Quick Reports App: Final Documentation

Jesse Willams, James Jasinski, Matthew Coleman

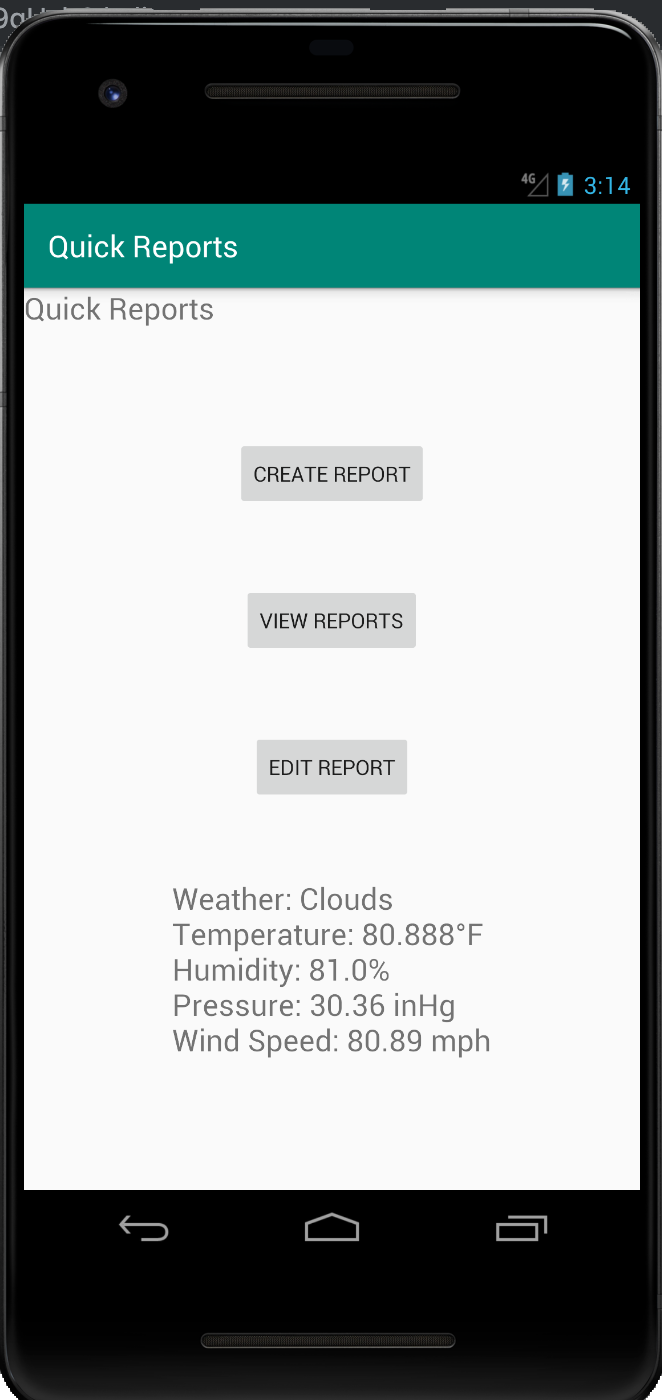
Functional Description:

From the beginning of this project we have set out to create an effective and well documented application that any user could use with relative ease. Out of this drive we developed the Quick Reports application. Its base function is to act as an easy way to document any sort of incident that may happen to the user and allow the user to keep those documents for later review. The application allows the user to report any information on the incident as well as enter their name and an id for the report. Once that is done the report is stored in the database along with any images that may have been taken alongside the report. This report can then be viewed whenever the user wishes and can even be edited. Given permission, the application automatically obtains GPS coordinates and calls an external weather API to retrieve the weather at the location of the incident; this will allow the user to document the weather as needed in their report.

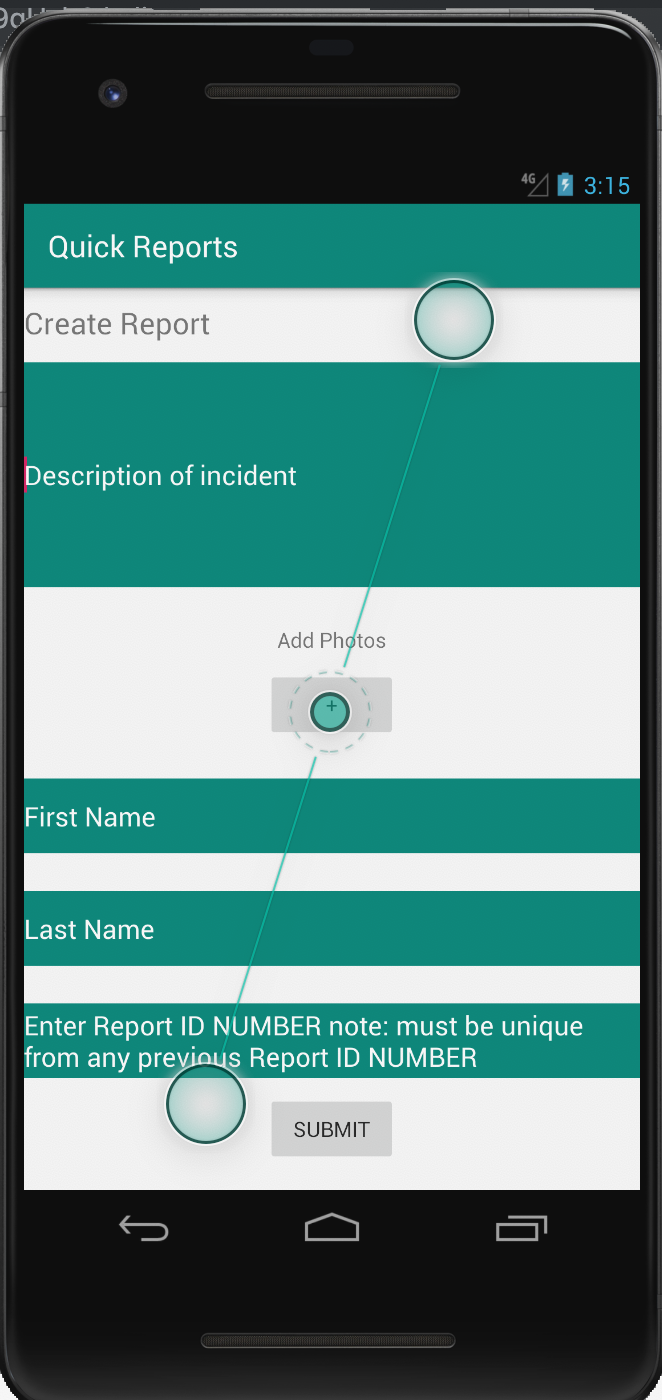
Step by step Instructions:

Step 1 Open App

Step 2 Select Operation



Step 3: Fill out form if selected operation requires it



Step 4: Hit Submit or Update if selected operation requires it.

Software Design:

The main change made between the design documentation and now would have to be the view model class. Before implementation began I had believed it to be more or less a derivative and mostly empty class. But as implementation went on it became clear that there were several major sections of code within the view model that helps it all work in the end. As such the current view model is different from the view model expressed in the UML. A less complex design is used for accessing the weather API compared to the original UML design; all functions for retrieving weather information are consolidated within a single class which runs in the background instead of being spread among helper classes. Additionally, the original design featured a string return value, but the implementation has no return value. Instead, it stores the result (which is still a string) in a designated text field.

Process Description:

As far as testing our application goes we are planning on using White Box testing to ensure that each path the user can take functions correctly, from entering all the information or leaving one or two fields empty to entering completely empty reports. We hit a rather large hurdle when we ran into the issue of the view model. Trying to use the database without it was nearly impossible and that is when it became clear that it needed to be fleshed out more than it already was. Creating it and adapting it to the report module proved to be slightly more difficult than we had imagined. As far as incomplete features go we could clean up the database a bit more and we could not get the list from the database to display or hold data as it is supposed to. Also there is no camera functionality currently as clicking the button will do nothing at the moment. The way we store our images seems to be a little clunky and it would be interesting to look into it more and see if it is possible to store them in a more streamlined way. The database would take in, or at least appear to take in the report and picture from the user but when the list was checked we would end up with a null pointer error. This error would need to be fixed before this app is actually published. However we were able to streamline the SQLite database system by instead using the Room database system. Room acts as an encapsulation around SQLite to allow for coding and information that more closely matches basic SQL queries and styling. If we were to do the project for a second time then we would spend more time on the UML document as well as try to search for coding errors earlier in the coding process. The UML document we initially made helped with our design process but could have been fleshed out more than it was.