Group Report for YSC3232 Software Engineering Final Project

SnapCat

Haroun, Hebe, Bryan

Yale-NUS College

Source: github.com/tanyhb1/YSC3237FinalProject

In this group report for our YSC3232 Software Engineering final project, SnapCat, we explore the challenges, technical or otherwise, that we faced as a group and how we solved them.

SnapCat is an Android application that allows users to take photos of cats that they encounter and upload them to a platform where everyone else can enjoy the photos. It provides geo-tagging and provides a sleek user interface for viewing cat stories that users upload. We implemented SnapCat in Kotlin, which was a language that none of us had prior experience with. This report documents the challenges that we faced along the way, and we hope that it provides some insight into our experience.

Challenges

Server-Side Issues

Getting the Flask server to work proved the biggest obstacle in finishing the app on time. It only took about a week and a half to finish coding the basic functionality to pass testing. Unfortunately, that testing was only on localhost. This proved insufficient. When we tried to link the server to the app, we realized it was not accessible to the outside world. Several different attempts were made to rectify this; general debugging, moving the server from a laptop to a Digital Ocean droplet, changing the droplet permissions, even rewriting the server in NodeJS. Literally none of it worked. We're still not entirely sure why the server was so stubbornly unaccessible. This was finally resolved when we covered servers and pythonanywhere on the last day of class. The server was up and running on pythonanywhere by the end of the day, and it was generally smooth sailing from there. This bug was particularly challenging because it was a DevOps issue instead of a programming one, which

we have no experience with. It also caused some of the other parts of the app to be held up, although for the most part this was resolved by using dummy data. This caused other issues.

Inconsistent Interfaces

A key part of our application is the list of CatData, which is pulled from the server, used by the Map, CatsFeed and DetailView fragments, and updated every time the user takes a photo. All these different interfaces need to interact with the same format of CatData. Unfortunately, in the initial design, Map and CatsFeed/DetailView had different expectations of CatData. This was the result of slow development of the server mixed with poor planning: since we didn't have actual server data, we used dummy data. As people worked on different sections in parallel, dummy data and format wasn't standardized. For example, all the image-displaying views expected to be provided with a dummy image from the *res* folder, stored locally, but the server provided bitmaps that were held as in-app variables. Resolving this incompatibility was painful. Further, the dissonance was only realized once the server came online. Since that was late, it meant that a lot more had already been built on top of the different CatData interfaces, so refactoring was an even bigger job. We resolved this through a night of focused work.

Kotlin

Kotlin is a cool language to learn for many reasons, such as its conciseness, its integratibility with existing Java libraries, and the fact that is officially supported by Google for Android development. However, it is a pain to code in Kotlin, not because its syntax is more complex in Java or that is is hard to learn. In fact, knowing Java, it is easy to transition to Kotlin, and Kotlin has syntactic sugar that makes it as easy to write as Python. However, the issue is the lack of the resources about Kotlin; most of the Google searches redirected us to Java resources, and we resorted to relying on Google Developers documentation and on android's debugging tools, that are not very intuitive. As such, implementing a feature such as using Google Maps API was pretty straightforward and took a few lines of code. However, debugging it was much more time-consuming and less exciting.

Javadoc (or, Kotlindoc)

We came to the solemn realization that AndroidStudio only supports generating Javadoc for code written in Java. Since we wrote a majority of our code in Kotlin, this was problematic. Furthermore, the lack of documentation available online for tasks such as this compounded the issue. Thankfully, we managed to find a plugin, *Dokka*, that parsed our

Kotlindoc and generated something very similar to Javadoc, which is what we present as our final documentation.

Permissions

We had to request permissions from the user (Android mandated!) to access several components of their phone - *writing* to storage, *reading* from storage, and taking photographs. Permissions were sufficiently difficult to manage because there are numerous ways of dealing with permissions in Android. Firstly, a user would have to give permission upon installation of the app. To do this, we had to manually implement functions in Kotlin, to request for the permissions from the user. This initially caused us some trouble, because we were unaware that we had to work with permissions beyond the Android manifest, and so we could not debug the issue. Once the user gave us their permission upon installation of the app, opening the camera would trigger another bunch of requirements for permission, and again this had to be dealt with via manual requests written as functions.

Members working during different timelines

We were all motivated to contribute to SnapCat and make it the App all cat-lovers dream of. However, we were not available at the same periods throughout the project timeline; while some of us were more available during the start of the project, others were only available at the end of the project. This did not impact each member's contribution to the development of the App. However, it caused some dependency issues. For instance, when a member needed cat photo details to include on the Catsfeed and map fragments, the server had not been set up by another member to communicate with the app yet. As such, the former member used dummy data, which was inconsistent with the data the way it was stored on the server. Luckily, this was not a blocking issue and we were able to fix it.

Conclusion

The app looks awesome. The interface is sleek, the server connection is consistent and active, the pictures download fast, the map icons are cute, taking photos is pretty intuitive; generally, everything works. The only part we didn't finish was creating an editable profile - but honestly, that's not a key part of the app. We're planning to remove that page and put the app on the PlayStore.