KanaKards

by Ryan Moore

Table of Contents

Introduction..3

Technologies..4

Organization 5

Difficulties 6

What would I do differently?...9

What did I learn? 10

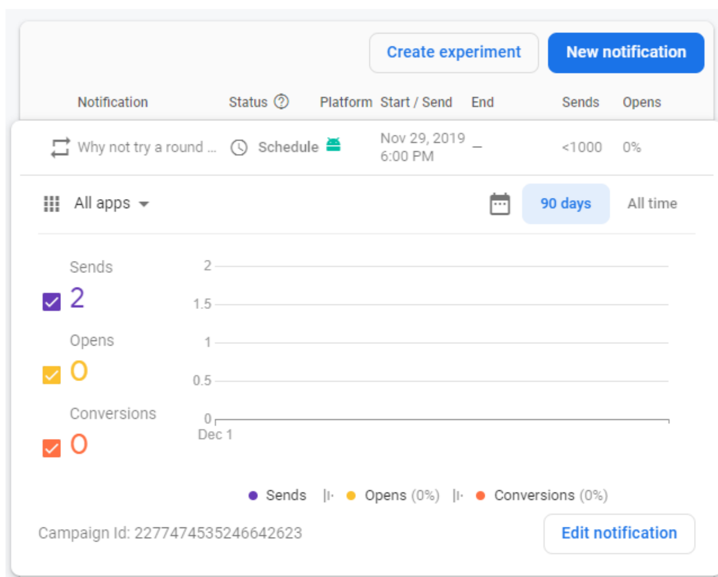
Introduction

My name is Ryan Moore, and the project I chose for CS 480 was an Android app that I call KanaKards. Before I took a trip to Japan with my wife in the Summer of 2019, I started frantically trying to learn as much of the Japanese language as possible. The biggest roadblock I hit in the beginning was with learning the two phonetic character systems (Hiragana/Katakana). In order to make my life easier, I quickly threw together a flashcard app to use in my spare time alongside traditional studying. While I used it, I'd think up additional features I wished I had added. Before long, I had a feature list that was more than enough for my Senior project.

The final rendition of the app offers a multitude of ways to supplement Japanese language learning. Like the original app I made back in the Summer, it offers the traditional flashcard mode. Beyond that, it offers audio pronunciations, voice input, character tracing, and even the ability to load in your own custom character sets. Almost every feature I had previously planned for made it into the app. The only one that had to get cut due to scope and time constraints was the stroke order guide/check in the tracing mode. Even with a feature cut, I'm very happy with what the final version ended up containing.

Technologies

The entire project was coded and designed in the IDE Android Studio. As support for GitHub is included natively, it was chosen for version control. Other than XML, the whole project was done in Java and Kotlin. Java was chosen for the majority of the coding as it's the de facto language of Android and objectively had more support in terms of documentation than Kotlin. Kotlin was chosen for the main activity due it being less type-heavy and having the exact same library support as Java. For Push Notification support, Google Firebase was used. The Firebase console allows for recurring notifications to be sent out under a variety of custom conditions. There isn't much choice when it comes to notification services, and Google's Firebase integrates incredibly easily into Android apps, so it was a clear choice.



Organization

Of the many roadblocks faced during the development of my app, none came up nearly as often as my organization practices falling apart. Throughout the whole process, I gained an increased familiarity with Android development, and the way I coded changed in response. While it was exciting to feel more confident, my coding practices began to lose any sense of consistency. It got so bad at one point that I scrapped the entire original project and re-wrote it. In response to the lack of consistency, I would often go back refactor older code. While this worked in my favor going forward, it only made the differences more glaring. If my code were a house, the majority of the building is structurally sound, but there's a few loadbearing beams that could do with some maintaining.

Difficulties

Many difficulties popped up during the course of development, and I would be hard-pressed to call any one of them the worst. Early on, I discovered that while Google Text to Speech is now native to Android, it refuses to pronounce “one letter” sounds. If I wanted it to speak the pronunciation of あ for example (short “a” sound), nothing would happen. In response, I wrote a small class called SpeechCorrect that would check what character was about to be spoken and make corrections to the String so that it would make the appropriate sound. Much later, I discovered that an easier solution would be to tack the “ー” character onto the end before it gets sent to the Text to Speech service. That only took one line.

While I should have expected it, I encountered something very similar with the opposite function, Speech to Text. One letter/character sounds are not accepted. Instead, it acts like it didn’t understand you correctly. After countless hours of looking for solutions, I ended up changing the trajectory of that feature a bit. While still providing the same core feature of Voice Recognition, I opted to have the user attempt to pronounce short words that incorporated all of the characters. This probably worked out for the best as it now has the user practicing pronunciation and stringing characters together.

When approaching the drawing/tracing feature, I heavily underestimated how much work it was going to take. It already started out fairly poorly when I discovered that free-form drawing is not natively supported. I instead would be creating my own custom View. I figured

that once I got that working, I would be taking advantage of a library to check the similarity between what was drawn and what was supposed to be drawn. It would turn out that such a library did not exist, or had lost support long ago. I wasted far too much time looking for solutions, and settled on quickly making my own.

It takes the character and the user attempt in the form of bitmaps and first makes sure they both have the same white background.

```
private Bitmap makeMonochrome(Bitmap bitmap){
    Bitmap newBitmap = Bitmap.createBitmap(bitmap.getWidth(), bitmap.getHeight(), bitmap.getConfig());
    Canvas canvas = new Canvas(newBitmap);
    canvas.drawColor(Color.WHITE);
    canvas.drawBitmap(bitmap, left: 0, top: 0, paint: null);
    return newBitmap;
}
```

Then, to make processing quicker without any substantial accuracy lost, it “compresses” them by a factor of 4.

```
private Bitmap compress(Bitmap bitmap){
    return ThumbnailUtils.extractThumbnail(bitmap, width: bitmap.getWidth()/4, height: bitmap.getHeight()/4);
}
```

Finally, it just runs through each pixel and checks if the colors match. For each match, a counter gets ticked. In the end, it's divided by the total number of pixels to get a percentage of similarity.

```
private double compareBitmaps(Bitmap bitmap1, Bitmap bitmap2){
    int pixel1, pixel2, r1, r2, g1, g2, b1, b2;
    int match = 0;
    int width = bitmap2.getWidth();
    int height = bitmap2.getHeight();

    for(int x = 0; x < width; x++){
        for(int y = 0; y < height; y++){
            pixel2 = bitmap2.getPixel(x, y);
            r2 = Color.red(pixel2);
            g2 = Color.green(pixel2);
            b2 = Color.blue(pixel2);
            pixel1 = bitmap1.getPixel(x, y);
            r1 = Color.red(pixel1);
            g1 = Color.green(pixel1);
            b1 = Color.blue(pixel1);
            if(r2 == r1 && g2 == g1 && b2 == b1)
                match++;
        }
    }
    double matchD = match;
    double volume = width*height;
    return matchD/volume;
}
```

What would I do differently?

If I were to do this over, I would definitely start by going further in depth on researching each part of the project beforehand. A lot of the time, I thought I had a better understanding of something that I had, in reality, only scratched the surface of. This would enable me to plan ahead with accuracy and not have to constantly make on-the-fly decisions.

For the next app I inevitably work on, I will need to plan around device compatibility from the very beginning. While I fine-tuned every graphical component on my own phone, certain portions of the app were flat-out unusable on other devices. This ends up only being a minor quirk to keep in mind if done from the beginning, but can quickly get out of hand if ignored all together.

Finally, keeping up to date on unit tests will have to be a must. Bug testing devolved into a lovecraftian nightmare toward the end.

What did I learn?

With this project being the largest program I have ever worked on alone, the experience was invaluable. Throughout the whole process, I found myself excited to put time aside to work on it. I was able to develop I much deeper understanding of Android as a whole, Kotlin, and even Java. Every mistake or roadblock reinforced the importance of proper planning, while every effortless implementation was pure positive reinforcement.

While some areas of the project were much simpler than expected, the vast majority ended up being far more complicated than I had originally planned for. At some points, it felt like I had gone beyond the intended scope of the project in terms of work put into it. However, each time it motivated me to keep looking for other solutions and evaluate every available option. It turns out, there are some people on StackOverflow who have no idea what they're talking about.

In terms of specifics, there's an objectively small bit of info I picked up along the way that had an unexpectedly strong impact on me. I was up late trying to get SQLite to cooperate when I had enough. I ripped every trace of it out of the project and scoured the internet for any other possible option. What I discovered just about broke my sleep deprived brain. Unique to Android exists an object called SharedPreferences. SharedPreferences points to a file with pairs of keys and primitive type (plus String) values and includes methods for reading and writing. It was here the whole time, and it suited every bit of my persistent data needs without dealing

with databases. It’s a small thing, but over the course of this large project, the small things made all of the difference.