Project Reflection Essay

WGU Scheduler App by Leonard Erwine

Leonard Erwine

Western Governors University

000356334

WGU Scheduler App by Leonard Erwine

When developing applications for mobile technologies, consideration must be taken for the limited resources of small, portable devices. These limitations include smaller screen sizes, as well as reduced memory and storage sizes (Manage your app's memory, 2020) when compared to laptop and desktop systems. Not only are the typically reduced screen sizes considered, but the range of differing screen sizes between different mobile device types is greater than the range of sizes between different models of laptop and desktop systems. Therefore, it is much more difficult to create a single UI layout that will look good on all supported devices than it would be for a desktop application (Build for Billions, 2020). Performance issues must be considered as well, in order to prevent the battery from draining too much battery life (Optimize for battery life, 2020). Lastly, features such as cellular communication, biometrics and GPS tracking are much more common with mobile devices. Although not technically a development constraint, some features have come to be expected with modern mobile applications, such as support for automatic screen rotation (Core app quality, 2020).

This application was developed targeting API level 30 (Android 11, 2020). The minimum OS version the app can run on is API level 26 (Android Oreo, 2020). This is because I wanted to use Java 8, which includes classes which were introduced with API level 26. The app was tested using emulated devices with API level 30. This was also tested on an AVD with API level 26, just to verify that it can indeed run on the declared minimum API level.

One of the first challenges I had was that some input fields were either too short and hard to read or would render partially off-screen. One specific challenge was the notes field, which needed to be multi-line. Quite often, the text box would extend mostly beyond the bottom of the screen or would be partially obscured by floating action buttons. It was very difficult to build a user interface that would fit on the display and still be intuitive in their placement. Adding to the challenge was that may of the activity layouts became very large and unwieldy.

Aside from the visually oriented problems related to layouts, I had also experienced problems with unexpected validation results and slow responsiveness, especially in the course edit activity. On top of that, I was even getting blue screen errors on the host operating system, and even had to restore my entire laptop two time. I was getting seemingly random null pointer exceptions, and some fields were showing validation errors when they shouldn’t have, and vise-versa. It was also very difficult to figure out what caused the app to slow down. After using the performance monitor and tracking down the source of any remotely possible memory leaks, the problem persisted. It was hard to determine whether the host environment failures were related to the validation and performance issues or even what the root causes were.

To resolve the issues with the user interface, each activity required different approaches in order to make the input screens usable. Combining scrolling containers (ScrollView, 2020) with constrained layouts (ConstraintLayout, 2020) helped in most circumstances. Placing a margin at the very bottom allowed the user to scroll up far enough so the floating action buttons didn’t obscure it. I also found it useful to put user input views in a separate fragment (Fragments, 2019), so the differing functional groups of components within each activity can be edited and maintained separately.

I discovered that the issues I had with validation and slow performance were related to similar root causes, and that the cause of the host OS failures was due to an IDE configuration option. In a previous attempt to track down an error within an asynchronous process, I had enabled an option where the debugger would break on any exception. When I cleared this option, I no longer got any blue screens in the host OS. As for validation and performance issues, this was caused initially by the fact that all validation calculations were invoked synchronously by the event handlers which triggered them, which means they were being done on the main application thread. When I attempted to remedy the speed issue by doing some of the more expensive validations asynchronously, I ran into problems where the calculations that were still being executed synchronously were causing unexpected results and even null pointer exceptions when the asynchronous and synchronous operations were updating a value that both were dependent upon. To resolve this, I used RxJava (ReactiveX/RxJava Wiki, 2017) classes so I could execute all validation routines on a thread different from the main thread. I first created a class which encapsulated a Subject object and exposed an Observable object whose downstream changes would be observed on the calculation thread. Values which were previously being stored in simple fields were replaced by this class or converted to calculated observable values, so their changes could be observed asynchronously. I then employed value mapping and aggregation of the observed source values to produce values that could be applied directly to specific properties or methods on UI elements in the main application thread, such as setting the error message, setting text values, or enabling buttons. I then created another class which encapsulated an observer and a LiveData object, whereby values produced on the calculation thread would be observed and then posted to the LiveData object. The LiveData object could then be exposed to the fragment or activity class, where results would be observed on the main application thread and applied to the intended property or method. I also created a common base view model class which all view models inherited from. This base class could properly detach all listeners once the view model object was no longer needed, thereby avoiding memory leaks. This combination simplified the code, made it easier to read, and allowed me to track down and resolve any true validation irregularities. These re-used classes also ensured that calculations and UI updates were each being done in the proper threads, resolving the performance issues as well.

If I were to do this over again, I would not try to take the fast track for completing the project. I learned that attempting to put all functionality of an activity into a single class resulted in unwieldy code that was hard to navigate, debug and organize. It was much more frustrating and took quite a bit of time to track down all the bugs and to do all the necessary refactoring to make things right after the fact. I would also be more deliberate when defining calculation triggers and routines, so they don’t get executed more often than necessary. I would also be more careful where and how I utilized observers, so as not to cause memory leakage or produce exceptions that would occur at a point where the origination of the unexpected value was difficult to determine. I learned that creating LiveData objects directly on the view class that observes it can present a confusing paradox as it relates to the lifecycle of the view, and I would realize that if I needed a LiveData object on the view class, then that means there is probably something wrong with the overall logic.

Emulators are used by launching virtualization software which executes the actual mobile operating system upon which the target app is deployed. The virtualization software utilizes a virtual device, which defines characteristics of a specific type of hardware, translating input and output between the host environment and the emulated mobile device operation system. By using an emulator, the app can be tested on multiple hardware models and configurations without having to purchase every possible type of hardware. Device emulators are also typically more generic in nature and do not have vendor-specific modifications or restrictions which might complicate testing and validation. Emulators, however, require more resources from the host operating system in order to achieve the same performance as a physical development device. Currently mobile device emulators are also prone to corruption within the virtual device, and a failure within the emulator can also cause a failure on the host operating system.

Utilizing a physical development device provides a more true-to-life experience for testing an application, whereby the tester performs physical interactions with the device in the same way that the end user does. For instance, the emulator can emulate the appearance and interaction through the on-screen keyboard, but the tester cannot evaluate the actual user experience of physically touching individual letters. Typing on a windows keyboard is typically easier than using the android keyboard, and using a mouse to touch the keyboard is typically more difficult. This could give a skewed assessment of the overall usability of the app. Physical devices also require less resources from the host operating system and can make the repetitive cycle of deployment and debugging/evaluating faster and less prone to environment-induced failures.

References

*Android 11*. (2020, 09 21). Retrieved from Android Developers: https://developer.android.com/about/versions/11

*Android Oreo*. (2020, 07 21). Retrieved from Android Developers: https://developer.android.com/about/versions/oreo

*Build for Billions*. (2020, 01 14). Retrieved from Android Developers: https://developer.android.com/docs/quality-guidelines/build-for-billions

*ConstraintLayout*. (2020, 9 30). Retrieved from Android Developers: https://developer.android.com/reference/androidx/constraintlayout/widget/ConstraintLayout

*Core app quality*. (2020, 08 07). Retrieved from Android Developers: https://developer.android.com/docs/quality-guidelines/core-app-quality

*Fragments*. (2019, 12 27). Retrieved from Android Developer: https://developer.android.com/guide/components/fragments

*Manage your app's memory*. (2020, 10 23). Retrieved from Android Developers: https://developer.android.com/topic/performance/memory

*Optimize for battery life*. (2020, 02 21). Retrieved from Android Developer: https://developer.android.com/topic/performance/power

*ReactiveX/RxJava Wiki*. (2017, 07 21). Retrieved from ReactiveX/RxJava: https://github.com/ReactiveX/RxJava/wiki

*ScrollView*. (2020, 09 30). Retrieved from Android Developers: https://developer.android.com/reference/android/widget/ScrollView