Google Summer of Code

Mifos

GSOC'25 PROPOSAL - Mifos Initiative

Migrate Android Client to Kotlin Multiplatform

Organization: Mifos Initiative

Project Name: Migrate Android Client to Kotlin Multiplatform

Candidate Name: Pronay Sarker

Expected Project Size: 350 hours

Mentors:

- Rajan Maurya
- Shashank Priyadarshi
- Chinmay Kulkarni

Contents

1. Project Idea

This year, the project will focus on migrating the Mifos Android field officer client from a Kotlin multi-module Android architecture to Kotlin Multiplatform (KMP), enabling cross-platform support for Android, iOS, desktop, and web. The migration process will involve updating the Android client to the latest dependencies to ensure compatibility with modern libraries and technologies.

The network layer of the Android client will be rewritten to consume the Fineract SDK using Coroutines rather than the existing RxJava implementation. This change aims to simplify the codebase and improve performance and maintainability. Additionally, the Kotlin multi-module architecture will be migrated to Kotlin Multiplatform, which will make the app more scalable and compatible across various platforms, reducing redundancy and enhancing long-term development efficiency.

To further improve the application's reliability, appropriate unit and integration tests will be written to verify the core functionality and prevent regressions. The CI/CD pipeline will also be updated to build the APK and analyze code quality automatically, ensuring the app's continuous delivery and maintaining high standards.

To streamline the release process, a Playstore release pipeline will be implemented using GitHub Actions and integrated with Fastlane. This will automate the app's deployment, making the release flow faster and more efficient. Additionally, the Google Play Store API will be integrated to enhance the release management and enable quicker updates and rollouts.

Finally, all the necessary documentation for building and deploying the app will be updated to reflect these changes, ensuring future developers have clear instructions for maintaining and improving the app.

2. Implementation of the idea

2.1 Rewrite Network layer to consume fineract SDK

The Mifos Android Client, Mifos Mobile, and Mobile Wallet currently contain all their network and API-related logic within each app, despite sharing the same backend. This results in duplicated code, increased maintenance overhead, and limited scalability. We will replace these manual network calls with the Fineract KMP SDK, a Kotlin-based library that simplifies interaction with the Apache Fineract 1.x platform. Built on top of the Fineract Client KMP, the SDK abstracts client and service layers, supports multiple platforms via the KtorFit network library, and leverages Kotlin Coroutines and Flows for asynchronous operations, streamlining the network layer and improving maintainability and scalability across the apps.

- I will start by adding SDK dependencies in the `build.gradle` file.

```
dependencies {
    implementation("com.github.openMF:fineract-client-kmp-sdk:$sdkVersion")
  }
```

- Next, I will configure the base URL, credentials, and tenant ID, then initialize the KtorFit client using `BaseApiManager`

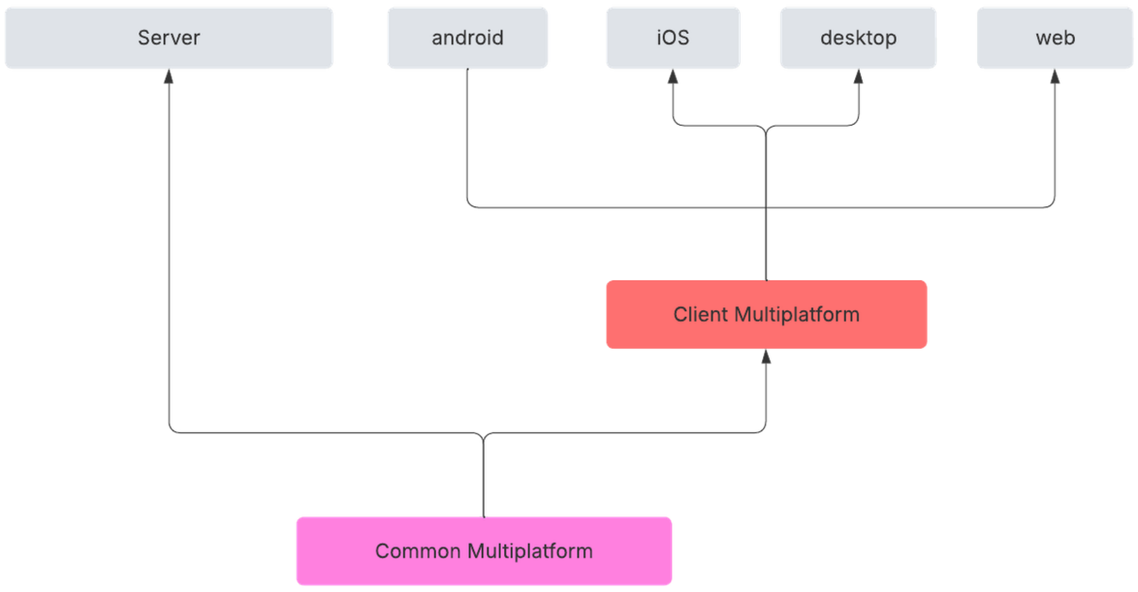
```
val baseUrl = PROTOCOL_HTTPS + API_ENDPOINT + API_PATH
val tenant = "default"
val username = "mifos"
val password = "password"
val baseApiManager = BaseApiManager.getInstance()
baseApiManager.createService(username, password, baseUrl, tenant, false)
```

- After adding the necessary dependencies and setting up the project, I will begin by refactoring the existing network layer to use the Fineract KMP SDK. As a starting point, here is a demo implementation showing how I will replace network layer to consume the fineract SDK

```
val req = PostAuthenticationRequest(username = username, password =
password)

lifecycleScope.launch(Dispatchers.IO) {
    val response = baseApiManager.getAuthApi().authenticate(req, true)
    Log.d("Auth Response", response.toString())
}
```

2.2 Migrate kotlin multi-module codebase to Kotlin Multiplatform

Kotlin Multiplatform (KMP) is a cross-platform development tool that allows us to write common code once and share it across multiple platforms like Android, iOS, Web, Desktop, and Backend, while still enabling platform-specific customizations. Unlike other cross-platform frameworks, KMP does not enforce a single UI framework—it integrates natively with existing platform code (Swift for iOS, Kotlin/Java for Android, etc.).



This approach allows us to reuse business logic (networking, data storage, API calls, etc.) while maintaining full control over platform-specific features such as UI and hardware access. KMP is officially supported by JetBrains and is widely adopted by companies like Netflix, Philips, and VMWare for building scalable, high-performance applications.

We have already started migrating this project to Kotlin Multiplatform (KMP) to enable code sharing across platforms while maintaining native UI experiences. As part of this process, I have submitted pull requests, and all of them got merged.

- PR #2354 - feat: Migrate [core/database] to KMP
- PR #2351 - feat: Core/common to KMP

2.2.1 Migrate core modules to KMP

The first step in the migration process is to replace existing Android-specific Gradle plugins with the KMP plugin, which enables multi-platform support. This will allow us to restructure the project so that shared logic can be written once and compiled for multiple targets, including Android and iOS.

```
plugins{
    alias(libs.plugins.mifos.kmp.library)
}
```

Dependency Management: KMP allows for both shared and platform-specific dependencies, which means we will categorize our dependencies accordingly:

- **Common Dependencies**: Libraries that support all platforms (such as `Kotlinx`, `ktor`, and `serialization`) will be placed in the source set.
- **Platform-Specific Dependencies**: Some dependencies, such as database frameworks and dependency injection libraries, are platform-dependent. These will be added to the appropriate platform-specific source sets (`androidMain`, `iosMain` etc)

```
kotlin {
    sourceSets {
        commonMain.dependencies {
            // commonMain is where shared code and dependencies go
            // These are available to all platforms
        }

        androidMain.dependencies {
            // Android-specific source set
        }

        iosMain.dependencies {
            // iOS-specific source set
        }

        jvmMain.dependencies {
            // jvmMain-specific source set
        }
    }
}
```

After adding the dependencies, we need to organize the codebase into platform-specific directories to properly separate shared logic from platform-dependent implementations. Since not all functionalities can be shared across Android, iOS, and desktop, platform-specific directories will ensure that each platform has its own tailored implementation while maintaining a common shared code structure.

- **Defining the Shared Module:** The shared module, which is commonMain, will contain business logic, data models, and APIs that are independent of platform-specific details. This will be the core of the KMP implementation.

- **Setting Up Platform-Specific Source Sets:** To support platform-specific implementations, the project structure will be refactored to include separate source sets for each platform

`androidMain` → Contains Android-specific implementations (e.g., Jetpack libraries, Android framework dependencies).
`iosMain` → Holds iOS-specific implementations (e.g., Swift interop, iOS-specific APIs).
`desktopMain`→ Includes code for desktop-based execution.

How will we implement platform-specific code?

- Kotlin Multiplatform allows defining common code in the `commonMain` source set while expecting platform-specific implementations in the respective platform modules using the expect and actual mechanism. As example, in commonMain, we define an expect class or function, which serves as a placeholder for platform-specific implementations.

```
expect class Platform() {
    fun getPlatform() : String
}
```

- Each platform will have its own actual implementation of the expect class, providing platform-specific behavior. The expect class is defined in commonMain, while each platform-specific module implements its own actual version. For example, in the Android module, the actual implementation of the expect class would look like this:

```
import android.os.Build

actual class Platform {
    actual fun getPlatform(): String {
        return "Android ${Build.VERSION.SDK_INT} (${Build.MODEL})"
    }
}
```

This approach allows us to share common logic while ensuring that platform-specific functionalities are handled independently in their respective modules.

2.2.2 Migrate feature modules to CMP

Once the core modules have been successfully migrated to Kotlin Multiplatform (KMP), the next step will be transitioning the feature modules to Compose Multiplatform (CMP). This migration will ensure that the UI components work seamlessly across Android, iOS, and desktop platforms, leveraging Jetpack Compose's declarative UI approach to create a consistent and maintainable UI layer.

Migration Process This transition follows a similar strategy as used during the Kotlin Multiplatform (KMP) migration, ensuring a structured and efficient refactoring process.

Example: Migrating `feature/about`
Current `build.gradle.kts` (Android-based):

```
plugins {
    alias(libs.plugins.mifos.android.feature)
    alias(libs.plugins.mifos.android.library.compose)
    alias(libs.plugins.mifos.android.library.jacoco)
}

android {
    namespace = "com.mifos.feature.about"
}

dependencies {
    implementation(projects.core.designsystem)
}
```

As seen above, the current setup uses Android-specific plugins.

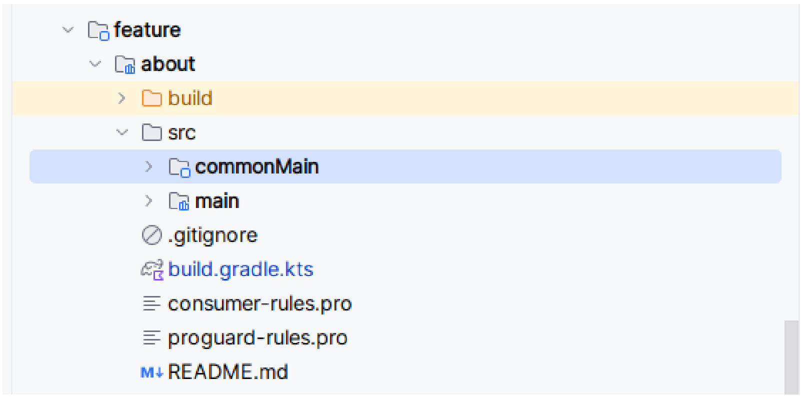
- **Step 1: Replace Android Plugins with CMP Plugin** To enable Compose Multiplatform (CMP) support for this module, we need to replace the Android plugins with the CMP equivalent. I have already added CMP-specific plugins in build-logic in a previous PR, so we can directly apply them here.

Updated `build.gradle.kts` (CMP-enabled):

```
plugins {
    alias(libs.plugins.mifos.cmp.feature)
}

android {
    namespace = "com.mifos.feature.about"
}

dependencies {
    implementation(projects.core.designsystem)
}
```

- **Step 2: Sync the Project** After updating the plugin, sync the project to apply the changes and verify that the CMP configuration takes effect properly.

- **Step 3: Adding Source Sets (commonMain, androidMain, etc.)** With CMP enabled, we no longer declare dependencies globally in dependencies { }. Instead, we define them within source sets.To begin this, create the appropriate directory structure under `src`. Right-click on `src`, Select `New` → `Directory` and then add `commonMain` (optionally: androidMain, iosMain, desktopMain, etc. depending on our targets). Now, our directory would look like this



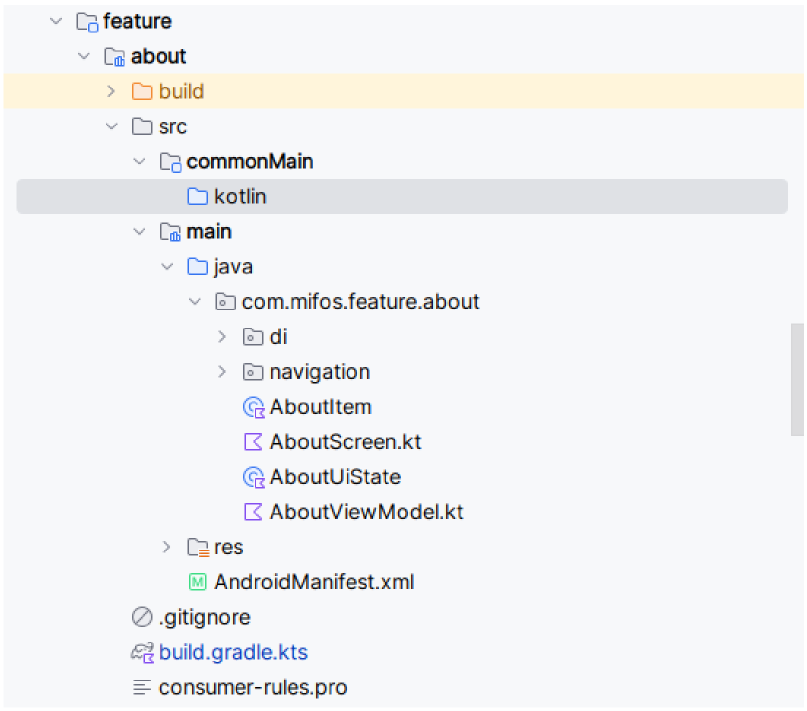
```
feature
  about
    build
    src
      commonMain
      main
    .gitignore
    build.gradle.kts
    consumer-rules.pro
    proguard-rules.pro
    README.md
```

- **Step 4: Add Dependencies to platform specific modules** Depending on how many modules we have added, now we have to add dependencies to our platform specific modules. In Compose Multiplatform, shared dependencies like `core.common` should be declared inside the `commonMain` source set block. So, Now we will modify our `build.gradle.kts` file and add those dependencies

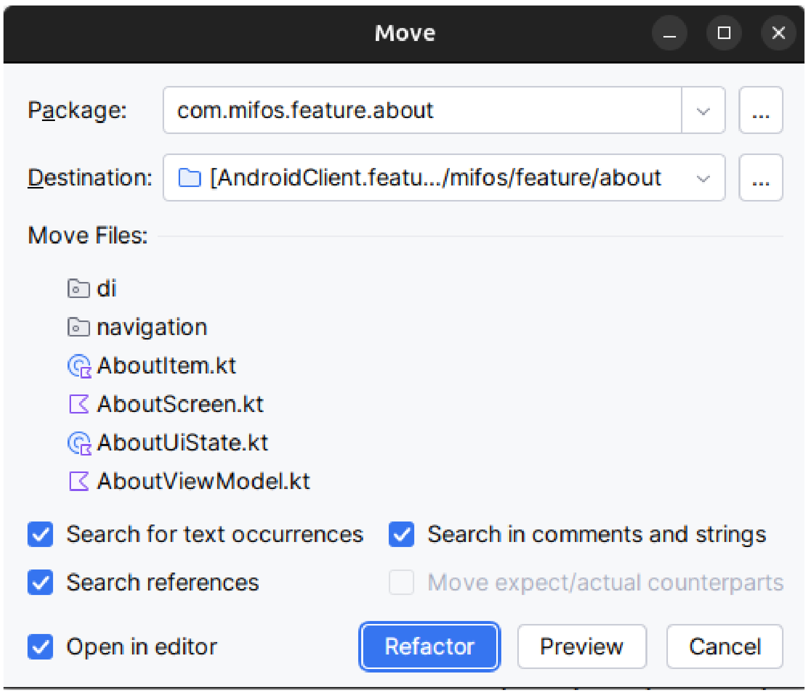
```
plugins {
    alias(libs.plugins.mifos.cmp.feature)
}

android {
    namespace = "com.mifos.feature.about"
}


kotlin {
    sourceSets {
        commonMain {
            dependencies {
                implementation(projects.core.designsystem)
            }
        }
    }
}
```

Since, we have already migrated `core.designSystem` module to KMP, now we can put it inside `commonMain`. This ensures `core.common` is accessible across all platforms supported by the module. After that, we have to sync again.

- **Step 5: Move `main` sourceSets into `commonMain`** : If you notice, you will see that our `commonMain` sourceSet is totally empty.


```
feature
  about
    build
    src
      commonMain
        kotlin
      main
        java
          com.mifos.feature.about
            di
            navigation
            AboutItem
            AboutScreen.kt
            AboutUiState
            AboutViewModel.kt
        res
        AndroidManifest.xml
    .gitignore
    build.gradle.kts
    consumer-rules.pro
```

Now, we need to move all the Android code from `main` to `commonMain`. This ensures that our code is supported across all platforms. While moving the files, make sure to select the **Refactor** option. This will ensure we don't break any functionality, as the codes in the main directory must have been used somewhere in the project.


Move

Package: com.mifos.feature.about

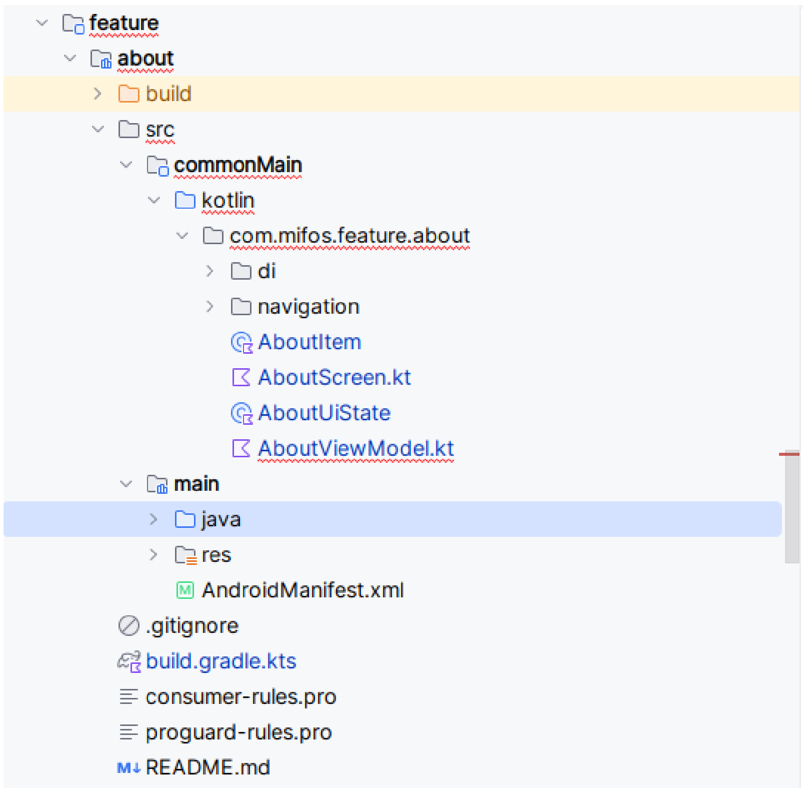
Destination: [AndroidClient.featu.../mifos/feature/about

Move Files:

- di
- navigation
- AboutItem.kt
- AboutScreen.kt
- AboutUiState.kt
- AboutViewModel.kt

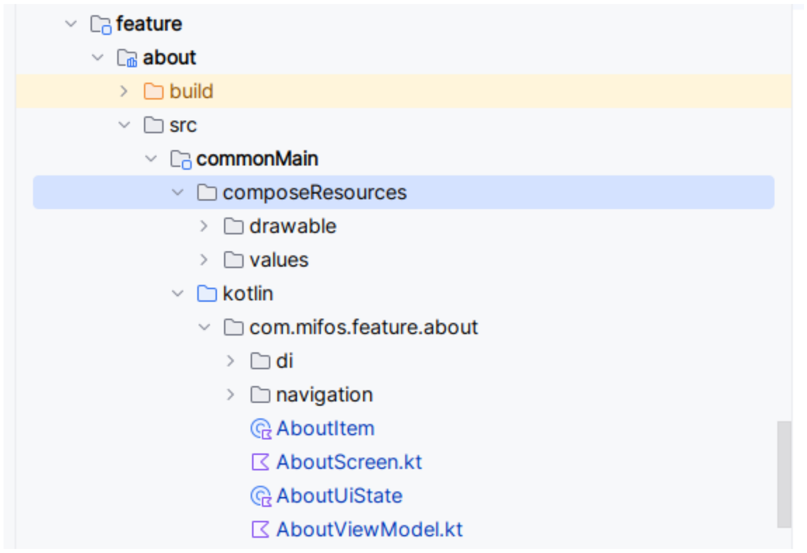
[✓] Search for text occurrences
[✓] Search in comments and strings
[✓] Search references
[ ] Move expect/actual counterparts
[✓] Open in editor

After we are done, it will take some time to move our code to `commonMain` directory and it would look like this.



```
feature
  about
    build
    src
      commonMain
        kotlin
          com.mifos.feature.about
            di
            navigation
            AboutItem
            AboutScreen.kt
            AboutUiState
            AboutViewModel.kt
      main
        java
        res
        AndroidManifest.xml
    .gitignore
    build.gradle.kts
    consumer-rules.pro
    proguard-rules.pro
    README.md
```

- **Step 5: Move the `res` folder from `main` to `commonMain`** : Following the same process, we need to move all our resource files from the `res` directory to `commonMain/composeResources`. To do this, we will first create a directory called `composeResources` inside `commonMain`. Then, following the same approach, we will move all the resource files from `main/res` to `commonMain/composeResources`.



```
feature
  about
    build
    src
      commonMain
        composeResources
          drawable
          values
        kotlin
          com.mifos.feature.about
            di
            navigation
            AboutItem
            AboutScreen.kt
            AboutUiState
            AboutViewModel.kt
```

- **Step 6: Resolve Compilation Issues** : Now, we have to go through each file and check for errors. The most common issues are resource-related. To fix these, we have to generate Compose resource accessors by running the following command in the terminal `./gradlew :feature:about:generateResourceAccessorsForCommonMain` Once the resources are generated, most errors can be resolved by importing the correct references. We might also encounter issues if the module uses Android-specific libraries that aren't supported in Compose Multiplatform. In such cases, we have toreplace them with KMP-compatible alternatives. Which varies for module-to-module.

I have already migrated a feature module to CMP in the Mifos Mobile project, Which got merged.

- PR #2808: refactor: [feature/guarantor] migrate to CMP
- PR #2802 - refactor: migrate feature/update-password to CMP

2.4 Use Dynamic Navigation

Dynamic navigation refers to the ability to create, modify, or adapt navigation paths at runtime rather than having them statically defined at compile time. If we use dynamic navigation in our app, we can give Role-Based Access Control.

- We can show only the screens a user has permission to access
- Adapt navigation based on the user's role
- Hide functionality that requires permissions the user doesn't have

Here is an example of Dynamic Navigation in Compose. We can set up permissions for each role and grant them access role-wise.

```
// Simple permission model
data class NotePermission(
    val entityTypePermissions: Map<String, Boolean> = emptyMap()
)

// Dynamic note screen that checks permissions
fun NavGraphBuilder.dynamicNoteScreen(
    onBackPressed: () -> Unit,
    notePermissions: NotePermission
) {
    composable(
        route = NoteScreens.NoteScreen.route,
        arguments = listOf(
            navArgument(name = Constants.ENTITY_ID) {
                 type = NavType.IntType
                },
            navArgument(name = Constants.ENTITY_TYPE) {
                 type = NavType.StringType
                },
        ),
    ) {
        val entityType = it.arguments?.getString(Constants.ENTITY_TYPE)
        val hasPermission =
notePermissions.entityTypePermissions[entityType] ?: false

        if (hasPermission) {
            NoteScreen(onBackPressed = onBackPressed)
        } else {
            AccessDeniedScreen(onBackPressed = onBackPressed)
        }
    }
}

// Permission-aware navigation
```

```
fun NavController.safeNavigateToNoteScreen(
    entityId: Int,
    entityType: String?,
    notePermissions: NotePermission
) {
    if (entityType != null &&
notePermissions.entityTypePermissions[entityType] == true) {
        navigate(NoteScreens.NoteScreen.argument(entityId, entityType))
    }
}

// Simple access denied screen
@Composable
fun AccessDeniedScreen(onBackPressed: () -> Unit) {
    // Access denied UI implementation
}

// Usage example
fun getRolePermissions(role: String): NotePermission {
    return when (role) {
        "admin" -> NotePermission(mapOf("clients" to true, "loans" to
true))
        "user" -> NotePermission(mapOf("clients" to true, "loans" to
false))
        else -> NotePermission()
    }
}
```

2.4 Write appropiate unit and integration tests

As we focus on making the Android client scalable through numerous migrations, we should also prioritize the testing aspect, which is crucial for scalability and efficiency.

2.4.1 Creating a Test Module

A common practice is to create a commonTest folder (or module) to store test files for the code in common. In a typical Gradle-based multiplatform project, the structure might look like this

```
common/
   src/
      commonMain/
      commonTest/   <- this is where the tests go
```

Adding the Necessary Dependencies: We need to connect several libraries in commonTest to begin with:

1. kotlin-test

```
    implementation("org.jetbrains.kotlin:kotlin-test:2.1.10")
```

This library provides a set of testing tools (for instance, assertEquals, assertTrue), annotations for testing (such as @Test), and the ability to run tests across different targets.

2. kotlinx.coroutines-test

```
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-test:1.10.1")
```

This library simplifies testing asynchronous code that uses coroutines. It provides the ability to use runTest, manage virtual time (for example, using runCurrent), and helps avoid race conditions and resource leaks.

3. koin-test

```
    implementation("io.insert-koin:koin-test:4.0.0")
```

This library makes it easier to work with the Koin DI framework in a testing environment. We can override real dependencies with fake or mock implementations, isolate and test individual components, and ensure they behave correctly without requiring full application initialization.

2.4.2 Determining Classes to Test

First and foremost, we need to decide which classes we will be testing. In a typical multiplatform architecture, these often include:

Repositories (accessing an API or database), Use Cases (business logic),ViewModels (contains the logic for preparing data for the UI), and any other components containing functionality that needs verification.

2.4.3 Creating Fake Classes Instead of Mocks

Currently, there's no stable support for libraries such as Mockito or MockK in KMP (though that may change in the future). Therefore, one practical approach is to write fake (manual) implementations.

We have a repository interface, for example:

```
interface LoanRepository {
    suspend fun getDemoTemplateFromApi(): LoanTemplate
}
```

we can create a fake class for testing:

```
class FakeApiRepository : LoanRepository {
    var demoTemplate: LoanTemplate = LoanTemplate()

    override suspend fun getDemoTemplateFromApi(): LoanTemplate {
        return demoTemplate
    }
}
```

2.4.4 DI Test Module

To manage dependencies in a test environment, it’s convenient to create a test Koin module that will override real dependencies and replace them with fake implementations:

```
val overrideModule = module {
    single<LoanRepository> { FakeApiRepository() }
}
```

Then, in the test class itself, we add this module to the existing (real) module:

```
@BeforeTest
fun setup() {
    startKoin {
        modules(testModule + overrideModule)
    }
}
```

Here, testModule is our main module declaring dependencies (for example, the implemented Use Cases).

2.4.5 Example Use Case Test

We have `GetDemoTemplateFromApiUseCaseImpl`, which depends on `LoanRepository`:

```
class GetDemoObjectsFromApiUseCaseImpl(
    private val loanRepository: LoanRepository
) : GetDemoTemplateFromApiUseCase {

    override suspend fun execute(params: Nothing?): LoanTemplate {
        return loanRepository.getDemoTemplateFromApi()
    }
}
```

Since we want to test the use case logic itself, we replace the repository with a fake (via Koin). The test class might look like this:

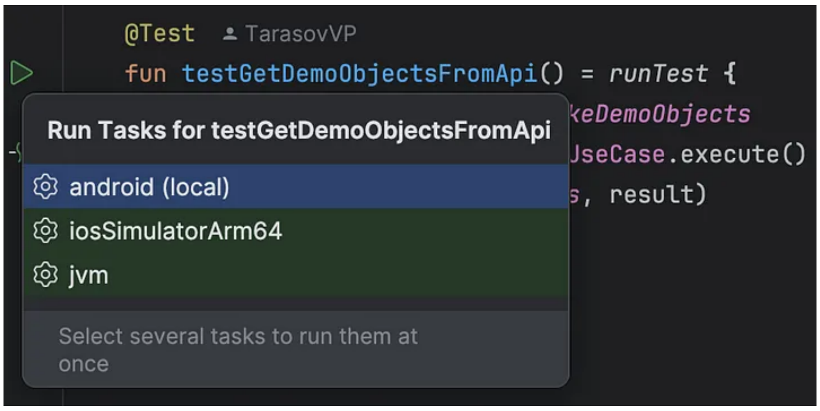
```
class GetDemoObjectsFromApiUseCaseTest : KoinTest {

    private val overrideModule = module {
        single<LoanRepository> { FakeApiRepository() }
    }

    private val getDemoTemplateUseCase by
inject<GetDemoTemplateFromApiUseCase>()

    private val repository by injectAs<LoanRepository, FakeApiRepository>()

    @BeforeTest
    fun setup() {
        startKoin {
            modules(testModule + overrideModule)
        }
    }

    @Test
    fun getDemoTemplateFromApi() = runTest {
        val expected = createFakeDemoObjects()
        repository.demoObjects = expected

        val result = getDemoTemplateUseCase.execute()
        assertEquals(expected, result)
    }

    @AfterTest
    fun tearDown() {
        stopKoin()
    }
}
```

- `overrideModule` replaces the implementation of `LoanRepository` with `FakeApiRepository`.
- In `@BeforeTest`, `startKoin(...)` is called to initialize the DI container.
- In the `@Test` method, we prepare the test data (expected), pass it to the fake repository, and then call the tested use case to compare the result.
- In `@AfterTest`, we stop Koin by calling `stopKoin()`.

2.4.6 Running Tests

After writing the tests, our IDE (or Gradle) will prompt us to choose the environment in which to run them. In KMP projects, there are usually a few options:

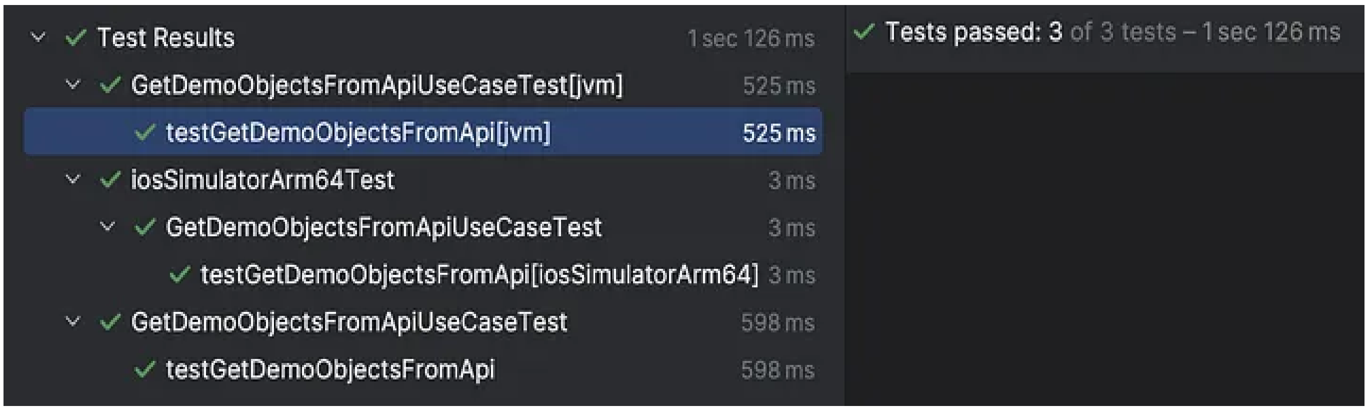
- **jvm** — runs tests on the JVM,
- **android** — (if configured) runs tests on Android (local or instrumented),
- **iosSimulatorArm64** (or other iOS variants).



```
@Test  TarasovVP
fun testGetDemoObjectsFromApi() = runTest {
                                    ...keDemoObjects
                                    ...UseCase.execute()
                                    ...s, result)
```

Run Tasks for testGetDemoObjectsFromApi
android (local)
iosSimulatorArm64
jvm
Select several tasks to run them at once

We can see the results of the tests in the run/test console output for each target platform.

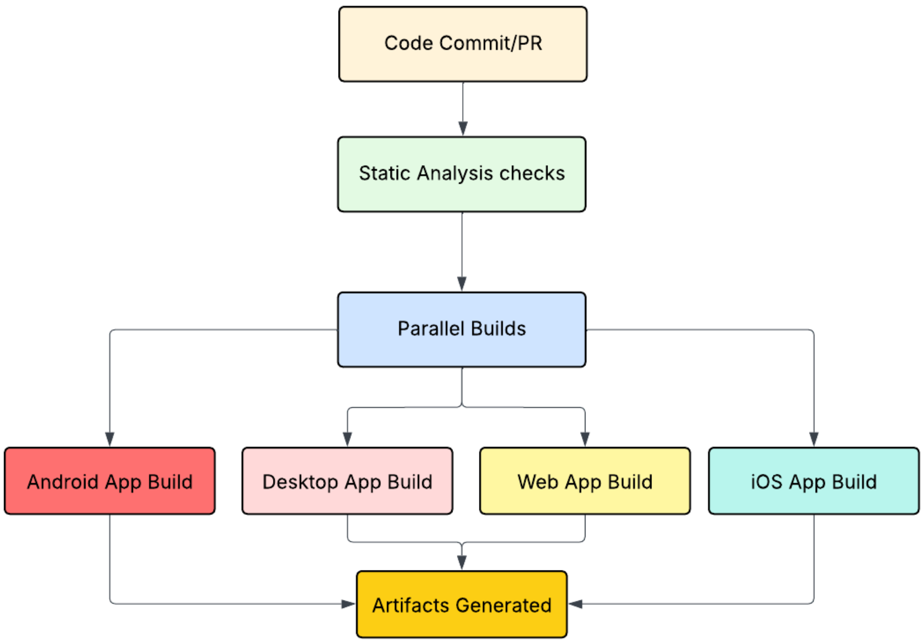


```
✓ Test Results                                              1 sec 126 ms
  ✓ GetDemoObjectsFromApiUseCaseTest[jvm]                   525 ms
    ✓ testGetDemoObjectsFromApi[jvm]                        525 ms
  ✓ iosSimulatorArm64Test                                   3 ms
    ✓ GetDemoObjectsFromApiUseCaseTest                      3 ms
      ✓ testGetDemoObjectsFromApi[iosSimulatorArm64]        3 ms
  ✓ GetDemoObjectsFromApiUseCaseTest                        598 ms
    ✓ testGetDemoObjectsFromApi                             598 ms

✓ Tests passed: 3 of 3 tests – 1 sec 126 ms
```

I will also have discussions with my mentor to review the initial tests and refine them based on feedback, ensuring that the tests are aligned with the project's overall goals and requirements.

2.5 Update CI/CD to build APK and analyse code quality

Mifos already have GitHub Actions Workflows for Multi-Platform App Development and this reusable GitHub Actions workflow provides a comprehensive Continuous Integration (CI) pipeline for multi-platform mobile and desktop applications, specifically designed for projects using Gradle and Kotlin Multiplatform (KMP).



Workflow Jobs:

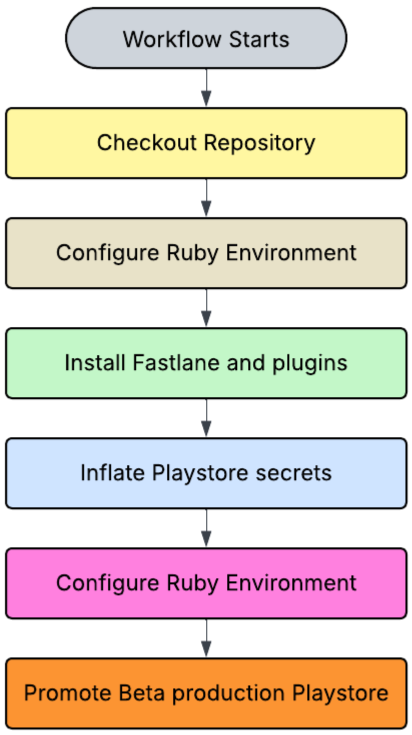
- **Static Analysis Checks**: Code quality check using custom static analysis.
- **Android App Build**: Builds debug APK on Ubuntu.
- **Desktop App Build**: Builds for Windows, Linux, MacOS (cross-platform).
- **Web Application Build**: Compiles web app on Windows.
- **iOS App Build**: Builds iOS app on MacOS.

Here is an example of its usage

```yaml
name: PR Checks

on:
  pull_request:
    branches: [ dev, main ]

jobs:
  pr_checks:
    name: PR Checks
    uses: openMF/mifos-mobile-github-actions/.github/workflows/pr-check.yaml@main
    with:
      android_package_name: 'mifospay-android'
      desktop_package_name: 'mifospay-desktop'
      web_package_name: 'mifospay-web'
      ios_package_name: 'mifospay-ios'
```

2.7 Implemented Playstore release github action pipeline

- In the fast-paced world of app development, the ability to deliver efficiently and quickly to market is vital for success. To optimize our release workflow and improve our app's performance on the Google Play Store, the Google Play Developer API plays a crucial role by automating various app management and publishing tasks.


Workflow Starts
Checkout Repository
Configure Ruby Environment
Install Fastlane and plugins
Inflate Playstore secrets
Configure Ruby Environment
Promote Beta production Playstore

Configuration Steps

Setup Firstlane Fastlane is an open-source toolchain that automates the process of building, testing, and deploying mobile apps (iOS, Android, macOS).

This would be our project structure

```
project-root/
│
├── buildLogic/              # Shared build configuration
├── gradle/                  # Gradle wrapper and configuration
│
├── core/                    # Core business logic module
│   ├── common/              # Common code shared across platforms
│   ├── model/               # Model classes and data structures
│   ├── data/                # Data models and repositories
│   ├── network/             # Networking and API clients
│   ├── domain/              # Domain-specific logic
│   ├── ui/                  # UI components and screens
│   ├── designsystem/        # App-wide design system
│   └── datastore/           # Local data storage
│
├── feature/                 # Feature Specific module
│   ├── feature-a/           # Feature-specific logic
│   ├── feature-b/           # Feature-specific logic
│   └── feature-c/           # Feature-specific logic
│
├── androidApp/              # Android-specific implementation
├── iosApp/                  # iOS-specific implementation
├── desktopApp/              # Desktop application module
├── webApp/                 # Web application module
│
├── shared/                 # Shared Kotlin Multiplatform code
│   ├── src/
│   │   ├── commonMain/     # Shared business logic
│   │   ├── androidMain/    # Android-specific code
│   │   ├── iosMain/        # iOS-specific code
│   │   ├── desktopMain/    # Desktop-specific code
│   │   ├── jsMain/         # Web-specific code
│   │   └── wasmJsMain/     # Web-specific code
│
├── Fastfile               # Fastlane configuration
├── Gemfile                # Ruby dependencies
└── fastlane/              # Fastlane configurations
```

Fastlane Installation

```
# Install Ruby (if not already installed)
brew install ruby

# Install Fastlane
gem install fastlane

# Create Gemfile
bundle init

# Add Fastlane to Gemfile
bundle add fastlane
```

we have to configure fastlane after installing fastlane.

After we are done, We will use Mifos mobile github action workflow to automate the promotion of a beta release to the production environment on the Google Play Store.

Prerequisites : Ruby Environment, Requires Ruby setup (uses `ruby/setup-ruby action`), Bundler version 2.2.27, Fastlane installed with specific plugins
Required Plugins: `firebase_app_distribution`, `increment_build_number`

- **Repository Setup** : We need to ensure our Android client repository is properly structured for Android app deployment, also We need to have a Fastfile configured with `promote_to_production` lane
- **Fastlane Configuration** : After that, we will create a Fastfile in our `fastlane` directory with a `promote_to_production` lane:

```
default(:android)
lane :promote_to_production do
  # Your specific Play Store promotion logic
  supply(
    track: 'beta',
    track_promote_to: 'production'
  )
end
```

- **GitHub Secrets** : also we have to ensure we have Play Store credentials configured in our repository secrets, which is in json format.

Here is an example

```
name: Promote Release to Play Store

# Workflow triggers:
# 1. Manual trigger with option to publish to Play Store
# 2. Automatic trigger when a GitHub release is published
on:
  workflow_dispatch:
    inputs:
      publish_to_play_store:
        required: false
        default: false
        description: Publish to Play Store?
        type: boolean
  release:
    types: [ released ]

concurrency:
  group: "production-deploy"
  cancel-in-progress: false

permissions:
  contents: write

jobs:
  # Job to promote app from beta to production in Play Store
  play_promote_production:
    name: Promote Beta to Production Play Store
    uses: openMF/mifos-mobile-github-actions/.github/workflows/promote-to-
production.yaml@main
    if: ${{ inputs.publish_to_play_store == true }}
    secrets: inherit
    with:
      android_package_name: 'android-client'
```

2.8 Update corresponding documentation for building the app

After completing all tasks, I will prepare a detailed document to support future development initiatives. This document will act as a reference for other developers involved in the project, promoting consistency and ensuring smooth continuity of design patterns.

I will use **docca plugin** and **cursor-ai** to generate documentations

Dokka: Kotlin's Official Documentation Engine :

Dokka is an API documentation engine for Kotlin. Dokka provides the foundation of our documentation strategy. As the official documentation engine for Kotlin, it generates comprehensive API documentation that captures the full structure and functionality of our codebase.

Key features of our Dokka implementation:

- **Structured API Documentation**: Dokka automatically extracts and organizes class hierarchies, method signatures, and property definitions into navigable documentation.
- **Kotlin-First Approach**: Unlike Javadoc, Dokka fully understands Kotlin-specific language features such as extension functions, nullable types, and coroutines.
- **Multi-Format Output**: Our documentation is available in HTML, Markdown, and Javadoc formats, allowing developers to access information in their preferred environment.
- **Source Code Integration**: All documentation is linked directly to the relevant source code, enabling developers to quickly navigate between documentation and implementation.
- **Module Organization**: The documentation is structured by modules, making it easy to understand the architecture of the application at different levels of detail.

Cursor AI: Intelligent Documentation Assistant :

While Dokka excels at structured API documentation, Cursor AI enhances our documentation with contextual understanding and natural language explanations.

We will use Cursor AI for:

- **Contextual Code Explanation**: Cursor AI analyzes code patterns and provides explanations that go beyond simple descriptions, offering insights into why certain decisions were made.
- **Gap Detection**: The AI identifies undocumented or poorly documented areas of the codebase, ensuring comprehensive coverage.
- **Natural Language Documentation**: Complex algorithms and business logic are explained in clear, readable prose that complements the technical specifications.
- **Example Generation**: Cursor AI creates relevant usage examples that demonstrate how to properly implement various components. Documentation Quality Enhancement: The AI reviews existing documentation and suggests improvements for clarity, completeness, and consistency.

2.9 My Ideas for the app

The ideas below are mine for the Android Client and will be considered after the main idea list is done.

2.9.1 Adding Crashilytis

- After releasing the Mifos Android Client app on the Google Play Store, it is essential to continuously monitor its performance to provide field officers with a seamless and reliable experience. Despite thorough testing, unforeseen bugs and crashes may still occur, sometimes in scenarios we might not have anticipated. This is where Firebase Crashlytics plays a crucial role.

- Crashlytics is a real-time crash reporting tool that helps track and diagnose crashes in production. It provides detailed insights into failures, including the affected screen, root cause, and number of impacted users. By integrating Crashlytics, we can prioritize and resolve critical issues efficiently, ensuring minimal disruptions for field officers. This enhances the stability and reliability of the Mifos Android Client app, delivering a seamless and efficient experience for field officers.

2.9.2 Detecting and Preventing Memory Leaks with LeakCanary

- One of the common issues in long-running Android applications is memory leaks, which can lead to excessive memory usage, app slowdowns, and even crashes. Detecting such issues manually can be challenging, which is where LeakCanary comes in. It is an automated memory leak detection tool that simplifies identifying and resolving memory leaks in the app.

- By integrating LeakCanary into the demo app, developers can easily monitor and identify memory leaks as they occur. This allows developers to detect and fix leaks before they impact performance in production.

2.9.3 Monitoring API Calls with Chucker

- Debugging API requests and responses can be challenging, especially when tracking data issues or failed requests. Chucker addresses this by providing an in-app HTTP inspector that allows user to monitor and analyze network activity directly on the device. With Chucker, field officers can easily view requested API calls, inspect server responses, and identify issues like slow responses or incorrect data, all without needing Android Studio.

2.9.4 Realtime Firebase notification

- In the Mifos Mobile app, when a user requests a loan, it is directed to the Android client, where a field officer can approve or reject it. However, the current issue is that field officers do not receive any notifications when a loan request is made. As a result, they have to manually check the app to see if a loan has been requested, which can cause delays in taking action.

- Integrating Firebase Cloud Messaging (FCM) can solve this issue by providing real-time notifications. With Firebase notifications, field officers will instantly receive alerts whenever a loan request is made, enabling them to take timely action—whether it's approval or rejection—without needing to check the app constantly. This ensures a more efficient and responsive loan approval process.

3. Contributions to Mifos

I have been consistently contributing to Mifos for the past year. Below are the links to my contributions:

Android Client

1. PR #2337: Feat : Migrate hilt to koin
2. PR #2351: feat: Core/common to KMP
3. PR #2336 : [kmp branch] Fixed app crash on loan disbursement screen
4. PR #2322: Fix: Login was successful even it is not authenticated
5. PR #2277: KMP dependencies setup
6. PR #2316: Efix: notes not showing in noteScreen
7. PR #2285: refactor: migrate [core.model] to KMP

8. PR #2315: Fix: failed to fetch client template on create new client screen
9. PR #2313: refactor: [database.group] dbflow to room

- **Open Pull Requests**

1. PR #2086: feat: Migrate [core/database] to KMP

So far, I have **more than 80 Merged Pull Requests** in Android Client alone. You can find all of my merged PRs here, and I am committed to continuing this contribution to ensure the Android client remains bug-free

Mobile Wallet

1. PR #1779: fix: history UI issue/a>
2. PR #1778: refactor: Faq UI redesign
3. PR #1771 : refactor: MW-143 Home UI redesign
4. PR #1767: Profile UI redesign

Mifos Mobile

1. PR #2808: refactor: [feature/guarantor] migrate to CMP/a>
2. PR #2620: refactor: qrCodeImport to compose
3. PR #2624 : Fixed scrolling issue in Registration Screen
4. PR #2616: Missing TopAppBar navigate back implementation in AddBeneficiaryScreen
3. PR #2591 : fix: Migrate locationsFragment to compose
4. PR #2611: fix: Scrolling Disabled in TransferProcessFragment
4. PR #2544: fix: Improved security of password

You can see all of my PR's in Mifos Mobile here

4. Week Wise Breakdown

4.1 Community Bonding Period (1 May - 26 May)

Week 1

- Connect with the mentor and the developers.
- Introduce myself to the community and take their inputs as well
- Identify any changes that need to be made to the project and discuss them. It would be better to discuss modifications, additions, and amendments as early as possible.

Week 2

- Become familiar with the Android Client codebase.
- Implement features that are currently missing from the Android client by taking examples from other openMF android applications.
- Go through the Official documentation of Fineract client KMP SDK and related documentation.

Week 3

- With the mentor, discuss the workings of the existing application. It is important to discuss how we are going to migrate the modules
- I will start by migrating remaining core module that is left

4.2 Phase 1 (27 May - 12 July)

Week 4

- I will not only perform the migration to KMP but also update the Android client to the latest dependencies , implementing flows, and incorporating all of this within the multi-module structure.

Week 5

- During this week, I will begin replacing the existing network layer with the Fineract KMP SDK.
- The SDK is still under development at the time of writing this proposal. If it is not completed by this time, I will first contribute to the SDK's development and then proceed with its integration.

Week 6

- This week, I will ensure that the migration of all core modules is completed.

Week 7

- This week, I will begin with the `feature.auth` module, as it is the entry point of the application and should be migrated first to Compose Multiplatform.
- The ViewModels will continue to use RxJava for now. I will gradually migrate them to Kotlin Coroutines and Flow as I work on migrating individual feature modules

Week 8

- I will continue working on the CMP migration, as there are a total of 19 feature modules that need to be migrated.
- Alongside the migration, I will keep improving the codebase wherever possible to enhance readability, maintainability, and performance.

Week 9

- *Buffer* for any pending tasks
- Prepare a report for evaluation
- Discuss brief plan for Phase 2 with the mentors

4.3 Phase 2 (12 July - 26 Aug)

Week 10

- This week, I will focus on writting unit tests.
- Then, begin developing unit tests for use cases and repositories with 100% coverage.

Week 11

- I will start working on UI tests for all of the compose screens

Week 12

- Once I complete all the necessary tests, I will update the CI/CD pipeline to incorporate the recent changes.
- Additionally, I will create the `android_client_release.yml` file to include the Play Store release GitHub Actions pipeline, ensuring smooth and automated deployment.

Week 13

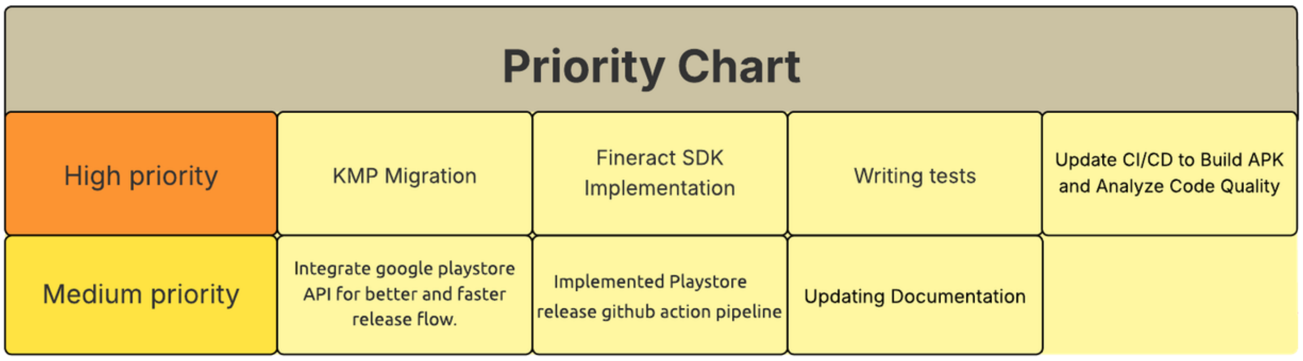
- This week, I will initiate the integration of the Google Play Store API into the Android client to enhance and simplify the release flow process.
- After implementing these updates, I will thoroughly revise the documentation to ensure it reflects the changes accurately. This will provide clear guidance for anyone interacting with the codebase, ensuring they have all the necessary information to effectively utilize the new features and understand the updates.

Week 14

- This week, I will focus on fixing the existing bugs in the app.

Week 15 and Rest of Phase 2

- If permitted by the mentors, I would like to implement some of my own ideas.
- Allocate buffer time to address any remaining tasks.
- Prepare a report for the evaluation.



Priority Chart

High priority	KMP Migration	Fineract SDK Implementation	Writing tests	Update CI/CD to Build APK and Analyze Code Quality
Medium priority	Integrate google playstore API for better and faster release flow.	Implemented Playstore release github action pipeline	Updating Documentation	

4.4 Post Phase 2 (After Aug 26)

- I will discuss future plans for the Android client with the mentors to align on upcoming goals.
- If permitted, I will implement my own ideas to enhance the project.
- I will continue contributing to the Android client to ensure its compatibility with future standards and technologies.

5. Why am I the right person ?

I have been working with Android application development for over a year, gaining valuable experience by contributing to Mifos's open-source Android apps and building my own projects. During an internship with Mifos from June 2024 to August 2024, I had the opportunity to collaborate with fellow interns on migrating XML screens to Jetpack Compose. Since then, I've continued contributing to Mifos and have gradually built a solid understanding of Jetpack Compose, Kotlin, and modern Android development practices.

6. Current area of study

I am a second-year Computer Science & Engineering student studying at Southeast University.

7. Contact Information

Name: Pronay Sarker

Email: pronaycoding@gmail.com

GitHub: https://github.com/itsPronay

LinkedIn: https://www.linkedin.com/in/itsPronay/

Slack Account: Pronay Sarker

Mobile Number: +8801639282567(whatsapp) / +8801309958167

Time Zone: GMT+6

8. Career Goals

As an Android developer, my primary goal is to continuously expand my skills and knowledge in the ever-evolving field of Android app development, focusing on emerging technologies and programming languages. By staying up-to-date, I aim to create high-performance applications that offer users exceptional experiences, solve real-world problems, and contribute to their productivity and ease of life. I also have a strong interest in mastering Kotlin Multiplatform (KMP) to build cross-platform applications and streamline development processes. Additionally, I am passionate about contributing to open-source projects, as I believe open-source is a powerful avenue to hone my skills and give back to the community.

At some point, I plan to pursue a Master's or PhD in Machine Learning, as I believe this field holds immense potential to revolutionize technology and create smarter, more efficient solutions. I am eager to explore this area further and integrate machine learning into my work to enhance the capabilities of the technology I develop.

9. Project (Source Code I have written)

1. Napify : *Playstore*

- Napify is an open-source relaxation app that helps the user to fall asleep in a noisy environment.

- It helps user to enhance productivity and relaxation by offering immersive sounds for focus and sleep

2. Charoom : *Source Code*

- ChaRoom is an application where users can create custom rooms for chatting and text among themselves.
- Used Firebase Realtime Database for real-time messaging.

10. Slack Channel

Yes, I have already visited Slack channel and my nickname is Pronay sarker

11. Other Open Source Contributions

I have been contributing to Open Souce for quite some time and here are some of my contributions:

1. Kubernetes

- Kubernetes (K8s) is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications.
- Here is my contribution to Kubernetes.

2. blanket

- Blanket is a gnome application that helps to focus and improve productivity.
- My contribution to Blanket here

12. Experience with Angular/Java/Spring/Hibernate/MySQL/Android

I have experience with Android development and have learned Spring Boot and MySQL through my college coursework. However, I do not have prior experience with Angular. But, I am always open to learning new technologies if required.

13. Interaction with mentor

Yes, I have had the opportunity to interact with Mr. Rajan Maurya

14. What motivates to work with Mifos

I am excited about the opportunity to contribute to Mifos through Google Summer of Code (GSoC) because of its mission to promote financial inclusion using open-source technology. Mifos has made a significant impact by providing accessible financial solutions to underserved communities, and I am eager to be part of this meaningful initiative.

Being involved in this project would allow me to apply and expand my Android development skills while working on technology that directly improves lives. The chance to develop solutions that empower individuals and small businesses aligns with my passion for using technology to drive positive change.

Additionally, Mifos has a strong and collaborative open-source community, where developers actively support and learn from each other. Contributing to this ecosystem would not only enhance my technical expertise but also allow me to exchange ideas with experienced developers and gain valuable mentorship.

This opportunity combines three aspects that I deeply value—working on impactful technology, contributing to an open-source mission, and collaborating with a passionate global community. I am eager to bring my skills to Mifos and grow both as a developer.

15. Previous Participation in GSoC

I have not previously participated in Google Summer of Code (GSoC). This will be my first time applying.

16. Application to multiple orgs

Mifos is the only organization I am applying to for Google Summer of Code.