CSCI 4176: Mobile Computing

Assignment 2

Seyd Razavi Lopez

B00751312

Intro:

The main purpose of this application was to develop a weather forecast app following the Nielsen's Usability Heuristics. This document explains the different approaches I followed to develop the app.

Software Design:

The application follows a very close design to the application submitted in Lab6. This is due to the fact that lab 6 required the same technologies as this assignment. That is, the volley and the org.json libraries since the android documentation also suggests them.

The volley library was is required for http requests and response. Hence a singleton pattern would be beneficial since the app can make many requests. A singleton patter would allow us to have a queue with all the requests and retrieve them as needed (suggested In lab 6).

The org.json library is a light json library that allows us to manipulate json objects. This is quite beneficial since many API requests return a string that can be translated into a json object (key-values).

The application contains 4 major classes: MainActivity, RequestSingletonPattern, WheatherItem and Utilities.

MainActivity: This is the main view of the app. All the logic is implemented here since it is a very basic application (only one view) that requires interaction with the user. This activity contains all the necessary elements to make the app work (mainly API calls and Event Listeners). Because the application has only one view which displays a single weather information item, this class also has a single WeatherItem object to maintain the information (Singleton). This allows us to update the real time information on the object and perform operations (eg, from metric to imperial) without retrieving the data from the UI, simply updating it. Due to the nature of activities in android, to perform a singleton patter, we simply need to create the object on the create method.

RequestSingletonPatter: All the API request are stored here. A single data structure (a queue) contains all the requests and there is only one queue for the app (singleton pattern).

WeatherItem: This class is a value holder for the values from the API call. In other words, this class is the link between the MainActivity and the RequestSingletonPattern. An api call is request. Once received, a WheatherItem object is created with the necessary values from the api response. This object is passed to a method that populates the UI elements so the user can see the response. The WeatherItem object is treated a singleton (only one instance), that is created on the OnCreate method in the main activity.

Utilities: This is a static class that simply modifies the temperature from C to F and vice versa.

Implementation problems, solutions and tests:

As previously mentioned, this is a very simple app containing only one view. Hence, the major focus was to develop an intuitive app with proper error handling, error prevention and without overloading the user with unnecessary information.

The major challenge of this app was to understand the nature of any API call. While the api call can be successful, it does not mean that the returned information will match the coded json structure or the keys. Hence, I first focused on displaying all the data available if the API call was successful. If the data returned did not match our structure or was missing, a N/A on that field would be displayed.

To achieve so, I created 3 major JSONObjects from the main one. For every single one, I retrieve the information that was needed. Android allows to use the opt_ method that allows to place a default value (in this case an ERROR_FLAG) in case that the return object does not contain the value or is missing. Hence, the user would be able to available data. This was tested, by modifying the expected data structure so the main view would show N/A.

The second challenge of this app was error handling. Fortunately, the volley library contains different exceptions that can provide information about the error. For this app I implemented and tested 3 errors:

TimeoutError: If the response takes too long, the user is informed and suggested to try again. This was tested by blocking the connection after the API call.

NoConnectionError: Since this app requires internet connection, if there is no internet connection, the app lets the user know that the device is not connected to the internet and suggests connecting it. This was testing by turning off the connectivity of the device.

ServerError: This is the 404 error, meaning, that the server side could not make sense of the request. For example a unknown city. If this is the case, the app lets the user know that the city could not be found. This was tested by adding extra letters to known cities.

Others: any other error is displayed as: Something went wrong.

Because this app is thought to be used by the user for a very short period of time and not much attention from the user is required, I decided to display the error as a textview and hide all the elements required to see the information (all or nothing approach). This is because a toast can only be displayed for a short period of time and will not caught the attention from the user since the user, in the first place, may not be paying attention. Hence, if an error occurs, the user will see it and the only action after that is to connect the device, enter the city again or try again. In other words, it forces the user to fix the issue as opposed to provide false information (for example a previous API request).

The third challenge was error prevention. I tackled this issue by cleaning up the string use in the API call. For example, a name should not have numbers, be empty or the format can only be city,country or city. This was implemented on the click listener before the API call. If any of these occurred, the application would provide the proper message. Furthermore, an API call may not necessarily be fast, which may get the user frustrated and make more API calls. For this reason, a message is displayed informing that the data is being retrieve and the main button is blocked during this time, until a response (error or successful) is received.

The last challenge was to adequately inform the user about the app. As suggested in the assignment, we place an example as a hint on the editText. Also, if a formatting error occurred, the error message informs how to use the app.

The documentation clearly displays that the API call can take a city name, a country and a metric. For example, Halifax will retrieve information from Halifax in Canada, however, there is information about a Halifax in US and GB. For this reason, I placed some other cities as autocomplete option so the user knows that further information (mainly the country) may be required (type Halifax). Also, the user has the option to retrieve the data in F or C. All these implementations were tested via web-API and app.

Screenshots: