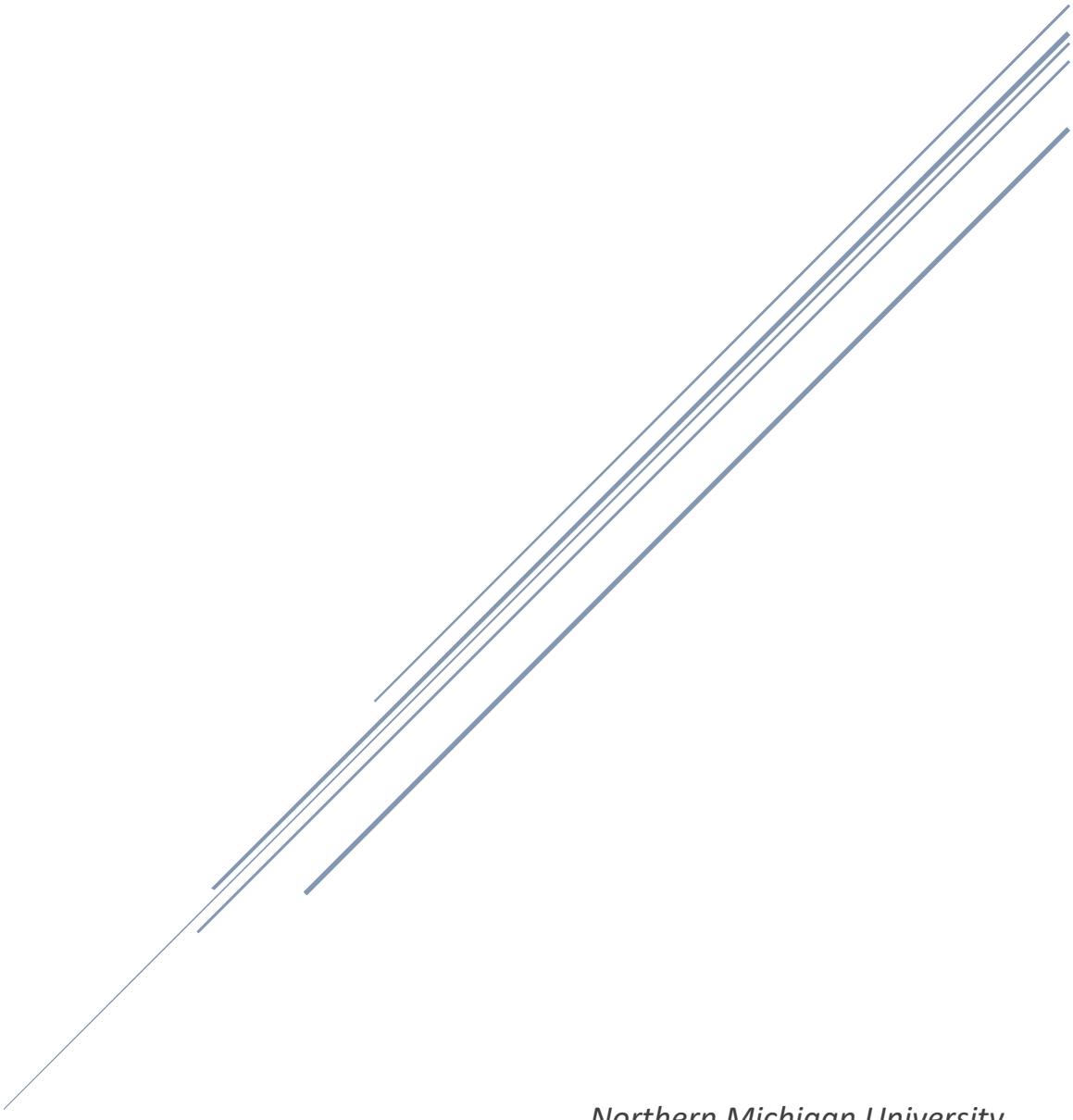
Forager

Tylor J. Hanshaw

CS480

December 1, 2021



Northern Michigan University

Mobile App and Web Design

Introduction

Ever since I was a child, I have loved the outdoors. Hiking, biking, and foraging – I have always known this and it is one of the reasons I moved to Marquette to attend Northern Michigan University. Although I knew all of this already, while I was brainstorming ideas for my senior project, I could not think of anything that "popped" at me. That is until one summer day while hiking the North Country Trail with my girlfriend, I noticed that she always had a foraging book with her to help identify plants near the trail. While she was trying to figure out what kind of fern she was looking at an idea popped in my head – an app that helps one keep track of plants that they have found. I always forget the location of new or interesting plants that I have discovered – such as a small patch of blueberries or maybe Indian Pipes – and this issue only increased while living up here. This is how the idea of Forager™ came to be.

From the start there were a few obstacles I had to work with – one being my language of choice, Kotlin. I had spent some time over the summer of 2021 learning Kotlin and its nuances, but by the time I started my project I was not as comfortable with the language as I hoped that I would be. Another obstacle was the behemoth that is the Android environment, which is almost a language in and of itself. At the time, I was taking CS345 Android Development with Andy Poe, so with that course alongside a handful of books and the internet I pushed on with my nose down. My first goal was to create a schema or wireframe for my project. With this being my first big project, having something concrete and on paper helped me visualize what I saw in my mind's eye more clearly. My second goal was to get a foundation laid out code-wise that I could start building upon. This foundation was my login and register fragments. The metaphorical ball started

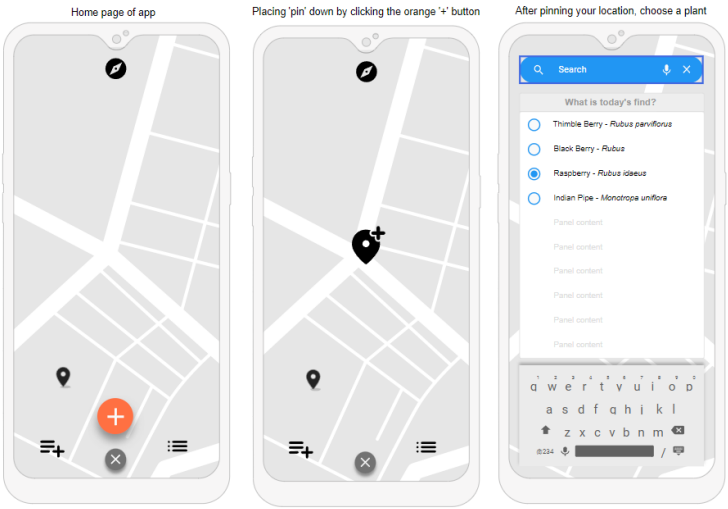


Figure 1: An early wireframe of the Forager app

rolling at this point and one can now see version 1.0 of Forager™, the app that helps one keep track of plants and fungi in the wild.

Architecture

The goal of this project was to not only build a foraging app, but to stick to Android Development best practices as best as I could. I wanted to try to build this app in a way someone or some company would do so in the real world, so from the beginning I chose to follow the MVVM (Model, View, View-Model) architecture pattern, along with the repository pattern and sticking to a single-activity application. With the MVVM architecture pattern the most important concept to keep in mind is the separation of concerns principle - that is keeping classes lean and only containing logic pertaining to the given class. This does a few things for Android development; it leaves an application less prone to lifecycle-related issues (View-Model) and adds the benefit of better readability and easier to test single points of your application.

If one looks at the diagram to the right (see figure 2), one can see what the hierarchy of the MVVM architecture model is based around. There are three main building blocks in this diagram: The View (Activity/Fragment), the View-Model (ViewModel), and the Model (Repository). The View is what the user sees – it is the UI of your application – and in my project my View consists of one activity and multiple fragments. Next is the ViewModel, whose job it is to store run-time (cached) data and manage the lifecycles of the UI controllers. In Forager™ I use only one ViewModel because I am only using one activity through my application. One generally wants one ViewModel per

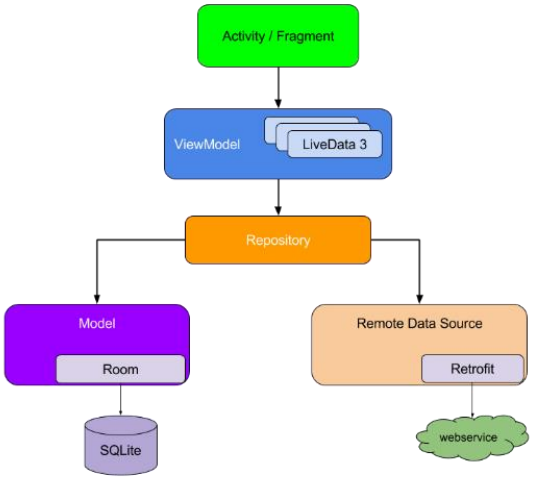


Figure 2: MVVM Architecture Hierarchy

activity, as the ViewModel class is tied to the lifecycle of its corresponding activity. Finally, you have the Model or in my applications case the *DataRepository* singleton. Together, the View, ViewModel, and Model classes form a sort of chain of command that all leads to either local or remote data.

Another important part of my applications architecture is the *repository pattern* that I am following.

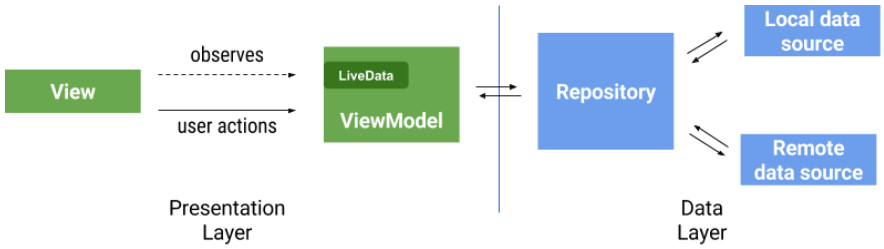


Figure 3: Another look at the MVVM architecture and how interactions are handled through the app

Repositories like the *DataRepository* singleton mainly manage data operations. Acting as a bridge between the application and one or more data sources, and adds an abstraction layer over my applications data sources and the rest of the app. In Forager™ my repository deals with talking to both my local data (MySQL database) and my remote data using Firebase's Realtime Database (NoSQL/JSON tree). In the end my repository only knows one thing, and that is talking to and dispersing non-cached data from my data sources to my ViewModel's.

The code snippet below is an example of getting data from my remote database in my repository. In this example I am using Kotlin coroutines to asynchronously fetch this piece of data for the user. On the second line I am creating an instance of the *UserResponse* data class, which has two member variables, exception of type Exception and user of type User (a custom data class that holds the user's information). With the way I structured this suspend function only one of the member variables of the *UserResponse* will be non-null, and I reflect this down the chain of command into my View where I check which one of these variables is not null, and

```
suspend fun getUserInfo(): UserResponse {
    val userResponse = UserResponse()
    try {
        userResponse.user = userDBRef
            .child(firebaseAuth.currentUser!!.uid)
            .get().await().getValue(User::class.java)
    } catch (ex: Exception) {
        userResponse.exception = ex
    }
    return userResponse
```

Figure 4: Kotlin coroutine function within my DataRepository singleton

deal with that outcome appropriately. When I call *get()* on the specific child node associated with the user I use the syntax; *getValue(User::class.java)*. What this does is after the data is retrieved from that child node the *getValue()* call will map that data to the data class User, and if this fails an exception will be given instead.

On the other side of things, my ViewModel (*HomeViewModel*) has a couple of jobs – it acts as a steppingstone between the View and the Model. Along these lines, the ViewModel also oversees any changes that need to be made to the data after being received from the repository or sent over from the View. So, if the user finds a plant that they want to save, the data gathered in my View will be passed to my ViewModel and converted to an instance of *PlantListNode*. From here it will be passed onto the Model and then added to the user's personal plant list in the remote database. One more important aspect of the ViewModel is that it caches data from the user, this cached data is kept locally until someone else logs in on that device or they remove the app.

Here is another code snippet from my *HomeViewModel* class, this is the corresponding call from the example I gave earlier in my repository. This is a prime example of caching data as the user uses the app. Here, the variable *observeUserInfo* is of type *LiveData<UserResponse>* and the *UserResponse* data class is that custom class I created from earlier. As stated, *UserResponse* will only hold one non-null member variable at a time. When an instance of *HomeViewModel* is created the code in the *liveData(Dispatchers.IO)* block will execute creating a coroutine on a "thread" designated for IO work. Inside that block the *emit(DataRepository.getUserInfo())* call is made which assigns the returned value of the *getUserInfo()* call to the *observeUserInfo* variable.

```
val observeUserInfo = liveData(Dispatchers.IO) {  this: LiveDataScope<UserResponse>
    emit(DataRepository.getUserInfo())
}
```

Figure 5: Kotlin coroutine function from HomeViewModel and an example of LiveData being used to cache user information

I would like to include one last code snippet, this time from my View – this snippet completes the journey of retrieving the user's data from my Realtime Database. To start, *homeVM* is an instance of *HomeViewModel*, what I am doing here is observing any changes made to the *observeUserInfo* variable in my ViewModel. This variable is of type *LiveData* which means that when changes are made to it anything inside of the *observe({ })* block will be executed. I first check to see whether an exception or a user was returned from the coroutine call. If no exceptions were thrown, I set up any UI widgets that are tied to the information that is being observed. And so, the journey is complete – the user's data will be retrieved the first time an instance of *HomeViewModel* is created and when that happens my observer in the View will execute and set up everything needed for this specific fragment.

```
homeVM.observeUserInfo.observe( owner: this, { response ->
    if(response.user != null) {
        currentUser = response.user!!
        numPlantsFound = currentUser.numPlantsFound.toString()
        menuHeaderPlantsFound = findViewById(R.id.user_plants_found)
        val menuHeaderFullName = findViewById<TextView>(R.id.user_full_name)
        val menuHeaderUserName = findViewById<TextView>(R.id.user_username)
        menuHeaderFullName.text = currentUser.fullName
        menuHeaderUserName.text = currentUser.userName
        menuHeaderPlantsFound.text = currentUser.numPlantsFound.toString()
    }
    else Log.e(LOG,  msg: "Error retrieving user's data: ${response.exception}")
})
```

Figure 6: Observing LiveData<UserResponse> to update the UI in my View

Difficulties and Setbacks

The hardest part of this project was learning how to deal with database calls – more specifically, asynchronous calls to my remote database. In hindsight, this seems obvious but going into this project I had never thought of or encountered an instance where fetching data took longer than expected: For example, retrieving user data from my Realtime Database. When I started diving into the database aspect of my project I ran into issues where my UI

would load before any data was retrieved from my database, leading to TextViews with “null” as their text or my app crashing. I was stopped in my tracks and did not know how to deal with this kind of issue. Luckily, I happened to be taking CS444 Parallel and Distributed Processing with Andy Poe which dealt with the exact issue I was running in to. Although Andy’s course was using C++ the ideas translate to any language, and that is when I dove into callback functions and later Kotlin coroutines. I started off using callback functions to solve this issue, but as I kept progressing, I realized that more Android developers are moving towards coroutines and away from callback functions to tackle asynchronous programming. Although both methods are feasible and are sometimes better than the other given certain niche situations, I decided to lean more towards Kotlin coroutines as they were what was being used more in the real world. And keeping to my vision of building an app with current day best practices in mind I felt it was the best move on my end.

I also wanted to add that for the first half of the semester I was splitting my time between this project and my other two classes unevenly. I ended up spending more time on other homework assignments than this one, and because of that I set myself back a good amount. Originally, I had planned to use a textbook I purchased to study for the first three weeks, so I had a good foundation to build off – but I realized that I was spending more time working through the textbook than getting any real work done on my project. So, by midterms I was only finished with my login and register fragments, I switched gears thereafter and this is when I started to see real progress being made. However, because of that setback I could not get everything I wanted done.

Knowledge Gained

With Forager™ completed I feel like I have learned a lot. I have become more familiar with the GoogleMaps API, Firebase Authentication, Firebase’s Realtime Database, MySQL/NoSQL and especially the Kotlin language. When I started this project, I knew almost

nothing about Kotlin or developing an Android app. Now, looking back I feel like I have gained so much more than I expected I would.

Before starting CS480 I also was not great at managing my time and resources or producing a set timeline, for bigger projects and smaller ones. However, sitting here writing this I now feel more confident when it comes to managing my time and resources. Sure, I have had final projects for classes before this, but this project forced me to really think about what I wanted and what steps I needed to take to get there. In the end, this course feels like it is the closest one can get to creating an environment that a computer programmer would be in on the job, and I cannot express how happy I am looking back on the progress that I have made.

Improvements

There are a few things I would have liked to do differently with my project, one of the biggest ones is using Jetpack Compose for my UI. For most Android apps, the UI is built up of a mix of XML, ranging from layouts, menus, drawable's, and even animations. But Jetpack Compose builds its UI entirely programmatically, no XML in sight. A little over halfway through the semester is when I discovered Jetpack Compose and I almost made the switch, which would have meant throwing all my current layouts out and starting fresh again. However, I decided against it. A few other improvements would have to be the UI design, I find it tough sometimes to work on the UI because in my eyes it is never good enough. With this mentality it really slows down any progress I make, I would create a layout for my login fragment and then end up scrapping the entire layout because it would look 'off' to me.

Lastly, there is one feature that I could not include due to time constraints, and it is something I plan to add within the next month or so. I called the feature *Groups*, and it was a way for you to create small groups of friends and share the locations of your plants with them. You would be able to see the plants that they have found on your map and toggle them on or off depending on your needs. There is a lot more I want to do with the *Groups* feature, and to

my app in general. But in the end, it came down to either polishing my UI and creating my documentation page or creating this feature – and I went with the former.

Citations

1. Alcérreca, J. (2018, August 23). *ViewModels and LiveData: Patterns + antipatterns*. Medium. Retrieved December 1, 2021, from https://medium.com/androiddevelopers/viewmodels-and-livedata-patterns-antipatterns-21efaef74a54.
    a. Figure #2.
2. Google. (n.d.). Guide to App Architecture. Retrieved December 1, 2021 from https://developer.android.com/jetpack/guide?gclid=Cj0KCQiA-qGNBhD3ARIsAO_o7yl1FmfrgOl0o4a5dblnMPH4DQuk8m3tWXXFgK1U49zLbILmMMl1ETYaAtyUEALw_wcB&gclsrc=aw.ds#overview
    a. Figure #3.