Reflections

For the School Progress Tracker App project

Andrew Cate

1-28-2023

Version 1.0

WESTERN GOVERNORS UNIVERSITY

CONTENTS

I. INTRODUCTION

In this document I describe my experience designing the Student Progress Tracker application, including the challenges, solutions, and alterations I would make if I was to do the project again. I also describe the operating system version the application was designed for, how I would design the user interface for a tablet, and the role of emulators in application development.

1. TABLET VERSION

The main consideration when thinking about running an app on a tablet is screen real-estate. With a larger screen size, the same layout used on a phone may look odd and be awkward to interact with.

Layouts and fragments are containers for organizing and displaying user interface components. The big difference is that fragments are much more flexible. Fragments can be reused or swapped out when the need calls for it. Fragments can be put side by side if space allows or can be stacked on smaller screens.

For example, a tablet interface can use a standard layout to display UI components that don't change, like the menu, and use fragments to display the main application content that changes depending on user interaction.

If I were designing this application for a tablet, I would use a standard layout to contain the menu on the left side of the screen, and a pair of fragments would take up the remaining screen space. One fragment would display the list view of the terms, courses, or assessments. The other fragment would show the detailed view of whatever item was selected from the list fragment.

2. OPERATING SYSTEM

This application has been developed for Android 12, version 12.1, API level 32.

The minimum operating system it is compatible with is Android Oreo, version 8.0, API level 26.

3. CHALLENGES

On the topic of challenges during the development of this application, two primarily come to mind.

Programmatic Buttons

The first challenge was with the look of the user interface, specifically the color and shape of the buttons. I programmatically added a DELETE button to the detail screens and could not figure out a way to match the default look of the declaratively created buttons. The declarative buttons have rounded edges and use the primary color as a background. When I programmatically added a button, the style, font, and text color matched nicely, but the button had a light grey background which didn't match the primary color. When I changed the background color, the button would drop its previous style and become oversized, blocky, and the font would change slightly.

Date Validation

The other, much more time-consuming challenge was with date validation. Each of the detail screens allow you to set start and end dates. I wanted to add logic that would validate the dates when the SAVE

button was clicked. If the start date was after the end date, or the end date was before the start date, I wanted to show a message and not allow the item to be saved. Over many hours and many attempted solutions, I could not get it to work without failing. At first, it would work unless the dates were the same *and* the start date was the most recently changed. Next, it would work at first, but then fail completely and let you save with a start date after the end date or and end date before the start date. A step backward to be sure.

4. SOLUTIONS

Of the solutions to the two challenges described above, one was satisfying, and the other disappointing.

Programmatic Buttons

As a first attempt to solve the problem of the button style changing when I modified button background color, I tried to locate the code for the default button style. The hints I could find referenced `styles.xml` which was not a file I could find in my project. Next, I looked through every button related object I could find in the API, searching for some indication that would give me insight into a solution. I kept returning to the themes used in my project. I knew that somehow, they had to be involved. Finally, I discovered that each theme has its own button objects. The theme I was using was Material, so if I used a `MaterialButton` instead of a `Button`, the default theme applied to my programmatically added button, and it worked. I was very satisfied when I figured this one out.

Date Validation

As you may have deduced, the solution to the date validation problem was not as triumphant. I began doing the date validation using `Calendar.after`. That worked unless the dates matched *and* the start date was the most recently modified. Imagine how long it took me to notice that. That major clue helped me realize that `Calendar` was also using time for its comparison, not just the date. The next solution I tried was to extract `Calendar.YEAR` and `Calendar.DAY_OF_YEAR` and compare them. That worked, at first, and then failed miserably, as I described before. I delved into the `Calendar` values with the debugger and discovered that sometimes, the date for one of the date fields would somehow be set to the *current date*, but the *displayed date* for the same field would still be accurate to whatever was selected from the date picker (you pick the first day of the year and behind the scenes it is set to today). No matter what I tried, I could not determine where the change was happening or what was causing it. After many hours of troubleshooting, I gave in and got help from a Course Instructor. Their suggestion was to extract the displayed date, convert it to a `String`, and then do the comparison. That is ultimately the solution I went with, and the way I was sending the data to the database anyway. I am disappointed because I didn't discover the source of the bug, but the workaround works well.

5. ALTERATIONS

If I was able to repeat this project, knowing what I know now, most of the things I would do differently revolve around visual design and quality of life improvements.

Material Design

About midway through the project, I re-discovered Material Design. I really like the simplicity and uniformity of the design philosophy. I had just recently finalized the design of my screens and couldn't spare the time to modify them at that point. If I were to redo this project, I would change all the fields to have a more Material Design look and feel with reactive borders around each `EditText` field, using a hint label instead of a `TextView` label. This would simplify the user interface considerably which would, in turn, simplify the code.

Quality of Life

As is often the case, quality-of-life improvement discoveries happen during testing, which occurred too late to include in the project this time around. I would have loved to have a Home button to return to the Main Activity screen while testing out the various functions of the application, especially from a deep screen like Assessment Details, which is only two clicks away (three at the deepest). Not too bad from a user perspective, but during testing when you are doing that repeatedly, it takes a toll (especially with a slow emulator).

The other quality of life improvement was discovered by a user testing my application. They suggested that when the start date field is picked from the calendar, the end date field should auto-populate with the same date. This would be especially helpful with the Assessments, which are usually scheduled to end on the same day that they start. This was a valuable reminder to me to test early and often, rather than at the end.

6. EMULATORS

Emulators are necessary and useful but cannot completely replace the real thing. Emulators are used to simulate a device and run on a computer. They mimic many of the functions of real devices and offer some advantages for testing over real devices.

Emulators are cheap, even free, and you can obtain an emulator for most devices. They can replicate many of the functions of the real device, including physical hardware simulation like motion sensors. This allows you to test your software with a wide range of devices for little cost.

Unfortunately, because an emulator runs on a computer, it doesn't completely mimic the performance and ability of the real thing. They use your computer's CPU, which behaves differently than the device's CPU would. There are also many things that you cannot test with an emulator, such as the stability of the provider's network. Device manufactures often modify the phone's operating system to offer more or unique functionality over the default Android configuration. These changes impact the way the device behaves and are difficult to predict with an emulator.

Actual devices, on the other hand, demonstrate how the application runs on them precisely. You can test all the hardware, network, and manufacture software impacts on your application. However, this

benefit also has some steep drawbacks. You must obtain each device you want to test, which is expensive. It also might take longer to run tests on the device, depending on the computer hardware you are running your emulator on.

To cover all the bases, the best strategy is to determine which tests are safe to run on an emulator, which to run on both, and which can only be run on a physical device.

7. SOURCES

I accessed this through the Android Studio IDE software but will include the online version here.

- "Android API Reference : Android Developers." *Android Developers*, Google, https://developer.android.com/reference.
- Android Studio Dolphin. 2021.3.1 Patch 1, Google, October 13, 2022.