GSOC'23 Final Report - The Mifos Initiative | Mifos Mobile 6.0

Contents

Pre GSoC

- I was profoundly engaged in the evolution of the Mifos Mobile project. Even before my selection, my commitment led me to diligently address the issues documented on the GitHub repository, meticulously rectifying existing bugs within the project's framework. Moreover, a significant milestone was reached with the successful integration of Biometric Authentication into our project. Beyond this, I extended my support to a cohort of aspiring contributors, guiding them through their inaugural steps into the realm of open-source collaboration.

- My tenure of dedicated contributions granted me a substantial head start in familiarizing myself with the intricacies of the Mifos Mobile project. The resonance of elation was palpable when the idea list was unveiled. While the tasks presented novel challenges, my confidence remained steadfast, bolstered by

the awareness that, with the right guidance, I could adeptly navigate through uncharted territories and execute these tasks effectively.

Work Accomplished

Butterknife to ViewBinding

- Mifos mobile project was heavily dependent on Butterknife hence we needed a better alternative to streamline the process of binding xml layout views to our kotlin code. The alternatives infront of us were Viewbinding and Databinding. My co-contributor and I chose the former for a lot of different reasons but the primary one was that we knew. down the line we had to migrate our project from xml to a compose view. Hence we didn't want go into the hassle of implementing Databinding. Besides `kotlin-android-extensions` is deprecated anyway so, it was always better to use view binding. We removed butterknife from all of the activities/fragments/adapters except the `PasscodeActivity` since the xml for this wasn't included in the project path of mifos mobile and so generating the Binding object for its xml wasn't possible. Simply put, we are using `Mifos PassCode View` dependency in our gradle making us unable to implement viewbinding here.I have implemented them in the below PRs:

  1. PR #2160: Migrated accounts overview to viewbinding
  2. PR #2162: migrated beneficiary adapters to viewbinding
  3. PR #2156: unreachable beneficiary fragments migrated to view binding
  4. PR #2159: migrated guarantor related fragments to viewbinding
  5. PR #2144: Migrated activities to viewbinding
  6. PR #2146: Migrated fragment to viewbinding
  7. PR #2149: Migrated adapters to viewbinding
  8. PR #2149: migrated savings,update,user and loan fragments to viewbinding

- We had created a document as to why this upgradation was needed , you can read that from here

Dagger2 to Hilt

- One of the prominent aspect of this years idea list was to integrate Hilt in our project for dependency injection. While both of them could have existed together , I wanted to get rid of dagger entirely. This was the earlier package structure

```
injection/
|-- ActivityContext.kt
|-- ActivityContext.kt
|-- component/
    |-- ActivityComponent.kt
    |-- ApplicationComponent.kt
|-- module/
    |-- ActivityModule.kt
    |-- ApplicationModule.kt
```

- After the migration to Hilt we no longer needed a bulk of the above mentioned files. Hence after cleaning up the unecessary files we only needed the following files :

- `ApplicationModule.kt` : This offers methods that utilize annotations to provide instances of PreferencesHelper for handling application preferences, BaseApiManager for API interactions, and DataManager for data management. These instances are initialized with necessary dependencies, enhancing code modularity and maintainability through efficient dependency injection.

- `RepositoryModule.kt` : This is responsible for providing the instances of repository interfaces such as *UserAuthRepository*, *SavingsAccountRepository*, *LoanRepository*, and others. These repositories are initialized with a DataManager instance, ensuring access to data management functionalities across the application.

- `NetworkModule.kt` : This is responsible for providing Retrofit instance and it also provides all the api that is being used in our project. Earlier all of this was being handled in the `BaseApiManager.kt` class but we wanted to leverage the power of hilt and this is what we have done by refactoring the mentioned class. In the future any retrofit related instance should be provided here.

This is the new package structure:

```
injection/module/
|-- ApplicationModule.kt
|-- RepositoryModule.kt
|-- NetworkModule.kt
```

- Advantages of using Hilt in the project :

  - No need to create factory for any viewmodels now, just annotate the viewmodels with `@HiltViewModel`
  - Eliminated much of daggers' intricacies
  - In future if our projects grows then Hilt will ensure its scalibility

- This Pull Request shows everything that was modified.

MVP to MVVM

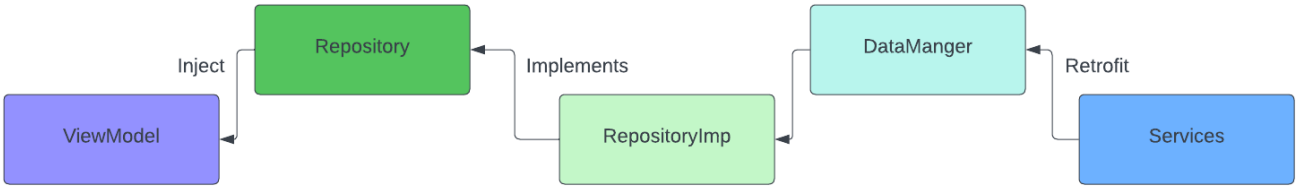
- Before the start of GSoc 2023, Mifos Mobile was centered around MVP architecture. However in this years' GSoC we have entirely migrated the project to MVMM style of architecture. The repositories are returning back data from the `DataManger.kt` and we are injecting the repository in its corresponding viewmodels' constructor.

- We have created a repositories package that lists all of the repository interfaces and their implementation. The reason for creating an interface was that it followed two fundamental SOLID principles:

  - `Open/Closed`: By Using an interface for repositories we made sure that they were open for extension but closed for modification. We can extend the repository by creating new implementations without modifying existing code.

    - `Dependency Inversion`: High-level modules (like ViewModel or UI) depend on abstractions (interfaces), not concrete implementations. This promotes flexibility, as we can change the implementation without affecting the higher-level component

- Besides, the above mentioned points by creating an interface we are abstracting away the actual implementation details from the rest of the application. This separation of concerns allows other parts of the app (like ViewModel or UI) to interact with the repository using a well-defined contract without needing to know the internal workings.



- We have used LiveData as well as StateFlow in some of our viewmodels to observe the states within our application.We also have sealed classes that holds the object and data classes for MutableLiveData/MutableStateFlow in the utils package of our project. I have listed down all of the PRs related to MVVM migration :

1. PR #2199: Recent transaction fragment to mvvm
2. PR #2209: Migrated loan related presenters to mvvm
3. PR #2218: Beneficary related fragments to mvvm
4. PR #2222: Migrated help fragment to mvvm
5. PR #2193: Guarantor related fragments migrated to MVVM
6. PR #2148: Transfer process migrated to mvvm & implemented coroutines
7. PR #2199: Thirdparty transfer migrated to mvvm & implemented coroutines
8. PR #2199: Client charge migrated to mvvm
9. PR #2253: QrCodeImport migrated to MVVM
10. PR #2273: User & Home fragment migrated to mvvm with state flow

Implementing Coroutines and Flows

- Earlier we were using RxJava to perform asynchronous and event based programming however it had problems of its own such as overhead, memory leaks, testing difficulty and things as such. To overcome most of the RxJava related issues we decided to switch to kotlin Coroutines in our project. While using coroutines we had removed Observable from methods return type. In most places we are using the Responses provided to us by Retrofit but in some places we have gone with Kotlin Flows. Advantage of using Flows were is that it provided us with a lot of Apis such as *catch*, *onCompletion*, *onStarted* etc which were really useful for our usecase

- These are some of the PRs related to coroutine integration:

1. PR #2271: Implemented coroutines in notification viewmodel
2. PR #2273:Implemented Flows in User & Home Fragment
3. PR #2279: Implemented Flows in Guarantor related viewmodel
4. PR #2280: Implemented Flows in Loan related viewmodel

5. PR #2250: Implemented coroutines in ThirdPartyTransfer viewmodel
6. PR #2248: Implemented coroutines in Transfer Process viewmodel

- The above PRs are the ones that specifically deals with the implementation of Coroutines and Flows in our project. However, there were PRs related to MVVM in which I had implemented coroutines. Such PRs have already been mentioned above

Unit test

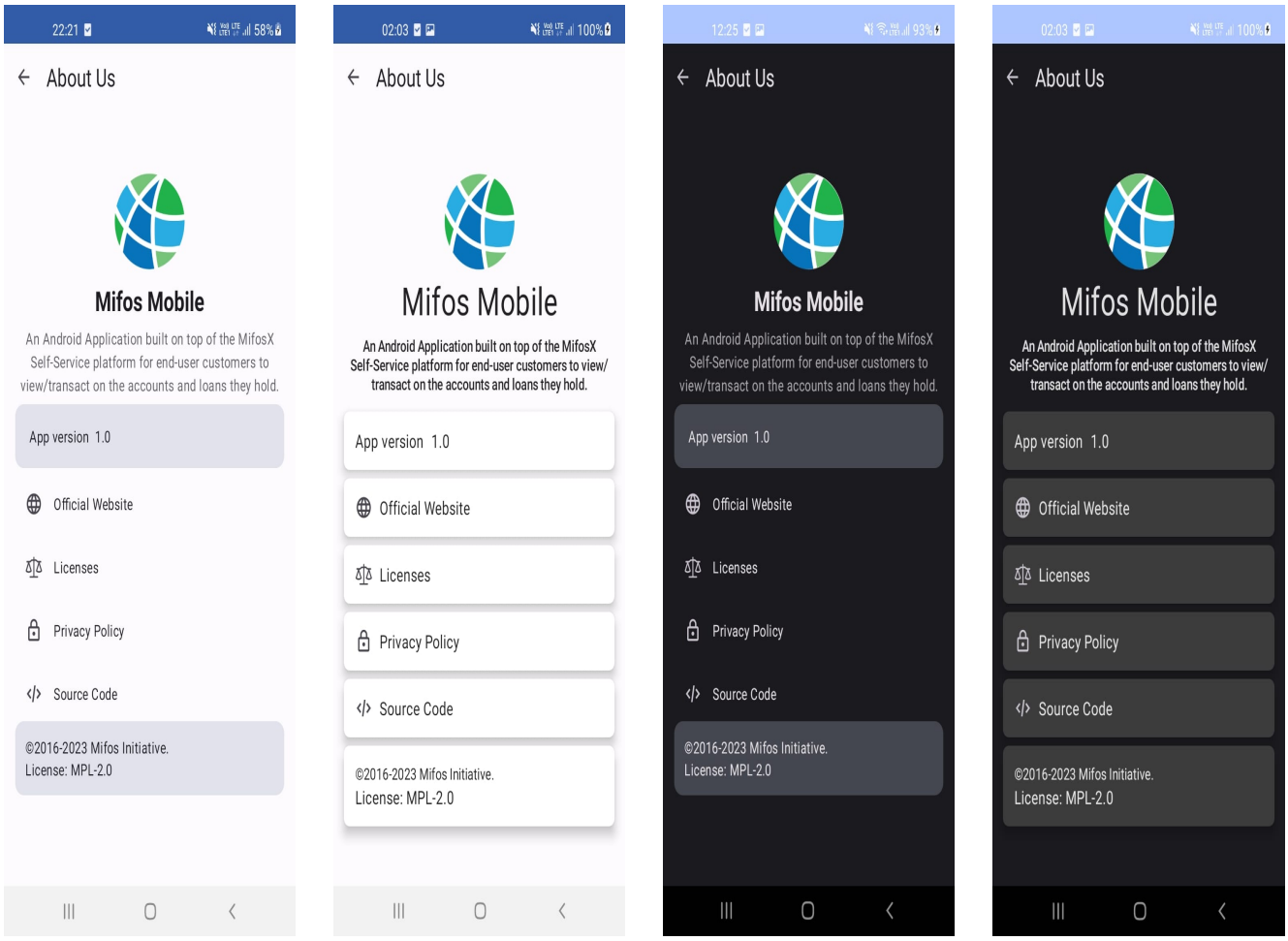
- We made sure that all of our viewmodels & repository implementations were covered with unit tests at all times. We wrote tests for the repositories as well as viewmodels in all of the MVVM related Pull Requests. Once we had covered all of our presenters we got rid of the presenters tests package.

- We have used two unit testing frameworks namely `Mockito` & `Turbine`. While *Mockito* is used in almost all of the repositories & viemodels , our usage of turbine is restricted to only those repositories/viewmodels that have *kotlin flows* associated with them. Turbine predominantly helps us to test the stateflows that we have used in our viewmodels.

- This is Pull Request contains file changes related to turbine . Anyone can take this as a reference and implement the same in the future

Xml to Compose View

- We had migrated the AboutUs screen from xml to compose and this contains all that was changed . However the pull requested raised for this wasn't upto the standard hence it was reverted. We are now planning to have an entirely different module for compose related components/widgets. This would not only make our project modular but also help future developers to comprehend and contribute in a systematic manner.



- Second & Fourth image is what we want to come up with whereas first and third were the xml designs.

Project Upgradation

- Mifos Mobile was earlier running on kotlin version `1.6.21` and it was also using deprecated `kotlin-android-extension`. I have now upgraded the project to `1.8.10` and also removed the deprecated version in favour of viewbinding. Moreover, this upgradation ensured that our module created for compose compiles successfully.

- Earlier we had a single module `app` that stored everything for us however we have created another module by the name of `core` which will store all compose related ui/widgets . Moreover in the next iteration we can further modularize our project which would be really helpful when migrating towards *Kotlin Multiplatform*

- Older version of kotlin had turned out to be a major blocker while running unit tests and upgrading it to `1.8.10` made sure that we were able to run our tests without any hassle

- This Pull Request contains all the files that were upgraded/removed at the time of refactoring

Improving Github Actions

- Earlier our project pipeline was capable of running only the most basic checks. Even though those jobs were helpful the most crucial jobs to run the unit tests were missing. In such a scenario anyone can push the code by writing wrong unit tests and our pipeline wouldn't be able to serve its purpose entirely.

- Hence, I've added a job to run unit tests for our project and at the same time I have ran an action to store the test reports as artifacts in case of a failure. The following snippet is of the new job that I have added:

```
  test:
    name: Unit Tests
    runs-on: ubuntu-latest
    steps:
      - name: Checking out repository
        uses: actions/checkout@v3

      - name: Test App
        run: ./gradlew test

      - name: Upload Test Report
        uses: actions/upload-artifact@v3.1.2
        if: failure()
        with:
          name: test-reports
          path: app/build/reports/
```

- This would ensure that our unit tests run on every pull requests. Besides this, maintaining the code quality & readability is crucial for any developers and it will surely help future developers as well in comprehending the codebase and for that purpose I have added jobs for checkstyle

```
checkstyle:
    name: Checkstyle
    runs-on: ubuntu-latest
    steps:
      - name: Checking out Repository
        uses: actions/checkout@v3

      - name: Checkstyle
        run: ./gradlew checkstyle

      - name: Upload Checkstyle Reports
        uses: actions/upload-artifact@v3.1.2
        with:
          name: checkstyle-reports
          path: app/build/reports/
```

- One of the most bothering aspect of our project were the stale issues & pull requests . Naturally, this isn't something that should happen and for that purpose I've added a workflow that will be responsible for marking stale issues/pull requests in our project

```
on:
  schedule:
```

```
    - cron: "35 * * * *"

jobs:
  stale:

    permissions:
      issues: write
      pull-requests: write
    runs-on: ubuntu-latest

    steps:
      - uses: actions/stale@v8
        with:
          repo-token: ${{ secrets.GITHUB_TOKEN }}
          stale-issue-message: <MESSAGE THAT WILL BE COMMENTED INCASE OF STALE
ISSUE>
          stale-pr-message: <MESSAGE THAT WILL BE COMMENTED INCASE OF STALE PULL
REQUEST>
          stale-issue-label: 'no-activity'
          stale-pr-label: 'attention-needed'
          days-before-issue-stale: 90
          days-before-pr-stale: 14
```

Impact

- The transformation of Mifos mobile project from MVP to MVVM architecture marks a significant stride in refining code structure and organization. This shift not only promotes a clearer separation of responsibilities but also sets the stage for more modular and maintainable development. The introduction of unit tests for repositories and view models contributes to heightened code quality, instilling confidence in the functionality and mitigating the risk of regressions down the line.

- By compartmentalizing Jetpack Compose into its dedicated module, we've tactfully decoupled UI development, streamlining updates and allowing for independent evolution. Integration of Hilt for dependency injection brings about a more intuitive and manageable way to handle dependencies, bolstering scalability and easing the integration of new features.The decision to upgrade Kotlin versions and purge deprecated plugins reflects a commitment to staying current and aligned with the dynamic Android landscape. This not only ensures access to the latest language features but also enhances security and performance.

- These collective enhancements forge a robust foundation for the future, simplifying development processes, accelerating issue resolution, and positioning the Mifos mobile project advantageously to embrace forthcoming advancements in the Android realm.

What more can be done?

- Currently the package structure especially the way ui package is structured is really primitive and it should be reformed in the next iteration. This would allows us to organize our project in a more efficient manner.

- In this GSoC we could convert only the `AboutUsFragment` from xml to compose view but in the next iteration covering the entire project with jetpack compose will be a huge plus
- Since we have started with modularization of our project, I believe modularizing it further based on the architecture would make our project more scalable which would eventually be crucial if we ever plan to migrate to *Kotlin Multiplatform*
- Currently we are using SQLite in our project, however migrating it to Room would be an awesome code refactor . Similarly getting rid of SharedPreference in favour of DataStore would again be a decent code refactoring
- Covering our application with UI tests either by using espresso or UiAutomator or Roboelectric would be really great
- If we could migrate from `build.gradle` to `build.gradle.kts` then it would bring the advantages of Kotlin's expressive syntax, type safety, and intelligent refactoring to our build scripts.

Conclusion

- Mifos Mobile is now at a way better place than it was three months back and I am sure it will just get better with the next iteration. With the landscape of Android Development ever evolving, constantly refactoring the code and keep a check on the quality will be the key
- Personally, this year's GSoC has been an incredibly enriching experience for me. The guidance and wisdom shared by my mentors, stemming from their extensive years of experience, have catalyzed a remarkable transformation in my skills as a software developer, evident even in comparison to just three months ago. My journey has traversed the landscape of knowledge, from acquiring a profound understanding of SOLID principles to seamlessly implementing them within our project. All these strides are solely attributed to the profound impact of the Google Summer of Code.