[GSoC'21]Irdest Android Client-Blog III

Ayush Shrivastava | Katharina Fey

Thursday 19th August 2021

Prelude

Hell yeah, we paved our way to the conclusion of summer of code while working on this magnificent piece of software, *[Irdest]* and I'm super excited that you too are here! It is super happy to see if you've been following the series of the trailing blogs where I shared the progress the project made with the time of the course of the summer and the proposed timeline. Okay, so in the final phase of summer of code I focused on implementing the features supported in the upstream by Irdest(in the Rust library) in the android application, along with implementing better CI(will touch upon it later) and revisiting how we used our pipelines, Jobs and our custom docker image for CI, also, easing the cross-compilation for developers and modernizing the application codebase via using the *best* practices, although we faced blockers due to internal changes in NDK v23 and could not go ahead with *all* the changes, yeah quite sad :(

Okay, so now let's see in detail and discuss the work done on the each component and brief thought process behind decisions made. This document first contains the work in final phase of summer of code

I. Implementation of Features Supported by Irdest

This was the crux of the project and quite a *tricky* and technical task to implement, all the work done on fixing and rewriting the FFI layer, whether it be from android application side or the `android-support` crate from the rust library, in previous phase comes into action here. Considering the time available to us and keeping in mind about not being overwhelmed or too excited to write a bunch of core-library functions, we decided to implement the basic functionality of user Registration and Login in the Application, and manually test these functionalities work fine and wrote a CI for them as well, to not let regressions creep in our codebase again. Yeah, so for implementing the Registration feature in the application, I fixed the FFI layer(again :P) and correctly set the wrap/unwrap functions in the rust side of FFI layer, fixing package name along with mentioned tweaks resulted in correct functioning of the Registration feature. So you can now create a new user and get a cryptographic ID assigned to it and use the credentials to login to the application. Making similar changes in the Login function of library, fixed stuff. With these library functions being fixed, the Auth began to function, *theoretically.* I had to change and fix the UI/Navigation setup in the application, how screens are changed/exchanged etc, in order to make Auth work, from point of view of an end-user.

II. Redefining & Re-architecting the App Navigation

Previously, the Register screen wasn't being displayed properly, it was nested or better word to use, split the screen in two parts, Login one and Registration one(see*[we/irdest/#21]* for more context and a clear picture). The problem turned out to be how Fragment transactions in the application were being handled and how we exchanged layout files *on-the-fly* along with the aforementioned Transactions. So previously everything was handled inside a single root file only, the layout(of login screen) was already present there by default and on going to registration did *not* entirely remove the Login layout instead split the screen, and some hardcoded dimensions too were present, making the problem persist more and *less* easy to fix . So what I did was creating an abstraction in the root layout file and keeping that abstract layout empty by default, with proper dimensions, which made sure the entire space is occupied by the concerned screen. So that root layout in the main file was essentially a `FrameLayout` which spanned screen accordingly and that `FrameLayout` held exactly which layout is going to be displayed on the screen. So you can consider this `FrameLayout` as a container which showed layouts as per requirement and initially contains nothing. Yep, you never get to see an empty screen, which is because we dynamically set the layout to be displayed in the `FrameLayout` via Kotlin files in the order the screens are supposed to appear.
Well that's enough discussion on the topic. I made all the changes discussed in the previous two sections in a single MR :P , so here it goes *[we/irdest/!38]*

III. Revisiting the Project CI

Okay, so by then we had our Rust library being compiled in our CI pipelines, but we wanted more than that, the usability of the components/artifacts that were being produced as a result of builds. So we decided to publish the rust library to GitLab CI directly from the pipelines and use those artifacts as per need. Also, we used to publish the APK but since no cross-compilation was taking place in the CI, hence the APK being published from there was pretty much useless, so we enabled the cross-compilation in the CI and continued the APK uploading, as a result of which, the application installed using the APK from CI pipelines was running properly on the device. Next were some productivity related changes made in the CI, e.g., by design the APK obtained from application build is stored deep down in the `app/build/.../.../debug/app-debug.apk` and was being uploaded to same path in the artifacts archive from GitLab CI, I removed this Matryoshka dolls style hierarchy and moved the needed build files/reports to the top level directory.
You can find the corresponding MRs below:

- Enabling the Cross-compilation: *[we/irdest/!34]*
- Uploading Rust Library as CI artifact: *[we/irdest/!35]*
- Removing Matryoshka dolls style artifacts archive hierarchy: *[we/irdest/!40]*
- Uploading Lint Reports on Failure: *[we/irdest/!42]*

IV. Modernizing the Application Codebase

In the final days of the summer of code, we took active and fast steps to migrate chunks of our application codebase to follow *Modern Android Development* practices. Although, due to some NDK version incompatibility with the cross-compiler plugin we were unable to merge these changes and unable to fix our docker image too. But anyways, since the CI was green previously with optimized build time and our exhaustive docker image, so it'll have to work again this time too! Okay, so coming back to the topic, the first MR I created in this direction was the migration from Groovy Gradle files to Kotlin DSLs, this migration already as numerous and obvious benefits over conventional Groovy Gradle files, but the cherry on the top was that with these commits in the MR, the cross-compilation was automatically being triggered on hitting the build button/icon only! Previously we had to compile the library first and then the application, to link the library to the application, but this MR saved us a huge time and PITA :)
The next step was regarding improvement of application performance via reducing Memory consumption while it is running. To achieve it we first eradicated all the `findViewById()` calls and the *not* so recommended Kotlin Synthetics as well, you can learn about the reason for the change in the linked issue(s). We instead used ViewBinding to bind and reference the views in runtime without worrying about the application crashes, this was a huge asset and since no view scans were being performed in the application runtime, the application runtime speed also increased and resulted in a decrease of memory consumption. But sadly, we couldn't go ahead with the merging of these MRs because of the mentioned NDK version and cross-compilation plugin incompatibilities : (
We'll be able to merge these as soon as we fix the docker image. Although, there is a way to fix it but that is not *elegant*, also, we want to do it for once and all, like no need to touch that CI file again unless we *have* to introduce some entirely *new Job*. Find the corresponding linked MRs here:

- Migration from Groovy Files to Kotlin DSLs: *[we/irdest/!36]*
- Using ViewBinding & Remove *Slow* stuff: *[we/irdest/!41]*

And, the issue:

- Using ViewBinding instead old methods: *[we/irdest/#22]*

Cheers, until next time we meet and hope to see 'ya in the final report!
∼Ayush Shrivastava /