Mobile Application Development Reflection

Jeramy Dichiera

C196 Mobile Application Development

April 21, 2020

Android Software Development

Applications created on the Android platform are built using a Java API framework that provides access to all of the features that are available on a given Android operating system. This framework allows developers to design and create mobile applications with practically unlimited functionality. Applications are able to make use of views, resource management, user alerts and input, and all of the same options and features that native Android system applications have access to and were built with.

Beneath the Java API Framework layer is a series of native C and C++ libraries which handle the Android operating systems system components and services. This layer controls various low level functions of the operating system such as the Android RunTime and Hardware Abstraction Layer. The Java API framework provides abstracted access to these native libraries, and developers also have the option to use the Android NDK to access the platform libraries directly if there is a need to do so.

The Android RunTime and Hardware Abstraction Layer provides a unique and separate area for each application instance to run and controls features such as code compilation and garbage collection. Additionally, applications have the ability to access the hardware of a device directly, allowing applications to bridge the gap from software to hardware and access the camera, attached hardware devices, and bluetooth transmitters and receivers. Both the ART and HAL make use of the Linux kernal that sits at the base of the Android software stack. This kernal controls all of the low level drivers, memory management, and hardware access.

Viewing the entire stack through the lense of a developer creating applications for the Android platform we can see that the bulk of the actual development work happens through the Java API framework, but also potentially touches every other part of the stack. The application is built with the framework, granting it the same capabilities as a native application, and also facilitating communication between the two types of application. When running, the user built application will run in a unique process created on the Android Runtime that depends on the native C and C++ libraries. The application could also control physical hardware through the Hardware Abstraction Layer. And all of these items are running on the underlying Linux kernal.

While planning the application I weighed the different Android operating system versions to answer two questions. I wanted to target a version that allowed the architecture that I planned on using, and I wanted to find an API version that could be run on a large number of user devices and had a wide adoption. Luckily I was unable to find an operating system version requirement for the Room persistence libraries, so there were no hard restrictions there. When creating the application I used information in Android Studio’s Create New Application Wizard to identify API 26 as being well adopted and able to run on a significant number of devices. There were later and earlier versions that I could have selected but I ultimately chose API 26 as both a minimum and target API for a good mixture of adoption and availability since there were no specific restrictions or features that I needed from the API version. I also felt that there was also no need to plan ahead for a new version of the Android operating system so there was no need to target a future API version.

Challenges Faced

While developing the application I faced a number of obstacles. One challenge that appeared while planning my application I discovered that there were two ways that I could persist the data for my app. I could use the Room persistence strategy, or directly access the SQLite database. In my research I found that there were advantages and disadvantages to both strategies.When working directly with the database I would have more direct access, compared to Room persistence where access would be abstracted behind a number of classes.

Another challenge I faced while developing the application was how to handle testing and bug fixes. Should I find and fix bugs as they appeared or should I take a more systematic approach? I have been in environments before where issues were addressed as they were found, but in some cases this led to a spiral of refactoring code as changes were made on the fly. In some cases this works and the bug is cleared, but in other cases fixing a bug could lead to more bugs or even unplanned changes in architecture.

Overcoming Challenges

While designing the architecture for the app I weighed the benefits of accessing the database in a more direct manner, or using the Room persistence strategy where the database was accessed through a series of classes that built upon each other to allow database access and manipulation. My initial thought was to use direct access, but after reading the Android developer documentation I found that this was not a best practice, could be error prone, and I code using this strategy would not generate compile-time messaging. The Room persistence library would require the creation of a few classes, but

I found that a number of the tedious tasks could be performed automatically. For example, defining a database and database schema would be done automatically through annotations in my entity classes, removing the necessity of creating them programmatically. After researching I felt that the Room persistence strategy was the best way to approach my application. It would require a few additional classes, but it was a safer and less fragile solution for what I was trying to accomplish.

To overcome the bug resolution challenge I identified two different workflows. In the first workflow, I would address each bug as it was identified. In the second I would work from a set of milestones and regress the app at different stages. The advantage of the first workflow would be that I could identify and resolve each bug as I found it, with the risk of changing code while it was still immature and potentially introducing new bugs. For the second workflow I would test for bugs at specific milestones and record the bugs to be fixed while working towards the next milestone. After examining both paths I chose to test after specific milestones were reached. I felt that this method introduced the least risk, and allowed me to test at specific points in the development lifecycle. Additionally it reduced the number of updates that I was making to my application at a time and provided a sequence of events that could be iterated over while developing.

What I Would Do Differently

While examining my completed app I found a few areas where I could improve. I noticed that there were a number of basic CRUD operations that all of my view models required. These common operations could be refactored into a parent class that my view models could extend from, reducing the amount of duplicated code. A second area where the application could be improved would be to make a set of helper classes that could collect methods, constants, and enumerations which were used throughout the application and store them in a single location. This would reduce duplicated code and also give a single known source for methods that are used in other classes but technically could be a shared resource if they were not duplicated. A third area for improvement would be to give the application a standard set of design choices that would permeate throughout all of the activities. For example, in each cardview there could be a set padding for all cards, and main elements would appear in the same areas on each page with the same formatting options.

The Use of Emulators

Android Studio provides a way to simulate an Android device using your computer hardware. This emulated device can have almost all of the same functionality as a physical device, and can be controlled in real time using your computer's input devices. Using a simulated device provides the developer with the ability to test on a number of different Android devices and on all of the different Android operating system versions. One drawback for using emulated devices is the fact that they require a specific set of computer hardware capabilities and may not be able to be used on every development machine. Another downfall of emulated devices is the fact that they are a very close approximation to physical devices, but there is a chance that they will behave differently. If the developer were to use a physical device they can have a more accurate feel for how their app will perform and interact on real world hardware. The developer can also perform usability testing that could be more accurate using a physical device rather than a simulated one.