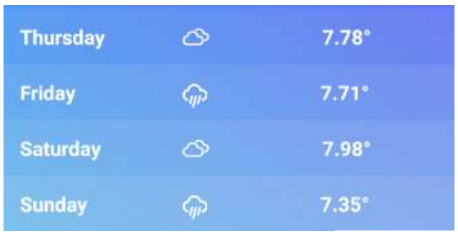


Fig 1.2

Similarly I have a fragments for each of the future forecast elements shown in **Fig 1.2**. Basic information includes the day of the week, weather icon and temperature only. I chose the information supplied at midday for this information as a fair overview for the day. Similar to the 'today' fragment, the classes associated with these fragments are responsible rendering and displaying the relevant data from the database to the view. The parent activity also binds an 'on click listener' to each of these fragments, upon clicking will send an explicit intent to the 'detailed activity', with an extra integer as to how many days in advance the forecast day clicked is. Using this information, the same 'Detailed Activity' can be used to construct all the needed different detailed views on demand. The 'Detailed Activity' takes the same format as **Fig 1.1**, taking up the whole screen to show the details about the weather forecasted for this day. Again, as with the basic forecast, I have used midday to represent an average detailed overview for the day.

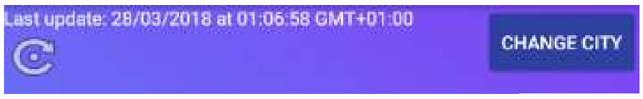


Fig 1.3

Looking back at the home screen, we can see there are some components existent in the home activity itself, as shown in **Fig 1.3**. Additional features outside of the specification I have added includes the last known update field, this allows the user to see how up to date the current stored and displayed weather information actually is. Even though automatic refresh is also a feature, I have also added the functionality of a manual refresh represented by the circular arrow. Upon clicking, the registered 'on click listener' will perform an immediate request for new weather information and display it, details on the actual data retrieval process covered later. Even though these features are not defined in the specification, I believe they will be of great convenience to the user. On the right we see the 'change city' button, this creates an explicit intent and starts the 'Location Activity'.

Location Selection & Shared Preferences

The 'Location Activity' allows the user to select a city from a list of locations, these location are defined in the project resource as an array of item in an xml file. This data is utilised by an adapter and then bound to the list view in 'Location Activity', since location activity extends 'List Activity'. An 'on click listener' is registered to the list view, following a click, the element location clicked is determined at runtime. This determined location is then stored as a shared preference, allowing the selected location to be accessible application-wide, and maintained following app closure and re-opening. Another example of shared preferences that I am using would be the last update time, since this also needs to maintain state throughout multiple loadings of the app. In order to provide a convenient interface for my classes to work with shared preferences, I built a class for each shared preference used. This promotes code clarity and acts as a useful interface for manipulation of shared preferences, performing functions such as defining default values if they are not yet instantiated.

Following the city preference being set, an explicit intent is then sent to the 'Home Activity' in order to resume it. By using the adapter binding method to display the list of locations, it would be simple in future to implement the ability for a user to add or remove locations dynamically if needed.

Database

In terms of storing the data, I have a class 'Weather Database' with the purpose of defining the contents of the database. This class extends 'SQLiteOpenHelper', allowing organised and effective creation and maintenance of the database. I also developed a database interaction class ('DatabaseCommands'), this class acts as an interface for all interaction with the application contextual database. Having this level of abstraction means that raw SQL never has to appear elsewhere in the codebase, instead only method calls to a 'DatabaseCommands' instance is required.

The class itself uses the singleton design pattern meaning there is only ever one readable and writeable database reference created per session, which is efficient as obtaining these references is expensive. This also secures an implicit synchronicity of all transaction occurring in the database, useful if different threads are accessing the database simultaneously. The readable and writeable database references are lazily instantiated in the class for further optimisation.

Retrieving, Rendering & Updating

When the app is first opened, it immediately attempts to update the stored weather data for all supported locations. The specification made it clear that lack of internet was an important feature to be supported. Therefore, by performing all needed internet dependent operations upon app start, internet loss during app operation is not an issue. This also avoids the user needing to click on every location in order to update the relevant data in case they lose internet, which is very tedious. If the ability to add or remove locations was to be implemented, it would be maxed at 5 to keep performance reasonable. A manual update is also supported as highlighted previously.

I have a class 'RetrieveData' dedicated purely to establishing a HTTP connection and retrieving a JSON object from the API. This class returns null if the internet request was unsuccessful, if this is the case, the app will keep the previous 'last updated time' the same, as well as display a toast to notify the user of the internet connection failure. The app with then continue running normally using the data already stored in the database if there is any. (Note it takes a short while for the app to determine it has no internet on start-up after searching for a host name) . If there is no data in the database to begin with, a message is displayed signifying that no data has ever been downloaded on this current install.

Since the retrieving and storing of relevant data is a very intensive process, it cannot occur on the main UI thread. As such, I have a class 'UpdateStoredData' which is an asynchronous task responsible for utilising 'RetrieveData' to get the JSON objects and storing only the required results in the database. When the app attempts to download new weather data, the UI thread executes this asynchronous task as a new thread, offloading the computation from the main UI thread and preventing a UI lock. Whilst the data is being downloaded, the user is notified of the progress via a progress dialog spinner on the main UI as shown in **Fig 1.4**. Once retrieved and stored, the data can then be accessed by all classes in the app via the database and rendered to respective views. Since rendering data from the database makes no noticeable delay to the UI thread, rending can occur on the UI main thread.

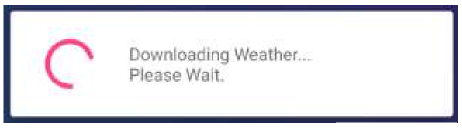

Downloading Weather...
Please Wait.

Fig 1.4

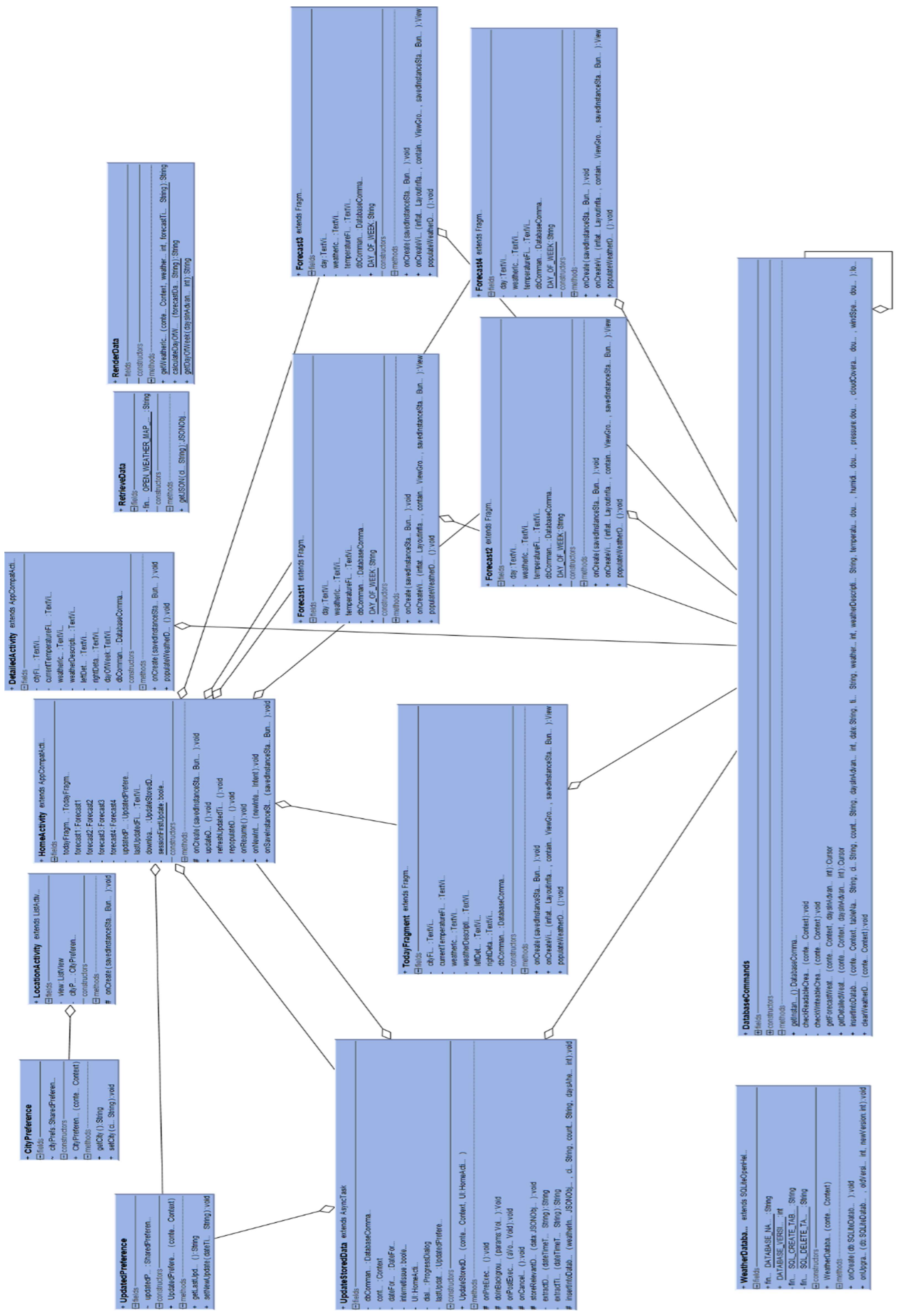
Optimisations

I have made a few optimisations in terms of avoiding unnecessary retrieving or rendering of data. When the app returns to the home activity via an explicit intent, instead of performing the default activity lifecycle behaviour of re-creating the activity, the app uses the same activity that was running previously. This is achieved with intent flags, and avoids redundant data downloads, database fetching, rendering and repopulation. I also keep a Boolean value 'sessionFirstUpdate' in the home activity class, this ensure that upon device rotation, the data is not downloaded again unnecessarily. This would have been caused by home activity recreation upon a device orientation change.

Orientation and Screen Sizing

In terms of compatibility, I have constructed multiple xml layout files where necessary to ensure the view is function in both landscape and portrait orientations. To help support multiple screen sizes, I have avoided using hard coded widths and heights, and have used 'match-constraints' or 'wrap-content' instead in a constraint layout. Finally, an issue with orientation occurs if an asynchronous download thread is running whilst an orientation takes place. This is because is attempts to run on the now destroyed old UI thread post execution, causing a crash. In order to deal with this, I check if an ‘Async’ thread is active upon ‘onSaveInstanceState’ called due to an orientation change. If so, I cancel the thread, and start a new download on the new created newly orientated activity. This maintains app stability.

Overall this app has aimed to satisfy the specification, providing ample stability when there is limited internet access via a database. It displays all details mentioned in the specification. It also incorporates some unrequired functionality discussed in this report, for the user’s convenience, app stability and performance. Finally, the generated UML diagram follows:



RenderData
RetrieveData
Forecast3 extends Fragm...
Forecast4 extends Fragm...
Forecast1 extends Fragm...
Forecast2 extends Fragm...
DetailedActivity extends AppCompatActi...
HomeActivity extends AppCompatActi...
TodayFragment extends Fragm...
LocationActivity extends ListActiv...
CityPreference
UpdatedPreference
UpdateStoredData extends AsyncTask
DatabaseCommands
WeatherDataba... extends SQLiteOpenHel...