Professional Object Oriented Programmers

Michael Aliberti, Liam Murray, Alexander Pozin, Theodor Waalberg, Dmitry Zimin

EC 327

Professor Densmore

7 December 2019

Aero Nav Project Documentation

Project Description:

Aero Nav is an android app game which presents the user with an icon indicating the location of a plane and a line representing the path from the plane’s point of origin to a random location mid-flight. The user is presented with four potential cities, one of which is the plane’s destination. The locations of these cities are also plotted on the map. The user is then able to examine the map and use logic and a bit of luck to attempt to guess which city is the plane’s destination.

Challenges:

Various challenges were encountered over the course of this project. Android Studio cannot emulate devices on certain products, making it impossible to properly test code on certain group members’ computers. Another challenge came in the form of OpenSky API. While OpenSky API proved to be the best option for collecting airplane information in that it is both free and thorough in its data collection, the provided code for accessing and parsing data in Java was defunct. However, the website also provided guidelines for formatting http requests to their REST API. By employing the Volley library and including internet permission in the App’s Manifest, a url string can be passed to the REST web service, returning either a JSON array or a JSON object.

Despite being able to retrieve data, another issue then arose in terms of what data could be retrieved. For the app to function, it needs to collect at least one pair of latitude-longitude coordinates mid-flight, the origin airport, and the destination airport. Due to the nature of the API, this required two separate API calls. While the “State Vectors” call - which provides real-time tracking information and allows inquiries to be bounded within a limited radius of the user - was one of the initial frontrunners, it turned out that it could not be used in conjunction with other calls to get departure and arrival airports, the data for which is on a full day’s delay. This narrowed down the main call to the “Flights in Time Interval” request, which returns flight all flight call signs, departure airports, and arrival airports within a given timespan. When supplemented with the “Track by Aircraft” request, which provides historical data on a plane’s route, the program would have all of the tracking data it needed. As these website calls require unix time stamps in their urls, unix time is calculated on-system, converted into a string, and concatenated to a base url string before being sent to the website. In order to accommodate for the API’s data delay, all calls are made for data collected two days prior to the data of access."

Front End:

The GUI consists of three pages: the main page, the about page, and the page for gameplay. The main page is designed to be aesthetically pleasing and intuitive. It contains two buttons, one of which says “ABOUT” and the other says “START”. The about page contains a small amount of information regarding this group. The play page presents a map, four destination choices, and a way to exit back to the main page. The four choices are set up as buttons, where one button is randomly assigned the correct answer. When a button is clicked; if wrong, the button will turn red; if right, the button will turn green. Play continues until the player exits out of gameplay.

The map interface consists of several parts. There is the map itself which takes the typical google maps informative appearance as well as several markers. One marker represents the plane’s location and another it’s origin; these two points are then connected with a line to indicate a rough facsimile of the path the plane has taken. Four other markers appear on the map at the locations of the four potential destinations. The plane is indicated using a marker of one color, while the destination markers appear with a different color. The map is integrated into the play page in such a way that it is interactable. The user can zoom in and out and scroll across the map. The map was also implemented in a way so that attempts to drag or otherwise modify markers would neither modify the markers nor adversely affect the program. The goal in implementing the map was to afford the user the most information with the best accuracy for their guess without sacrificing the stability or security of the app.

Back End:

Upon pressing the “START” button, the program uses a call to the OpenSky API to collect data from a two hour period two days in the past - a process which takes about five seconds but varies with the day. Every screen thereafter, the data is parsed with a static iterator, and the callsign of the next plane with a full data suite is in turn used in the “Track by Aircraft” call to get mid-flight information.

The AirportsFinder API is accessed next. The destination and departure airports have their three letter codes passed to this API, which returns an array. This array is in turn parsed for the coordinates of these airports as well as the names of the cities in which they are situated. The coordinates and names for three other random cities are also gathered, found by feeding random strings of three letters into the AirportsFinder API and error-checking to ensure and output..

All of these coordinates are then passed to the Google Maps API. This API plots the coordinates with the desired marker colors and a line connecting the point of origin to the plane’s midair location. The names of the cities corresponding to the possible destination airport waypoints are then displayed as four separate buttons, with one representing the correct answer, and the others incorrect answers.

With each guess by the end-user, a delay will be put in place before the next plane is selected, picking up where the static iterator left off. This process repeats in perpetuity until the user presses the “BACK” button.

Error Checking-

Error checking is necessary in order to present all of the information necessary to play the game. The game requires mid-flight, point of origin, and destination coordinates, as gathered from OpenSkyAPI. However, not all flights provide all three data points. As a result, before a flight can be presented to the user, the program must first ensure that all three points are accounted for. This is done by iterating through the initial data set and only sending full data suites to “Track by Aircraft” as explained above. Error checking also occurs when retrieving the city names to be used as incorrect options. Since random strings of three letters are sent to the API, it is possible that some of these will not return anything. The code is designed to handle an invalid airport code and send another if need be to create three wrong possibilities total.

Security-

In terms of security, there is no way to directly access the code from the constructed app. The user has no way of accessing the code from within the GUI nor of modifying the elements which make up the GUI.

Marketability:

The market for this product is quite broad. The design is purposefully simplistic in its GUI and concept in order to make it available to children but also palatable and interesting for adults. This game could reasonably appeal to anyone with an interest in aeronautics, aircraft, or geography, serving as a test of one’s knowledge of aviation routes but also of city locations. While it can be used as an educational tool in that sense, its ease of use and casual nature makes it an ideal game to play for consumers during short periods of respite during the day.

Targeted Description:

Aero Nav provides a fun way to interact with real airplanes all over the world. This game presents the player with a map of the world and a pin representing the location of a real plane just recently in the air and a line drawn between it and its point of origin. Four possible landing spots are also depicted on the map. By pressing one of four buttons, players may test their ability to discern the plane’s true destination.