Waterford Institute *of* Technology

Mobile App Development 2

Bachelor of Science (Hons) Applied Computing

Kang – Assignment 2

Ciaran Roche - 20037160

April 16, 2019

Plagiarism Declaration

Unless otherwise stated all the work contained in this report is my own. I also state that I have not submitted this work for any other course of study leading to an academic award.

Contents

1 Introduction **3**

2 Outline **3**

2.1 Kang 3

2.2 Features 3

2.3 Navigation 9

2.4 Layout 12

3 Summary **13**

1 Introduction

This document will outline and discuss my assignment submission for mobile app development 2. Taking a look at the application outline, listing features of the application and explaining their role within the application and highlighting my own observations. A main aspect to the application is the use of the Nav Component, this will be the final feature explored before the document finishes with my own conclusion to Android Development.

2 Outline

2.1 Kang

Vinally an application for record collectors. Modelled to be a cloud-sentric crowdsourced database that allows record collectors to share, store and contribute their records too. While the application lists releases in all genre formats, it is especially aimed toward a specific subset of metal music known as Doom and all the sub-genres which fall under the Doom umbrella.

2.2 Features

This section will list and explore features of the application.

2.2.1 Multiple Models (User, Vinyl, Genre, Rating, Chat)

Multiple models where created in the app to allow for easier transitions between interactions with Firebase and displaying content. Ones worth mentioning are the User and Vinyl model, who both have their own Firestore which allows for the individual models to communicate with Firebase Services. Genre and Ratings are both used in the User and Vinyl Model.
The chat model describes a chat message outline and provides the basis to build in the chat capabilities into the application.

2.2.2 CRUD – View, Update, Delete, Add (Users & Vinyl)

As mentioned above, both User and Vinyl have individual Firestores, within each firestore are functions that allow us to create, read, update and delete items in the Firestore services.

2.2.3 Filter Views

Tabs and swipe support were implemented to allow for easy to use filtering on the data. By implementing different filter patterns forced the development of different view types.

2.2.4 Multple View Types (Recycler, Adapter, Firebase-Adapter, etc.)

One of these view types were an Adapter view. This allowed for the creation of lists within the app. By overriding the Android AdapterView I could customise the lists, allowing for the creation of list views, spinners and a grid view. I had to extend to RecyclerView, again overriding the RecyclerView Adapter, I created a list of genre types, by attaching a listener to each item, I could create a dynamic menu via the adapter.
As mentioned above in order to provide features like swipe and tabs, the RecyclerView needed to be extend via the FragmentStatePagerAdapter, following the same pattern as above, by overriding this adapter I could create a swipe function between fragments.
From exploring these different views and approaches it became apparent the pattern in which the all follow. Each view type had a specific adapter, and each adapter can be overridden to allow for customisation.

2.2.5 AndroidX

Early on in the semester it was decided to build this application using AndroidX. This is a major improvement to the original Android Support Library. As Android Development is quite complex with a lot of dependencies built in under the hood which are abstracted from the developer, the support library grew, with the popularity of Android and with it became quite unmanageable. AndroidX is the solution to this, while providing backwards-compatibility across all Android releases. It also has to be said, AndroidX is extremely new, and to avail of some of the enhancements to the old support library, a lot of the dependencies to this application where in alpha and beta, which was both equally exciting and challenging.

2.2.6 Model View Controller

One of these enhancements included an improved Model, View, Controller pattern. This is where I abstracted functionality within the code, in that the models where separated, the view handled only the render of the view and all or any calculations where handled by a controller. In androidX this controller is known as a viewmode.

2.2.7 Firebase Database

To circle back to AndroidX, the main resources used was Android Jetpack, this is the home to all documentation on AndroidX, and with the aim toward cloud sentric application, Google tend toward Firebase as the defacto in Android Development, with resources such as Android Jetpack all pointing toward Firebase integration. With that, for this assignment I tried to explore as must as firebase as I could, one of these services was Firebase Realtime Database. First the pro's, one of the main benefits was the offline support, in that an app can function as normal and once it reconnects to the internet, it can sync, you get all this for free with out any additional code. It also worked and felt native to the typical Java Model structure, that said it brings me to my main con. In that when you verge away from Firebase adapters, and try to use the Database as a standalone database to your application and roll your own handlers, things get quite complex, as data is returned in a snapshot which requires a lot or parsing and casting my own models to it. This proved tricky but once nailed down it was quite a good developer experience.

2.2.8 Firebase Authentication

Another service I used was Firebase Auth, this in my opinion is the best Firebase service offered, I have experience with other auth services, such as Cognito (AWS), Keycloak (Openshift) and I feel Firebase provide a simple to use service, that abstracts a lot unnecessary configuration from the developer. A simple plug and play API that can be extended to incorporate other sign in options. All salting and hashing of passwords is handled and the developer, in fact unlike other services out there, the application has no access or view of the passwords and is handled by Firebase under the hood.

2.2.9 Firebase Storage

The third service I used was Firebase Storage, in order to be considered cloud sentric, I needed a place to store images and media, so that on sign in, regardless of device the users media could be seen. I found this like all Firebase services pretty native to android development, with a solid API. But that said, there is a notable latency from storage to device. This seemed to be a common complaint online, while the service was easy to use, I would not recommend it based on latency.

2.2.10 Google Sign In

In order to improve a users experience I added Google Sign In and incorporated it into my Firebase Auth. This is quite a proud feature of mine, as do to how I handle users in my app, on creation I use their Firebase Auth ID as a unique ID which I store their user details under within my Firebase

Database. So I had to come up with a solution to handle google users. I found it best to maintain the current flow, so on a successful google sign in, I use the user details to query the database, if the user exists I return the user and launch that app. If no user exists, I fetch all user details from google and create my own user in my Database and launch the app. A simple solution to the problem, but one I am proud of.

2.2.11 Main

As I mentioned previously, I opted to roll my own handlers for Firebase, this stems to each Model having its own firestore. To avoid problems and ensure the app stays synced to the database, a main class was created in that only a single instance of these firestores are created. Pointers to this instance can be then accessed by individual fragments.

2.2.12 Activities

To avoid going into the theory of Activites, I am going to explain how I use them in this application. Trends in Android development seem to lean toward the use of Fragments due to their re-usability, so keeping with this I kept my usage of Activites to a minimum, in that I only used 2 Activites, one to handles the Application start up and the other to handle the Application usage. All views where created with fragments. Each Activity had its own Nav Component, which will be discussed later, the main app activity included a bottom nav, this kind of nav follows the material design guides in that 4 or less views should use bottom nav, any more view should use a drawer layout.

2.2.13 Fragments

Fragments have a similar structure to Activities, a similar lifecycle, but can be reused more easily, bringing not only their interface but also their logic. Fragments also fit nativity into a Nav Component. In total the application is comprised of 15 fragments.

2.2.14 Listeners

Listeners are a common practise within development, waiting on an action to trigger them. A number of custom listeners where needed in this application to add functionality to the application, such as onclick listeners to list items. These followed the same sort of patterns as the adapters mentioned previously. An interface was created, a component then extended this interface in which I override the listener functions to create my own logic.

2.2.15 ImagePicker and other Intents

As mentioned before Android development is quite complex under the hood, there is so many capabilities developers can plug into. One of these capabilities is Intents, it allows us to plug into functionality native to Android, such as calendar, camera, storage etc. So to further explore this avenue, I added an ImagePicker to the app allowing users choose images from device. I also allowed users share vinyl through different mediums available to the device. Such as WhatsApp, Email etc. Quite simple to use, but quite a powerful tool to have within an Android Developers arsenal.

2.2.16 Picasso

So like most developers, and with a plethora of open source software and libraries to use, it is often quicker and more efficient to plug in to one of these libraries over rolling your own. In the case of my app I opted to use a third party library to display images within the App.

2.2.17 Espresso Testing

Testing is a must have in any development today, so I thought it valuable to break away from unit type testing and explore the avenue of UI testing. I done this in the form of espresso testing. With a small learning curve to over come, Espresso has quite a rich API that allow for the testing of UI. The only down side I found was the speed at which these tests run, I found the emulator often could not keep up. So with that, I had to add multiple breaks to timeout the tests to give the emulator time to catch up.
Not long into the development of the tests I began to discover multiple nuances which I over looked in the original development of feature. Reinforcing the importance of correct end to end testing.

2.2.18 GitHub

GitHub was used extensively throughout the development of this application. The real power of GitHub comes through usage that stretch's beyond pushing directly to master. By correct versioning control, multiple branches and a PR structure with some form of CI built in. Correct is a relative term as everyone seems to have an opinion on Git usage. This project allowed me to explore a simple approach of creating a separate branch for each assignment submission and merging back to master of completion. I found this became a burden and harder to manage then smaller more feature focused PR's. I also insured that each commit was built to highlight any potential errors which may be commited to my repo.

2.2.19 Circle CI

As I used Circle CI across my final year project I thought it a good idea to incorporate it into this project. As the flow is the same, there is quite a number of differences in configuring continuous integration into different paradigms. Configuring an Android App would give me another notch in my belt for Circle CI. The checks which my CI covered where a successful build, and a lint check. I spent quite a lot of time trying to incorporate my Espresso testing into my CI to only eventually find out emulators are not supported within the Circle ecosystem. As Circle is a third party hosted CI, they leverage containers for all builds. To overcome this, I wrote a simple script to perform my UI tests locally before committing code to GitHub.

2.2.20 ktlint

As mention above, I performed lint checks within my CI, I done this in the form ktlint. Which extends the regular lint checks we get with gradle, and is focused solely on Kotlin. This insured the code base was in unison and easy to read, with out the added counter-productivity of configuring my own linter.

2.2.21 Kotlin

This whole app was developed in Kotlin over using Java. Kotlin is interoperable with the JVM but allows us to write more concise code avoiding lots of boilerplate that is the norm with Java development. I begun Kotlin development last semester, it was nice to spread my wings and get my hands dirty as I got more comfortable with the syntax this semester.

2.2.22 Helpers

One of the cool features of Kotlin is how we can simply declare functions. I done this in the form of helper files. Similar to a Java Util, except these helper files are just functions, that do not need to be declared or instantiated and can be called by any class throughout the application. This is a really cool feature and helps maintain a Dry codebase.

2.3 Navigation

Navigation is interactions that allow users to move around the content of the app. Android Jetpack's Navigation components is a new implementation of navigation. This implementation ensures a consistent and predictable user experience by adhering to a set of defined rules.

The Navigation component consists of three key parts that are described below:

- **Navigation graph:** An XML resource that contains all navigation-related information in one centralized location. This includes all of the individual content areas within your app, called destinations, as well as the possible paths that a user can take through your app.
- **NavHost:** An empty container that displays destinations from your navigation graph. The Navigation component contains a default NavHost implementation, NavHostFragment, that displays fragment destinations.
- **NavController:** An object that manages app navigation within a NavHost. The NavController orchestrates the swapping of destination content in the NavHost as users move throughout your app. As you navigate through your app, you tell the NavController that you want to navigate either along a specific path in your navigation graph or directly to a specific destination. The NavController then shows the appropriate destination in the NavHost.

For this application two nav components where created, one for each activity. The graph for the startup Activity can be seen in Figure 1, and the graph for the main activity can be seen in Figure 2. I found this a cleaner approach over a single component managing both activities. Overall my experience with the nav component changed my approach to android development as it forces you to think of the routes and paths a user takes through your app while insuring the flow of the app does not break. That said, this was a huge learning curve in itself just to get working and understand it.

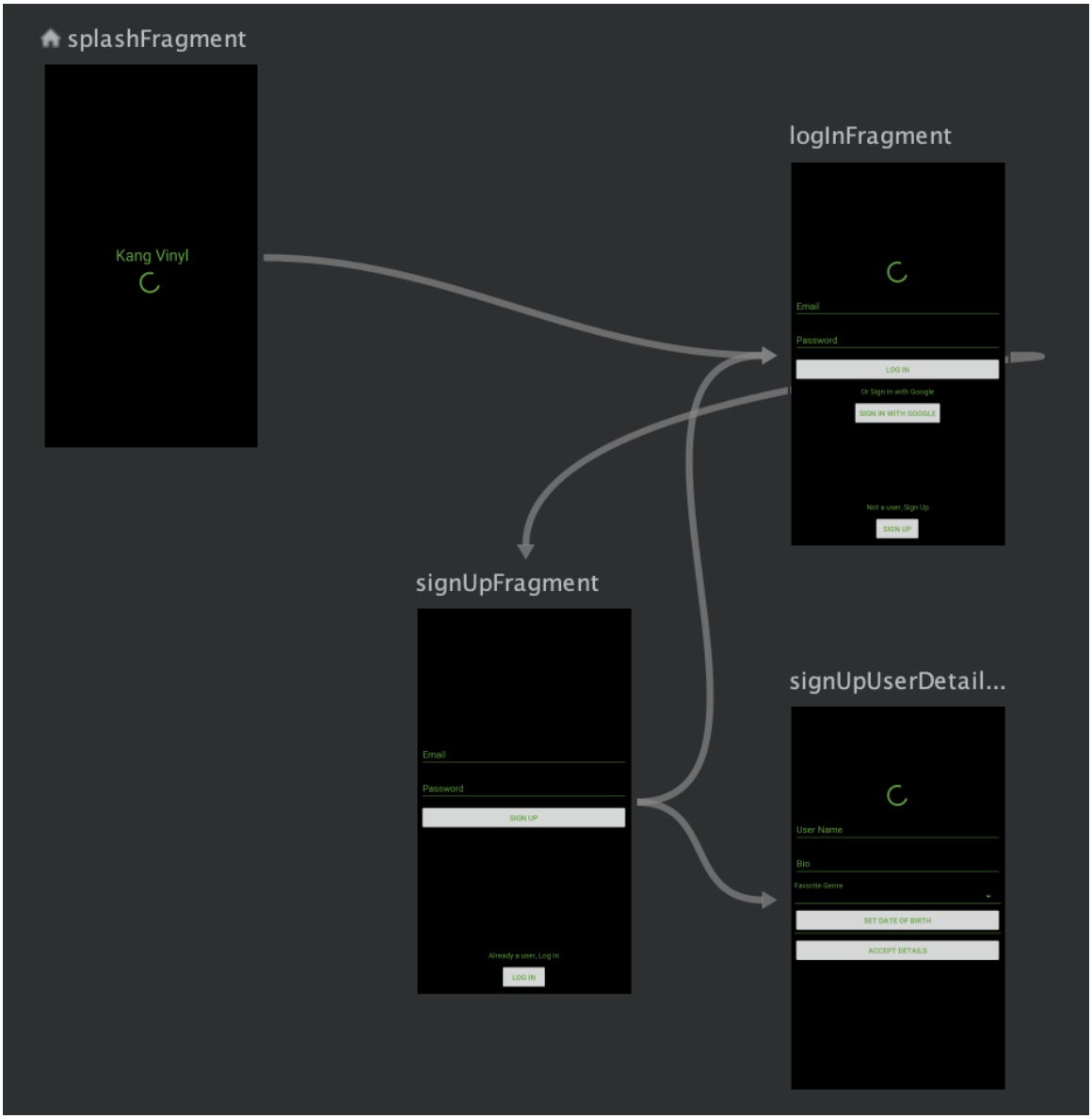

Figure 1: *Start Up NavGraph*

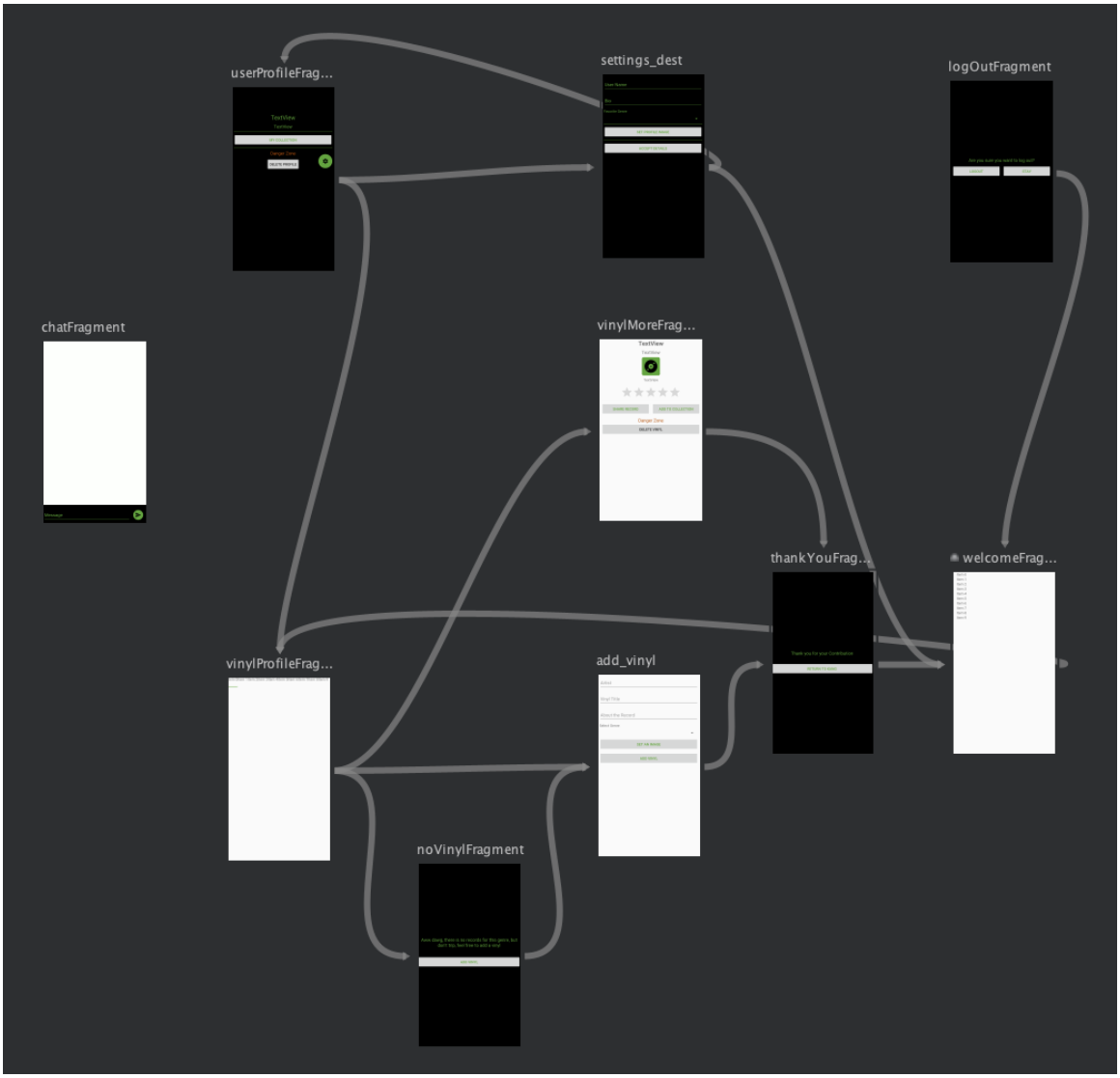

Figure 2: *Main NavGraph*

2.4 Layout

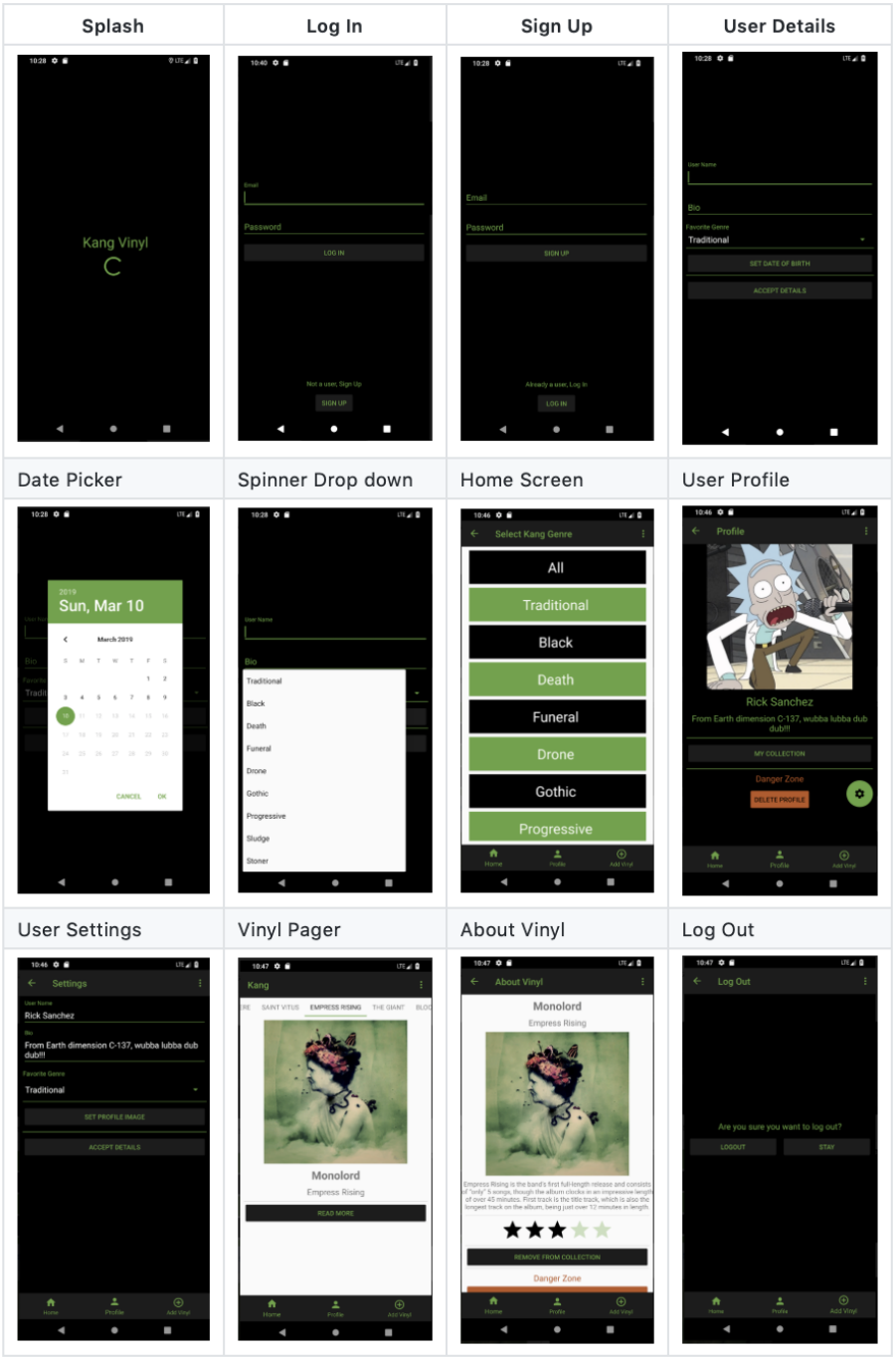

Figure 3: *App Layout*

3 Summary

My main outcome to this assignment was to build confidence and refine my skill as an Android Developer. I feel I successfully achieved this and leave this semester confident in being able to take and complete any android task laid out in front of me.
There was multiple proud moments during the development of this application, one of which I mentioned already in incorporating both google sign in and firebase auth. Another was getting my head around the nav component and dealing with the challenges of AndroidX. All these moments added to building the confidence I strive for in my Android Development.
The exposure to Kotlin was an enjoyable one, it is nice to sit back and compare and contrast differences between paradigms. Seeing how Kotlin leverages Java, abstracting away the boring boilerplate required by Java and allowing my to focus on the task at hand.
Mentioned a few times in this document is the complexity that is abstracted under the hood in Android away from the developer. While giving us the freedom to plug into the powerful underlying architecture and features Android has to offer. I was often left asking myself, how did Android become so popular? Before we had these support libraries, how did developers go about building apps. The complexity of leveraging core components within Android must have been massive. So for the work on the support libraries, and AndroidX, I definitely am grateful as they make developing on Android an enjoyable experience.