Report – Mobile development theory

Description of the application

The application I decided to build was the one described in the assignment, however, I decided to add a bit to it for the purpose of learning more about Android. Basically, the application pulls information from the GPS sensor to figure out where you're located, then it uses the Geocoder class to figure out the name of your location (e.g. Gjøvik or London.) Lastly, it uses your GPS coordinates to get temperature information from a weather API. I made a wrapper class for the weather API that handles HTTP and JSON parsing. The class exposes a method that conveniently returns an object with weather information. For HTTP requests, the Android Async HTTP library was used to reduce the amount of boiler plate code, and to make the code slimmer and tidier.

Once the location coordinates, the place name and the weather information has been extracted, the app will present it to the user in the form of plain text when the user clicks on a button. The user can then specify if he/she wants to store place and temperature information in a list. The list will be stored in a database for later retrieval. All the user facing strings have been translated into Norwegian, so the app is available in two languages, namely English and Norwegian. The app will use Fahrenheit instead of Celsius and feet instead of meters depending on the locale the device is set to.

The app has the following external dependencies:

- Realm
- Realm adapters
- Android Async HTTP

Data persistence

When the need for persistent storage arose, I had the choice between SQLite and Android preferences. Android preferences would not suffice for my needs, so I decided to go with SQLite. The default way of creating and interfacing with an SQLite database is to use the SQLiteOpenHelper class provided by the Android SDK. This class works, but it's quite

verbose. You need to write a lot of code just to get started, and I find that to be a bit tedious. This made me search for alternatives that abstract away all the boilerplate code that I'd prefer not to write. The first library I found was Sugar ORM. It was quite easy to install and implement into my project. You simply make all your models extend the SugarRecord class, and that's essentially all you need to do. The library will take care of the rest. This sounded great, but unfortunately, I couldn't get it to work. The first problem that arose was that I got a "over 64K methods" error. I enabled multidex support to get around this issue, which did work, but after that I got other problems. I'd get tonnes of exceptions from the library related to the table creation and so on, and I couldn't figure out why they occurred.

After a while I just gave up and decided that I had wasted enough time. This led me to search for something else. I came across ActiveAndroid. Like Sugar ORM, ActiveAndroid seemed promising. I tried to incorporate it into my project, but once again I had issues. I couldn't get the library to work, and I gave up relatively quickly when I realized that it had not been properly maintained for years. I continued my search for a simple and concise ORM for SQLite, when I finally stumbled upon Realm.

Realm isn't an ORM for SQLite. It operates differently, using its own special file format to store information. The website quickly convinced me to try it with statements like "Thanks to its zero-copy design, Realm is much faster than an ORM, and often faster than raw SQLite." and "Trusted by Fortune 500 mainstays, Global 2000 leaders, and #1-ranked app store successes, Realm is built into apps used by hundreds of millions of people every day." I installed it and incorporated it into my project, and I was blown away by how simple it was. I basically only had to extend the RealmObject class, like with Sugar ORM, but Realm was different in that all Realm objects are reactive. This is great for list view adapters. I simply save my new entry in a callback somewhere, and the realm adapter will automatically update the list view to contain my newly created entry. I don't have to add the newly saved object to the adapter, nor do I have to manually retrieve it from the database after saving it. I simply have to save it, and the Realm adapter for the list view will do the rest. It made my code shorter, easier to reason about, and more declarative. So not only is it really easy to use, but it also allegedly beats SQLite performance-wise in some cases.

Location

The app I was building needed precise location from the GPS sensor of the device. I was already using the GoogleMap class, and figured I could get all the location data I needed from it, but it turned out to be more difficult than that. The class had a method called #getMyLocation(), which would be suitable for my needs, but unfortunately, this method had been deprecated. When I looked up for an alternative to it I was amazed to see that there was no simple alternative. The recommended alternative is to use the fused location provider. This requires that your activity implements three interfaces and registers multiple callbacks. I did all of this and I got it to work, in the emulator. When I decided to test it on an actual phone it didn't work (Note: Up until this point I had not tested the app on a physical Android device because I don't own one. I had to borrow one.) The #onLocationChanged() callback was never called. I couldn't figure out why, so after I while I decided to test it in the emulator again. That didn't work either for some reason. The next few hours I spent trying to fiddle with the code to get it working. After a long period of fiddling with the code I finally managed to get it working, and I'm not even sure about what I did wrong in the first place. For all I know the app might only work on versions of Android prior to version 6.x. I think the updated permission system gave me troubles but unfortunately I didn't have time to investigate it further.

Native apps vs web apps

As smartphones became increasingly popular, the need for responsive websites became ever more important. A response website is a website that scales appropriately according to the size of the display of the device upon which the website is viewed. This is the core principle behind the "mobile first" philosophy. One should initially design a website with mobile devices in mind before moving onto making a design for desktops. This is important because a very large number of people nowadays use their mobile devices to browse websites. If a website doesn't display properly on a smartphone or a tablet it will lead to user frustration and dissatisfaction which, ultimately, might lead to users not using your website at all.

Because of this development, web developers have increasingly become better at creating web applications for mobile devices. With this in mind, one might wonder whether it's at all necessary to create a native app if you already have a response website that works just fine on mobile devices. What exactly are the benefits of using building a native app vs a web app?

There are in fact quite a few advantages to developing a native app, but also a couple of disadvantages. The main advantages will be performance, more control and native UI widgets.

A native app will always feel better than a web app because it's built specifically for the device in question. Web apps tend to be general and they are rarely aimed at just one platform. A well made native app should be designed in accordance with the UX guidelines of the device in question. This makes the app intuitive for the user because the user already knows certain design patterns that are specific to that device. A web app that targets both Android and iOS will have to be a compromise of general UI and UX ideas to make sure that all users will know how to deal with it.

Performance is a big factor. Native apps are far more performant than any web app because they can access the underlying APIs of the device directly as opposed to using webkit. This makes the app feel more responsive to interaction, which translates to a better user experience. A heavy app like Instagram would most certainly deliver a subpar experience if it were a mobile web app, and this is attributed to the limitations of web technologies. There are, however, some advantages to web technologies.

A web app will be device agnostic, so it will take less time and resources to develop because you won't have to be programming multiple apps for different platforms. It also doesn't require the developer to know multiple programming languages (e.g. Java for Android and Objective-C for iOS), all you need to know is JavaScript. Another advantage is the simplicity. Making a web app is generally easier than making a native app since you're just using typical web technologies that have a relatively approachable learning curve compared to native SDKs.

The app I decided to build is a native app, but it would be perfectly possible to make a web app that does the same thing. I do believe, however, that a native app would be far more desirable in this case because the app displays a big map view on the screen, and this undoubtedly performs better on a native app than on a web app.

Extension

The app could be extended by adding more elaborate weather information and possibly other information related the location of the user. The code was designed to be flexible and extensible. The WeatherRestClient.Forecast class for instance contains fields that are not used in the current application, but they could be used later if someone decided to extend the app to show more detailed weather information. The app was also built with localization in mind, so there are no hardcoded user-facing strings. Support for multiple languages could be added easily.

Individual assessment

Overall this has been a fun assignment and I learned quite a bit in the process, however, I do believe that the workload was a bit overwhelming given the short amount of time we were given. I can only imagine what it must've been like for the students who didn't know Java before the course.