✦ Member-only story

Migrating an Existing Xcode Project to a New Kotlin Multiplatform Mobile App

Going further with Kotlin Multiplatform

Mats Bauer · Follow
Published in Better Programming
6 min read · Nov 12, 2020

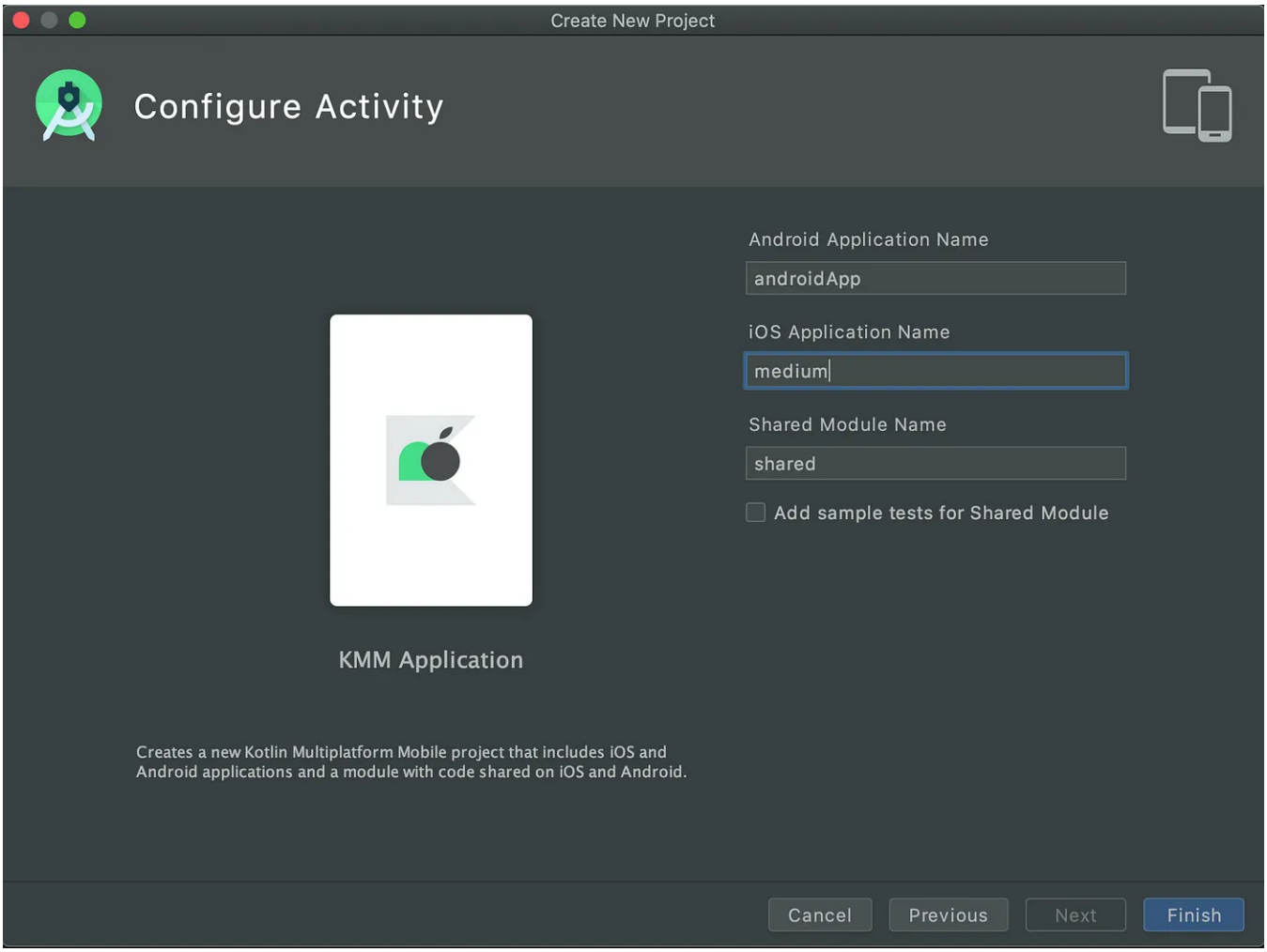
Listen Share More

Image by Ketut Subiyanto on Pexels

Manually setting up a Kotlin Multiplatform Mobile (KMM) project from scratch can be tedious. We want to migrate our Xcode project called `medium` to a new KMM application.

Getting Started

For this example, we'll use a blank KMM project with a twist. If you don't know how to create a KMM project, check out my previous article first. The twist: In the creation process this time around, we'll name the iOS application precisely the same as our Xcode project. Doing so removes the need to change the path and code-project variables in Gradle and property files.



Changing the iOS application name for our KMM project

Our created KMM project now contains three components:

- An Android app

- The common code (in Kotlin)
- An iOS app

In the next step, we want to replace the standard KMM iOS app with our existing app. Our existing app that's being used is called `medium` and can be a newly created blank project or your latest masterpiece. In this demo, `medium` is a newly created project.

Replacing the iOS App

First, we go into the directory of our KMM application in Finder. You can see three main folders: `androidApp`, `shared` and `medium`. Looking inside the folder `medium`, we see a typical Xcode project layout, containing the `.xcodeproj` file and all of the source and test files.

Action: Remove everything inside the `medium` folder, and drop all of your existing app's folders inside. From now on, we'll open the iOS app from here.

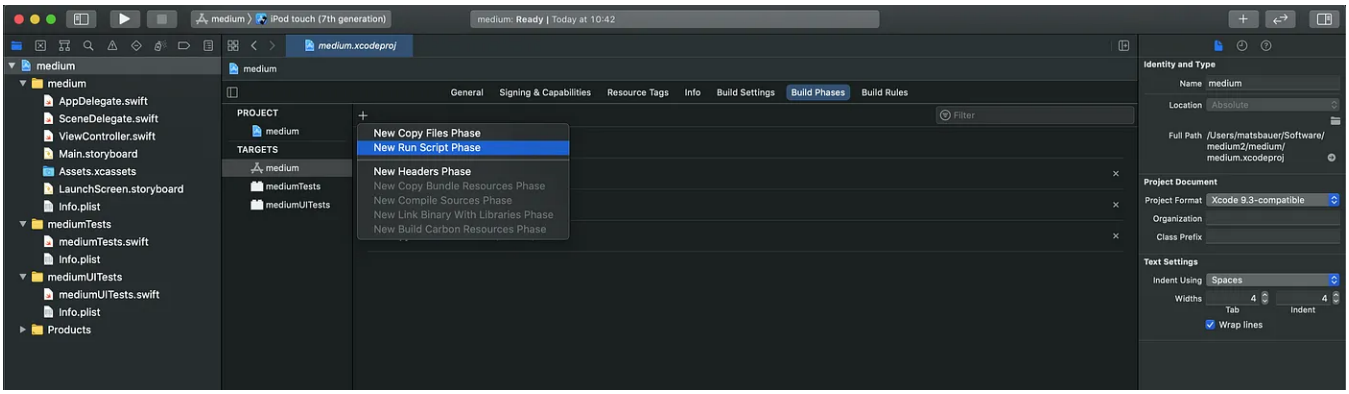
Linking Our iOS App With the KMM-Shared Components

For our iOS application to work with the shared KMM components, two things need to happen:

- The KMM project needs to be built for our iOS app.
- The iOS app needs to find and reference the built shared files.

1. Build the KMM project for Xcode

In your code project, go to your root project and open Build Phases. By clicking the +, create a new Run Script Phase.



The Run Script lets you run any shell script when building the project. This script enables us to build the KMM application for iOS without Android Studio or running Gradle in the terminal.

Into the Shell script field, paste:

```
#!/bin/sh

PARENTDIR="$(dirname "$SRCROOT/")"

cd "${PARENTDIR}"

./gradlew :shared:packForXCode -
PXCODE_CONFIGURATION=${CONFIGURATION}

UNIVERSAL_OUTPUTFOLDER="${APPNAME}/framework/${CONFIGURATION}-
universal"

FRAMEWORK_NAME="shared"

BASEPATH="${UNIVERSAL_OUTPUTFOLDER}/${FRAMEWORK_NAME}.framework"

ARM64PATH="${PARENTDIR}/${FRAMEWORK_NAME}/build/bin/iosArm64/debugFr
amework/${FRAMEWORK_NAME}.framework"

X64PATH="${PARENTDIR}/${FRAMEWORK_NAME}/build/bin/iosX64/debugFramew
ork/${FRAMEWORK_NAME}.framework"

mkdir -p "${UNIVERSAL_OUTPUTFOLDER}"

#copy headers, modules & info.plist of ARM64/X64 build into our
framework
cp -R "${ARM64PATH}" "${UNIVERSAL_OUTPUTFOLDER}/"

if [ ! -d "$ARM64PATH" ]; then
echo "error: Please on an iOS Device to create ARM64 build files"
fi

if [ ! -d "$X64PATH" ]; then
echo "error: Please run on a Simulator to create X64 build files"
fi

ARM64PATHAPPEND="${ARM64PATH}/${FRAMEWORK_NAME}"
X64PATHAPPEND="${X64PATH}/${FRAMEWORK_NAME}"
```

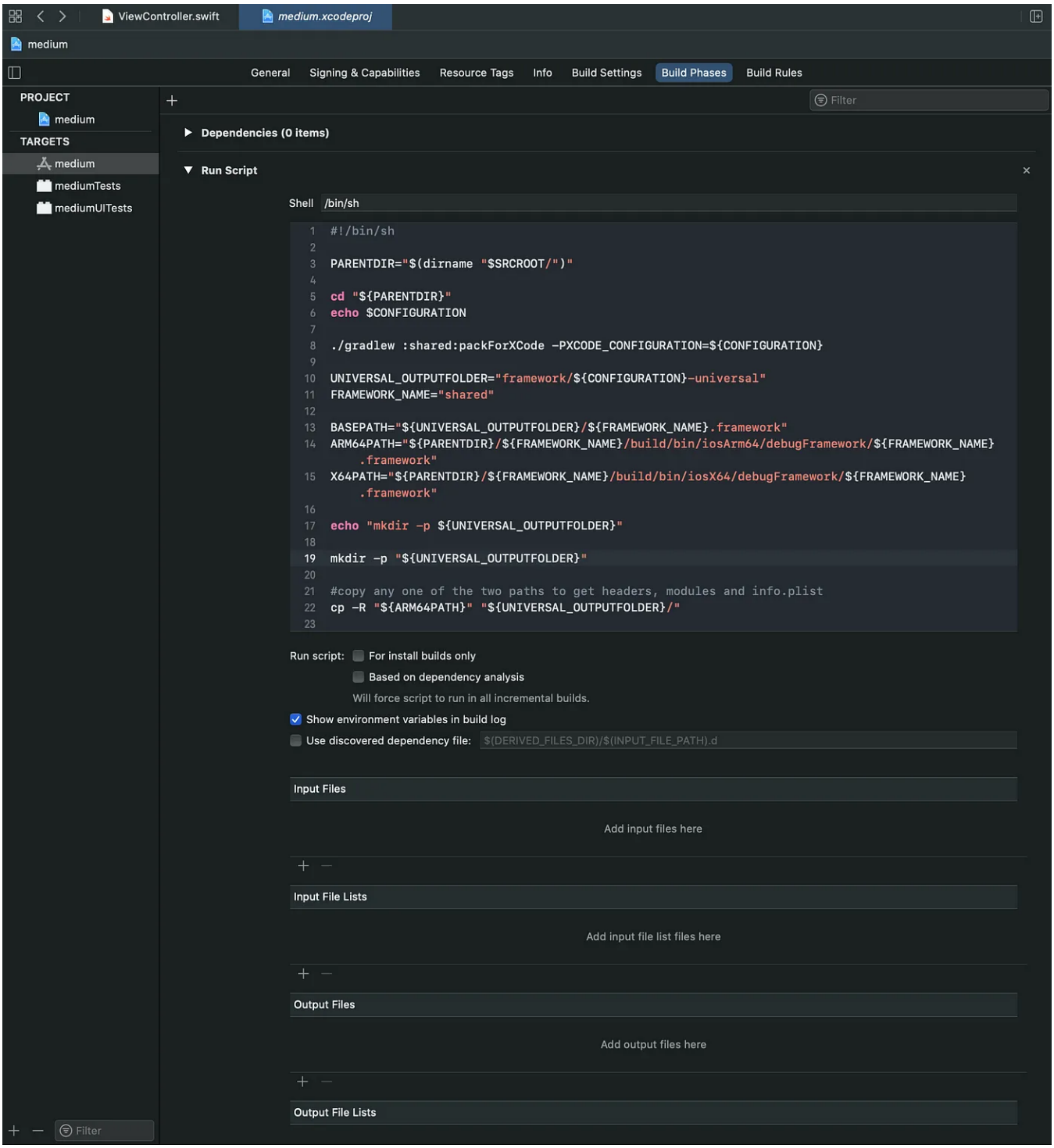
```
FULLPATH="${PARENTDIR}/${BASEPATH}"

lipo -create "${ARM64PATHAPPEND}" "${X64PATHAPPEND}" -output
"${FULLPATH}/shared"
```

I'll go into the functionality of the script later. The main execution is the line `./gradlew :shared:packForXcode`. This line of code runs the packing agent located in `build.gradle.kts`, which is in the shared folder. The code block looks like this:

```
val packForXcode by tasks.creating(Sync::class) {
    group = "build"
    val mode = System.getenv("CONFIGURATION") ?: "DEBUG"
    val sdkName = System.getenv("SDK_NAME") ?: "iphonesimulator"
    val targetName = "ios" + if (sdkName.startsWith("iphoneos"))
"Arm64" else "X64"
    val framework = kotlin.targets.getByName<KotlinNativeTarget>
(targetName).binaries.getFramework(mode)
    inputs.property("mode", mode)
    dependsOn(framework.linkTask)
    val targetDir = File(buildDir, "xcode-frameworks")
    from({ framework.outputDirectory })
    into(targetDir)
}
tasks.getByName("build").dependsOn(packForXcode)
```

I won't go further into this (standard) script, except to mention that it distinguishes the build between iOS Arm64 and x64. This differentiation lets us use the KMM components on a real iPhone (using an ARM CPU) and a simulator. You can read more about CPU architecture here.

Important: Move the Run Script to the top of the Build Phases!



Move the Run Script to the top of the list

The Run Script has two processes. First: It creates the build framework for the given CPU (ARM or x64). Second: It merges the two build frameworks into a single framework that can handle both ARM and x64 CPUs; let's call this our *universal framework*.

The next step is a little messy but creates the right setup and only has to be done once. So read carefully!

1. **Run the project on any simulator**. The build will fail, saying: `Command PhraseScriptExecution failed with a nonzero exit code.` And that's perfect! We now have the simulator framework (in `shared > build > bin > iosX64 > debugFramework`).

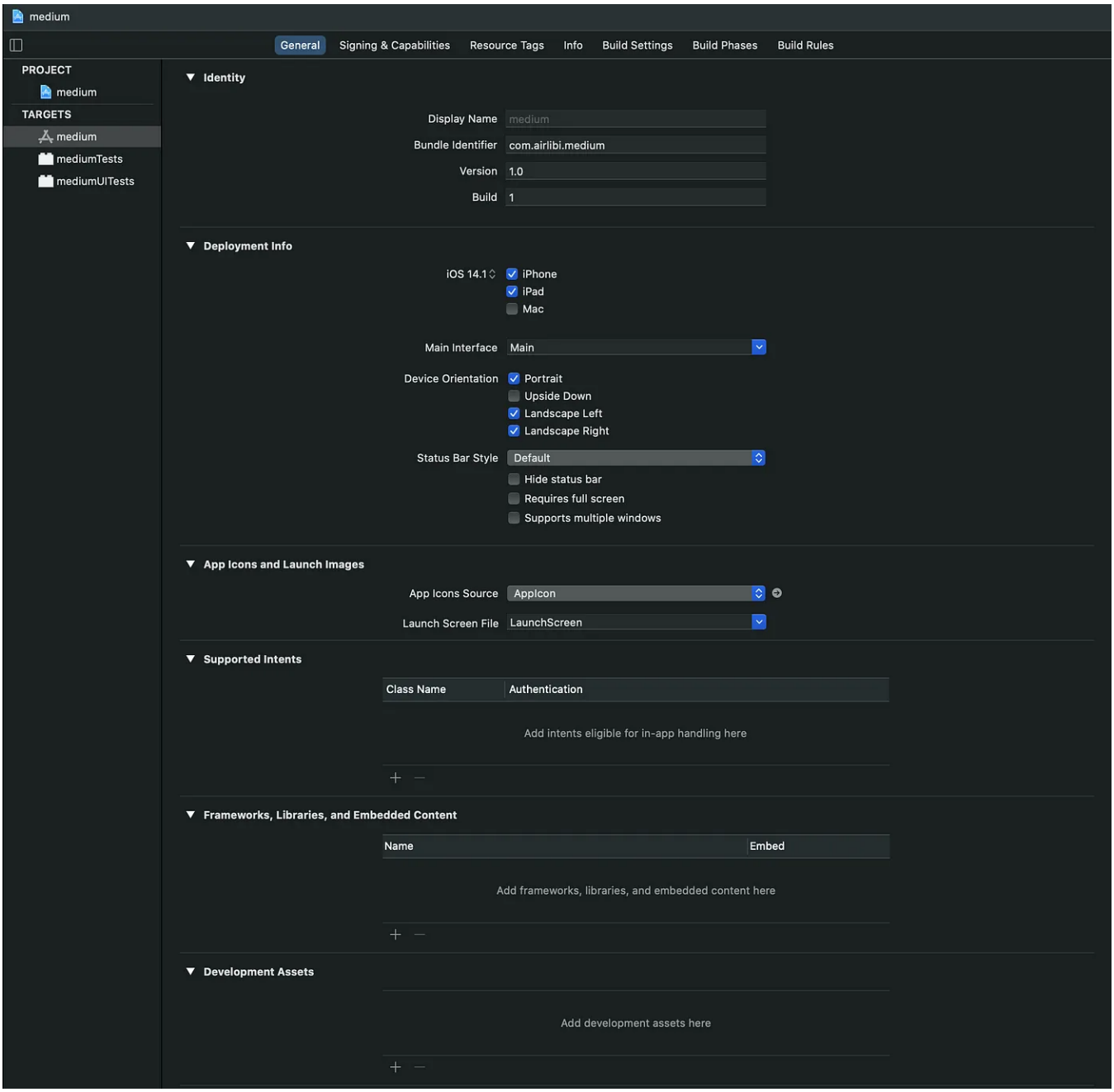
2. **Run the project on your iPhone**. The build will succeed, but your app will crash on runtime, saying: `Library not loaded: shared.framework.` And that's perfect again. We now have both the iPhone framework (in `shared > build > bin > iosArm64 > debugFramework`) and the created universal framework (in `framework > Debug-universal`). The error comes from not linking the universal framework to our project yet.

So let's get to importing the framework!

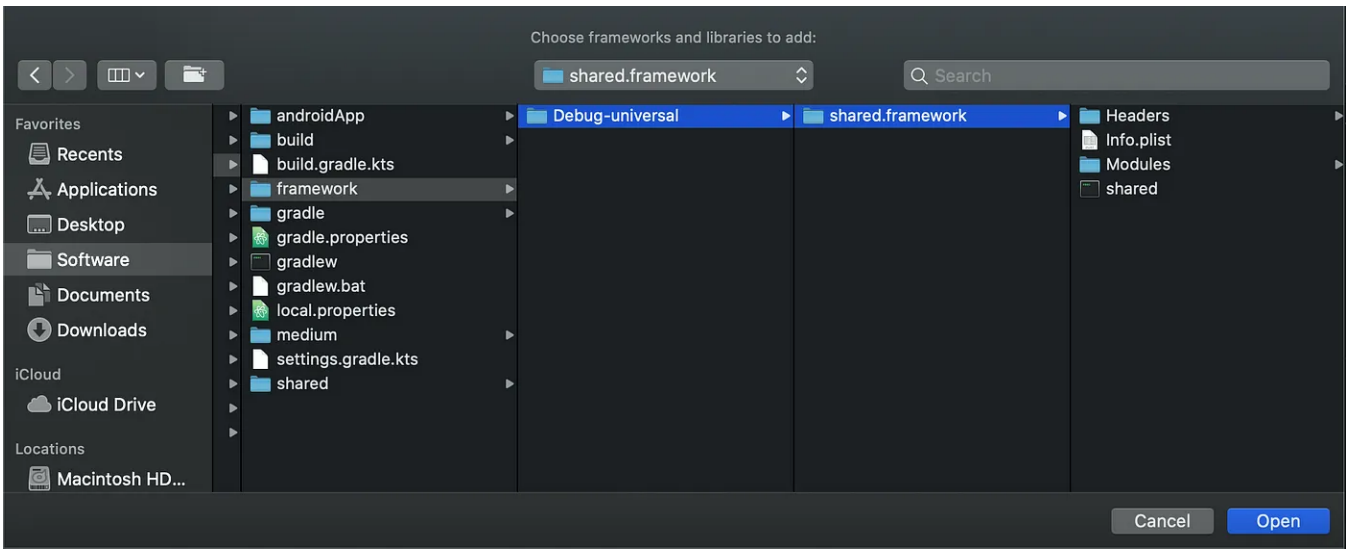
2. Importing our universal 'shared.framework'

You might read some KMM tutorials where you're supposed to import the shared framework from the build folder. But we created our universal framework for both the simulator and the iPhone, so forget that! You'll find our simulator- and iPhone-compatible framework in `framework > Debug-universal > shared.framework`. This directory has to be linked to our Xcode project now.

In Xcode, go back into the General tab. Here you see the Frameworks, Libraries, and Embedded Content section.



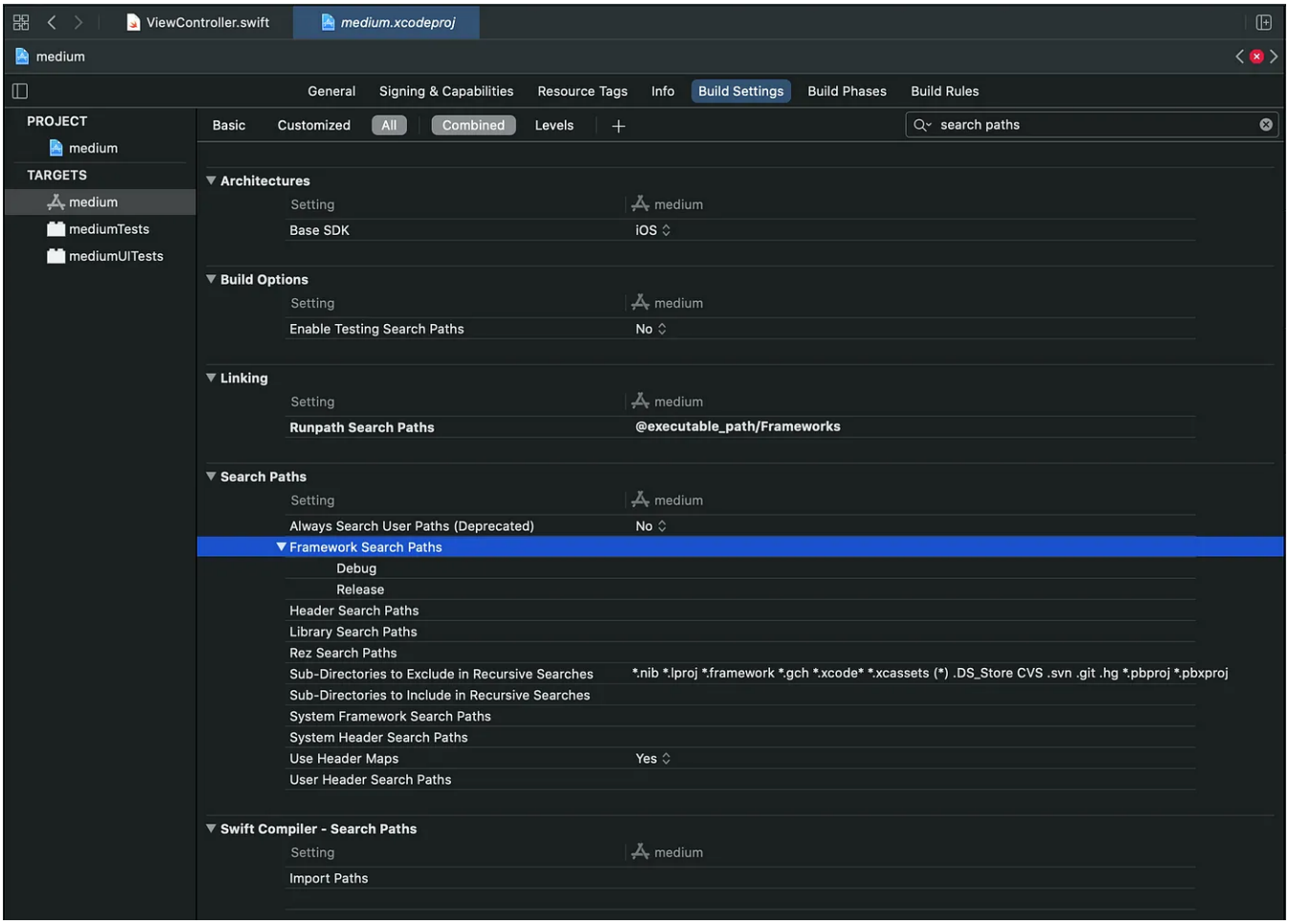
Action: Here, we include `shared.framework` from the `framework` directory:



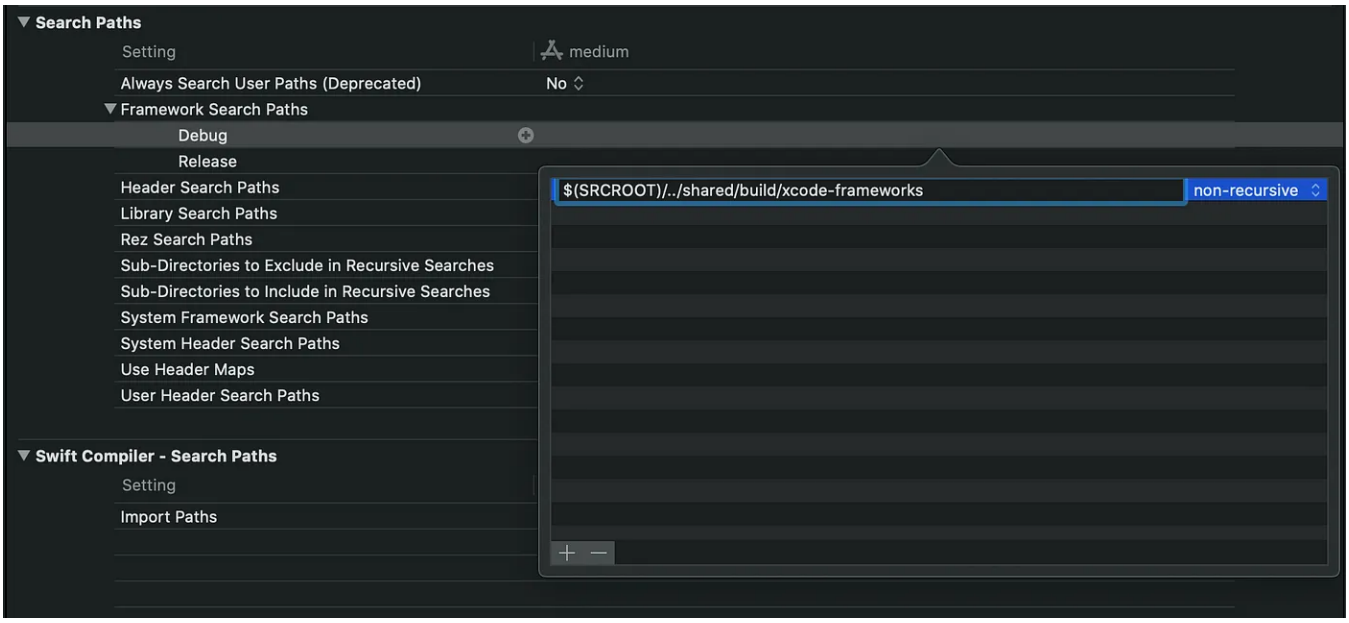
Import framework to Xcode

Before running our project now, we have to add a specific path to Framework Search Paths. This search path is essential for the project to find the Xcode framework `shared.framework`. Go to Build Settings and search for the Search Paths section.

This gives you the drop-down option for Framework Search Paths.



Action: Paste the phrase `$(SCROOT)/../.shared/build/xcode-frameworks` into both Debug and Release by double-tapping the respective field on the right, clicking on the plus in the pop-up window, and pasting.



Once that's done, we're ready to run the project anywhere, whether it's on your iPhone or a simulator.

Note: If we're running on an iPhone or a simulator, only the adjacent shared build folder will be rerun. This slims down the build process instead of always running both build types.

Congratulations!

Debug Section

```
Building for iOS Simulator, but the linked and embedded framework
'shared.framework' was built for iOS.
```

This sometimes happens when we fiddle around with the build files. To fix this, remove the `shared.framework` from the General tab (Frameworks, Libraries, and Embedded Content), build onto the iOS simulator again, and then readd the `shared.framework` as in step 2.

```
The operation couldn't be completed. Unable to locate a Java
Runtime.
```

This is due to a missing Java runtime installation. To fix this, try installing Java JDK8. If the error persists, a restart of your computer can help.

If you experience any other bugs or errors, feel free to ping me or comment so that we can share the solutions.

Kotlin Programming IOS Android Mobile

Follow

Written by Mats Bauer

448 Followers · Writer for Better Programming

Technical Product Executive at Herewith Inc.

More from Mats Bauer and Better Programming

Mats Bauer

How to run SSH terminal commands from iPhone using Apple Shortcuts SSH

Let's have a look at how to run a terminal command from your iPhone using Apple Shortcuts. First, we need Shortcuts installed and be...

✦ · 2 min read · Dec 16, 2020

19

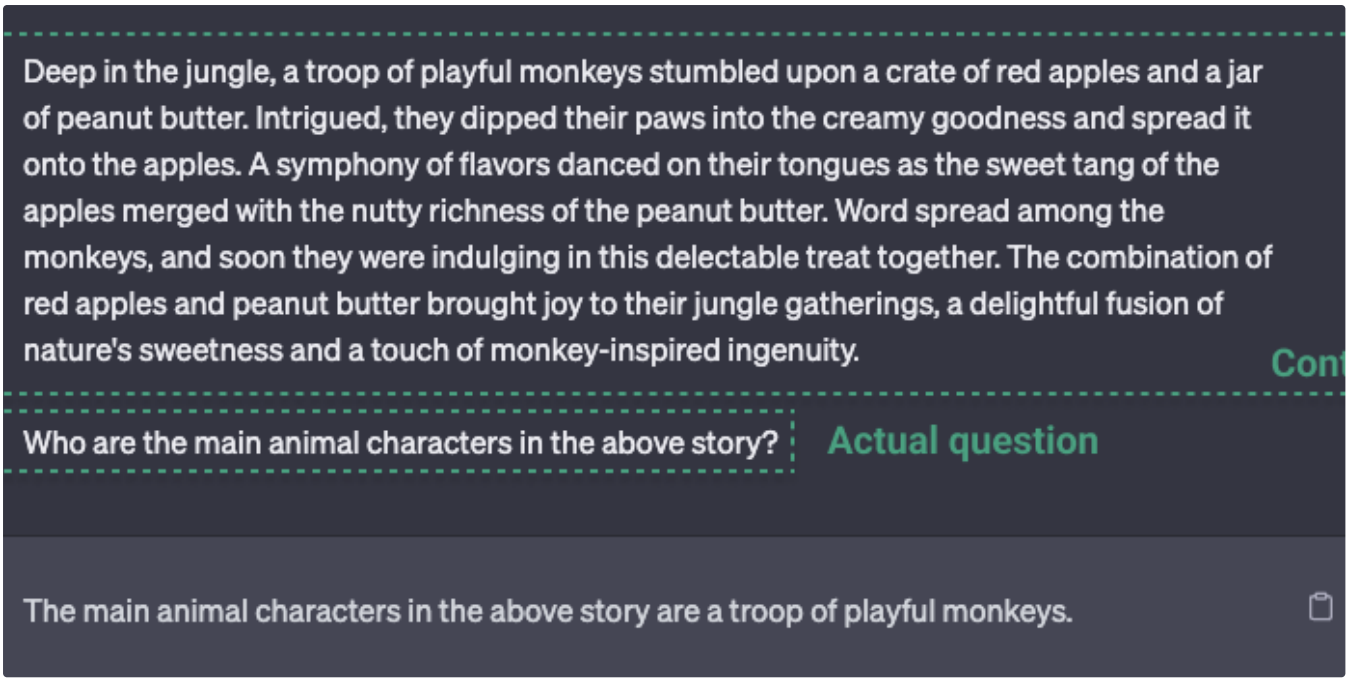
Benoit Ruiz in Better Programming

Advice From a Software Engineer With 8 Years of Experience

Practical tips for those who want to advance in their careers

22 min read · Mar 20, 2023

14.1K 264



Sami Maameri in Better Programming

Building a Multi-document Reader and Chatbot With LangChain and ChatGPT

The best part? The chatbot will remember your chat history

17 min read · May 20, 2023

1.5K 14

Mats Bauer in Next Level Swift

Auto detecting links in a TextView in 60 seconds

I will show you just how easy it is to auto-detect a link in a TextView!

✨ · 3 min read · Mar 16, 2021

56

See all from Mats Bauer

See all from Better Programming

Recommended from Medium

Kaushal Vasava

Expect and Actual functions in Kotlin for Kotlin Multi Platform

I will show you what is these functions? Why we use it? and How to use it?

4 min read · Oct 6, 2023

20

Kamlesh Lakhani

multiplatform-settings | KMM

This is a Kotlin library for Multiplatform apps, so that common code can persist key-value data.

1 min read · Nov 27, 2023

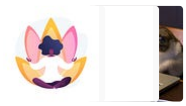
Lists

General Coding Knowledge

20 stories · 931 saves

Coding & Development

11 stories · 455 saves



Stories to Help You Grow as a Software Developer

19 stories · 834 saves

ChatGPT

21 stories · 474 saves

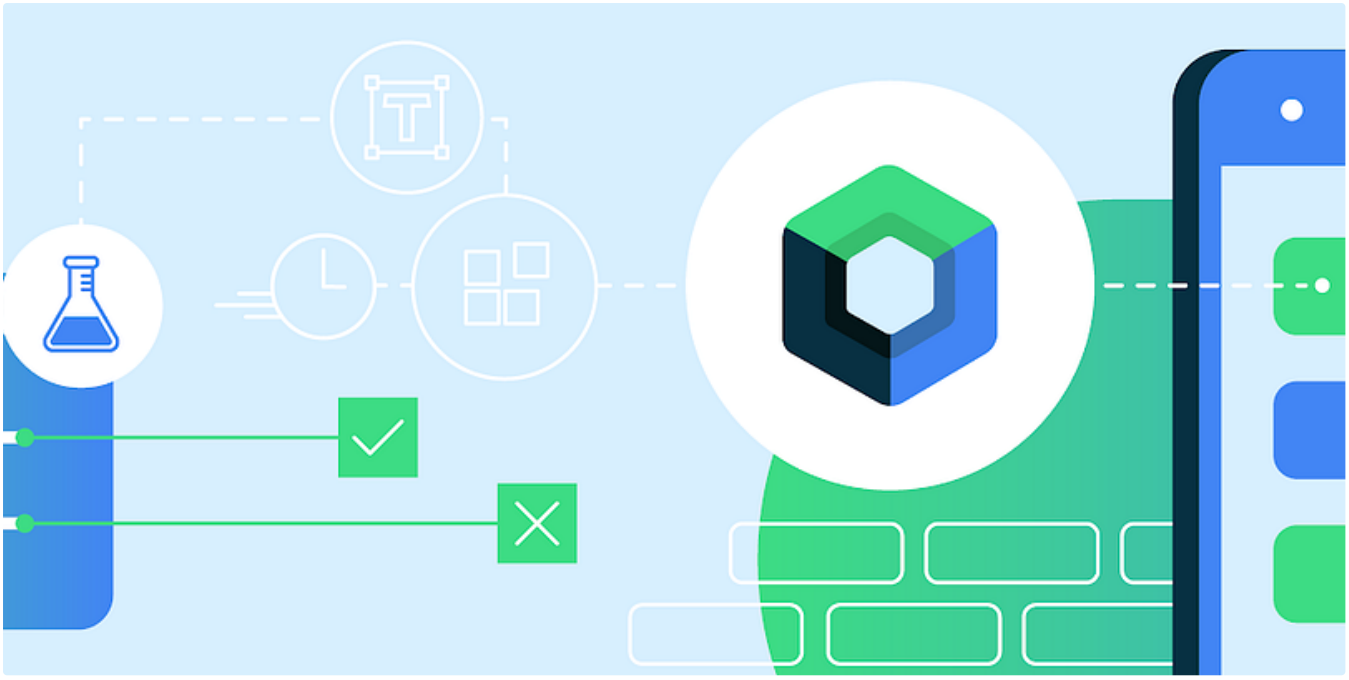
Carlos Ugaz

How to implement Firebase Firestore in Kotlin Multiplatform Mobile (KMM) with Compose-Multiplatform

A simple solution to use shared Kotlin code to develop an app for iOS and Android simultaneously, using compose-multiplatform.

10 min read · Sep 22, 2023

304



Jorge Luis Castro Medina

Modern Android Development in 2024

Set of good practices, guides, and tools to create modern Android apps in 2024.

18 min read · Feb 11, 2024

1.1K 5

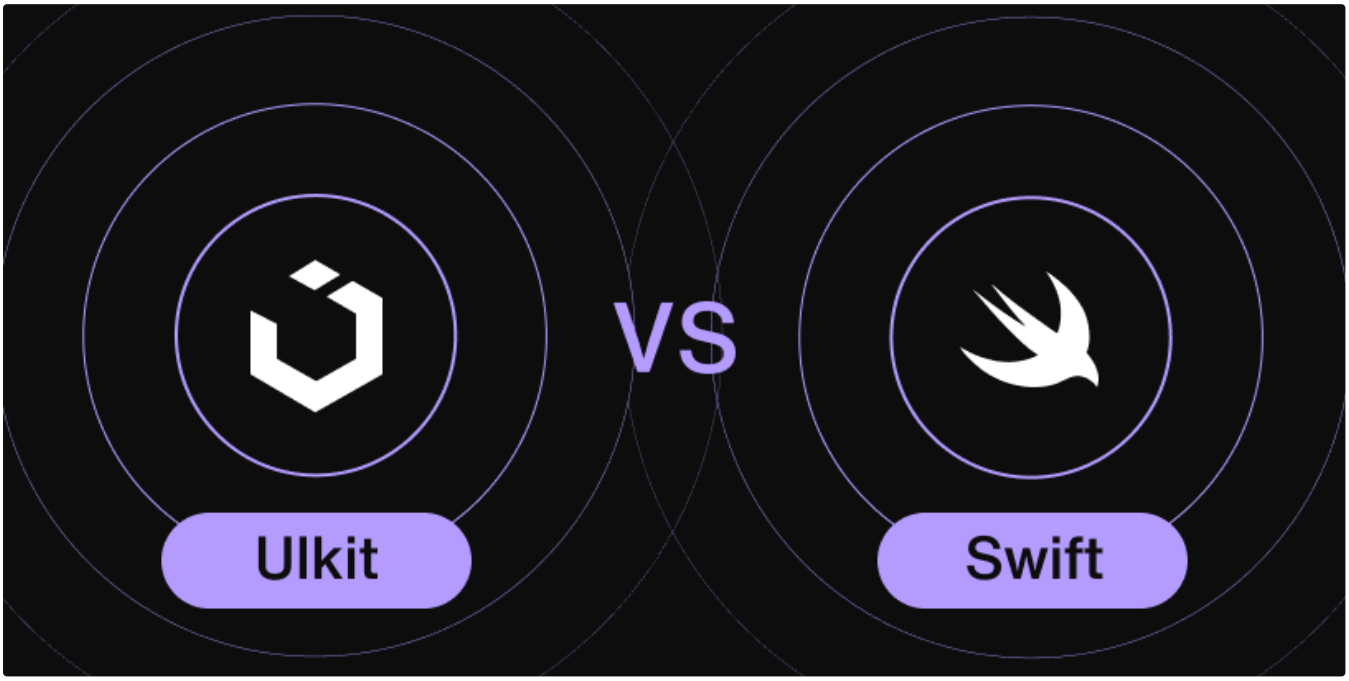
Ryan W

Build Apps for iOS, Android, and Desktop With Compose Multiplatform

- JetBrains KMP Webinar 3 of 4

6 min read · Dec 1, 2023

89 1



Sendbird in Dev Genius

SwiftUI vs. UIKit: What is the best choice for building an iOS user interface in 2024?

Help your iOS app development team finally decide between SwiftUI vs. UIKit.

10 min read · Feb 9, 2024

91 6