Create a Swift 5 Static Library: Part 2

Let's make our static library universal

Rostyslav Druzhchenko Follow
Nov 15 · 8 min read ★

Photo by Christopher Gower on Unsplash

The main idea this article considers is how to make a static

library *universal*, meaning containing binary code that works across a simulator and a real device.

To know how to make a static library on Swift 5 from scratch , check out the first part of this tutorial.

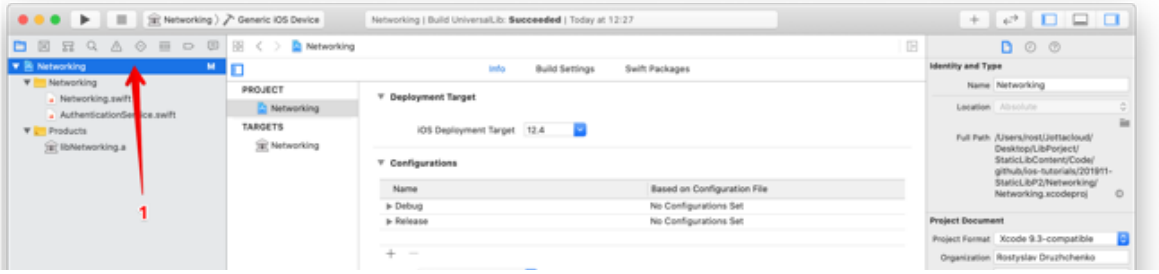
. . .

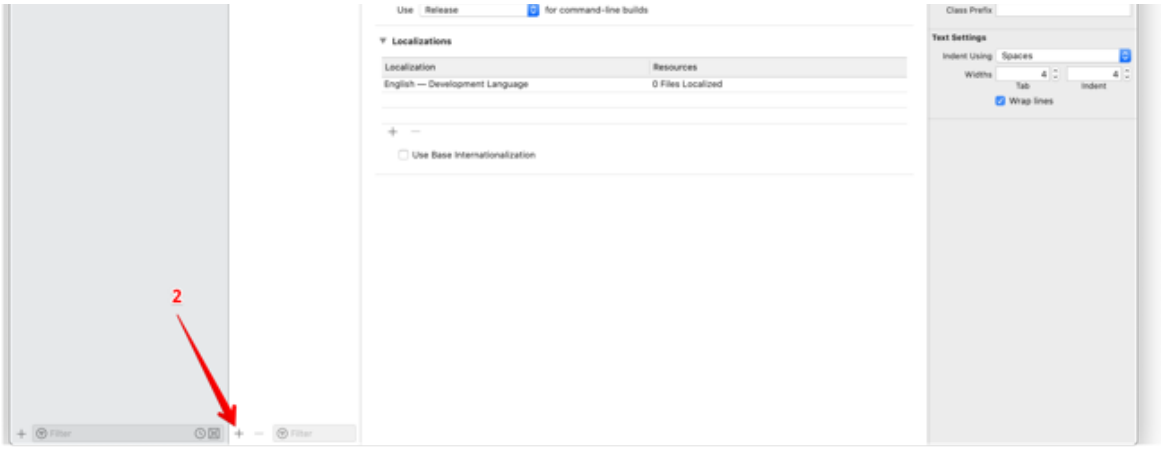
Make the Library Universal

To follow this article, you may use the final project from the previous tutorial, or you can download the ready to use starter project from GitHub. There are two projects there. Open the one named Networking, which is the project we'll use for a static library in this tutorial.

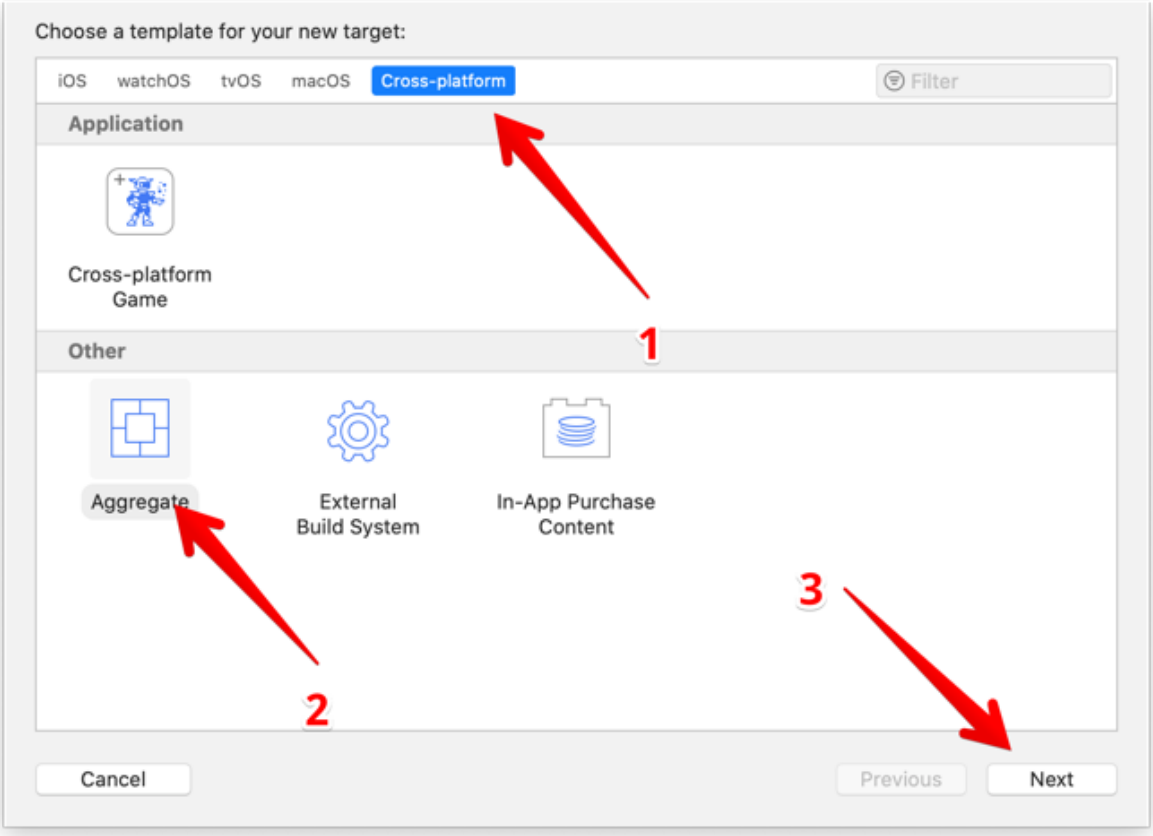
Add a new target: aggregator

First thing, open the Networking project, and click on the project name in the project navigator. Then click on the grey + in the bottom-left corner of the target list to add a new target that'll give us ability aggregate code for both architectures in one binary file.



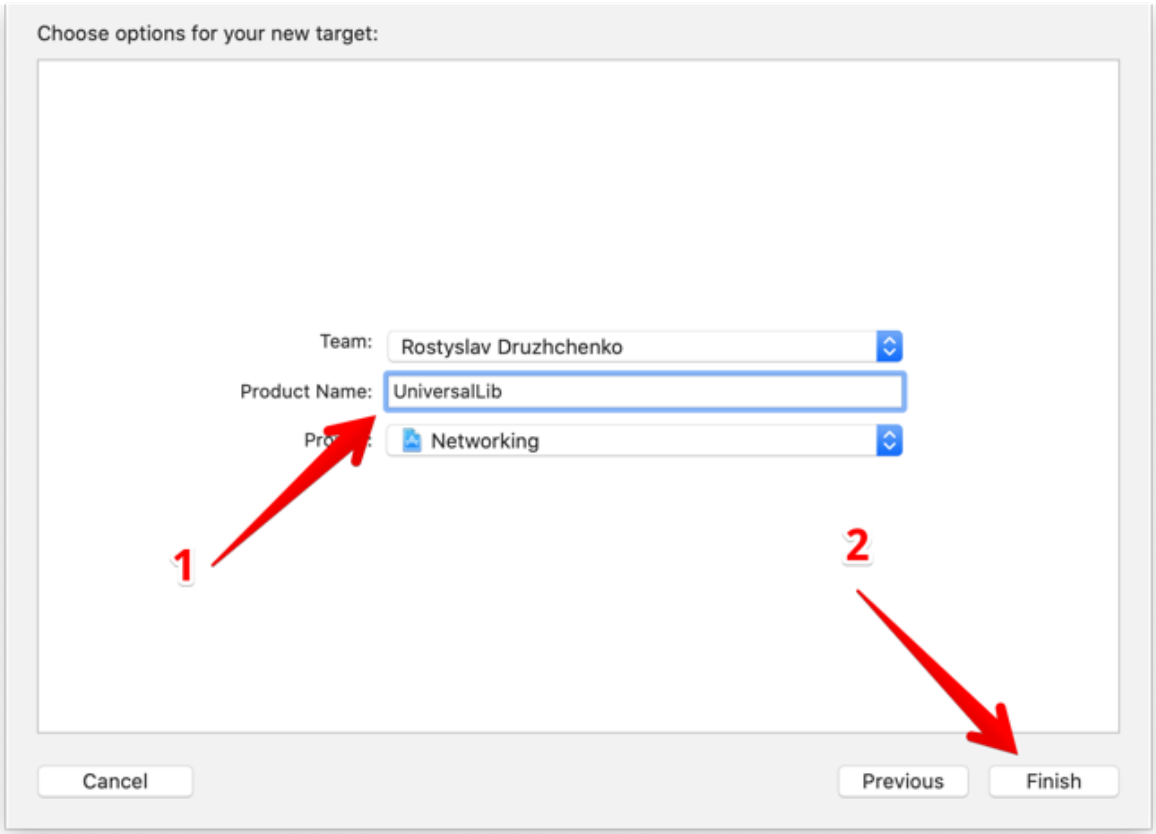


You'll see a new pop-up window where you need to choose the Cross-platform tab. Then press Aggregate. And, finally, press Next.



Name the new target whatever you want. In this tutorial, we

name it UniversalLib, then press Finish.



It'll create a new target in your project that can be used to build the universal library.

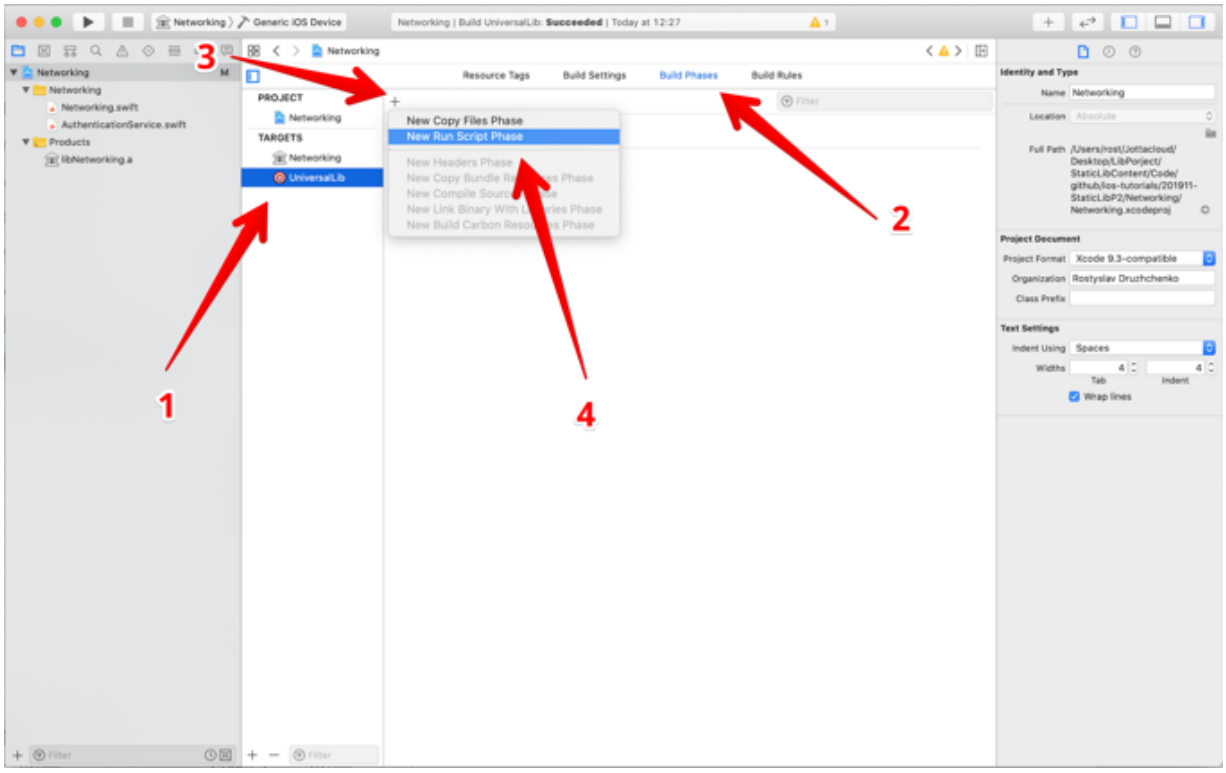
Create a run script

Select the created target in the target navigator (1).

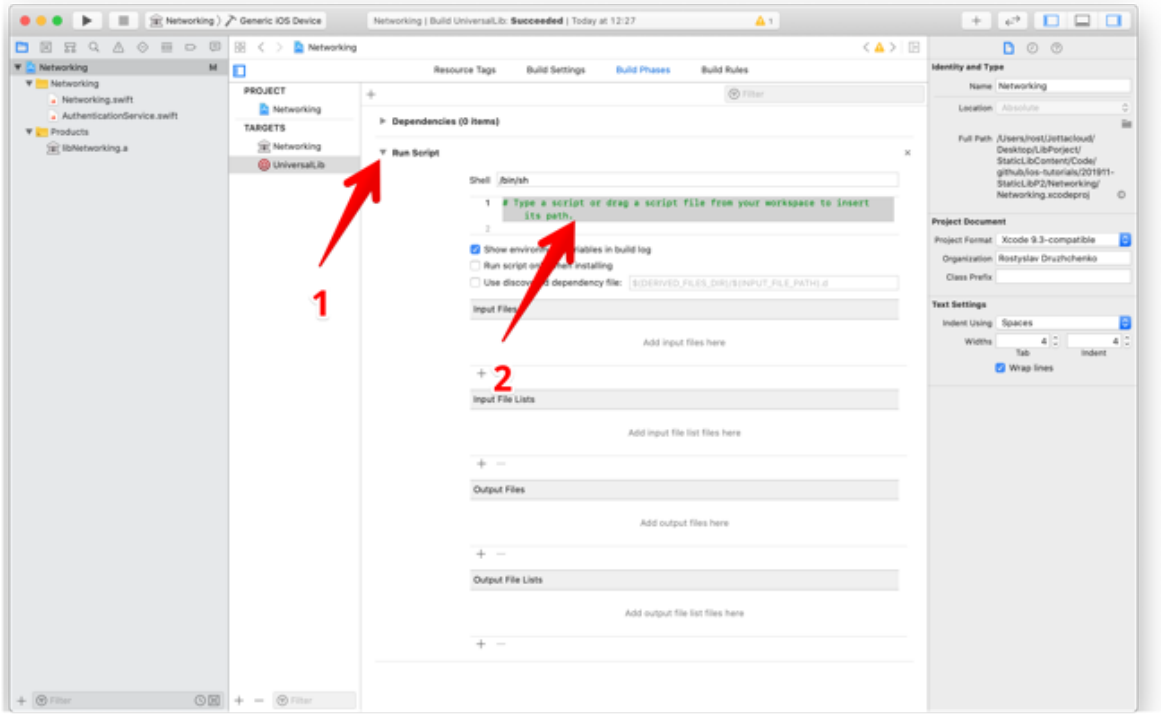
Then select Build Phases (2).

Then press the + button — as shown on screenshot (3).

And choose New Run Script Phase (4).



In the Run Script section that appears, press the grey triangle button (1). Then in the text view that appears (2), select all the text, and delete it.



Copy and paste this code to the run the script's text view:

```
# Created by: Rostyslav Druzhchenko
#

# 1: Declare variables
RESULT_DIR="libUniversal"

BUILD_DIR_SIMULATOR="Debug-iphonesimulator"
BUILD_DIR_DEVICE="Debug-iphoneos"

LIB_NAME="Networking"
LIB_BINARY_NAME="lib$LIB_NAME.a"
LIB_BINARY_NAME_SIMULATOR="lib$LIB_NAME-simulator.a"
LIB_BINARY_NAME_DEVICE="lib$LIB_NAME-device.a"

SWIFTMODULE_DIR=$LIB_NAME".swiftmodule"

# 2: BUILD
#
# Build for simulator
xcodebuild -target $LIB_NAME -configuration ${CONFIGURATION} -sdk iphones

# Build for device
xcodebuild -target $LIB_NAME ONLY_ACTIVE_ARCH=NO -configuration ${CONFIGU

# 3: OPERATE THE BINARIES
#
# Move to the build directory
cd $BUILD_DIR

# Completely delete the result of the previous build if any
# Suppress the error to avoid unnecessary logs
rm -rf $BUILD_DIR/$RESULT_DIR 2> /dev/null

# Create a new result directory
mkdir $RESULT_DIR

```

```
# Copy simulator's binary file to the result dir and rename it
cp ./$BUILD_DIR_SIMULATOR/$LIB_BINARY_NAME ./$RESULT_DIR/$LIB_BINARY_NAME

# Copy devices's binary file to the result dir and rename it
cp ./$BUILD_DIR_DEVICE/$LIB_BINARY_NAME ./$RESULT_DIR/$LIB_BINARY_NAME_DE

# Make the library "fat", means "universal"
lipo -create ./$RESULT_DIR/$LIB_BINARY_NAME_SIMULATOR ./$RESULT_DIR/$LIB_

# Delete simulator's binary file
rm ./$RESULT_DIR/$LIB_BINARY_NAME_SIMULATOR

# Delete device's binary file
rm ./$RESULT_DIR/$LIB_BINARY_NAME_DEVICE

# 4: OPERATE SIWFTMODULE
#
# Create ".siwftmodule" directory in the result directory
mkdir $RESULT_DIR/$SWIFTMODULE_DIR

```

Let’s discuss the main point of the script to figure out what it actually does.

In section #1 it declares variables for folder names. If you want to manage these names — feel free to experiment. There is a `LIB_NAME` variable that has a `Networking` value. You may change this value to whatever you want to give your library a name.

Section #2 runs xcodebuild twice, the first time for the simulator and the second time for the device. The compiled binaries will be placed in the corresponding folder, defined in section #1.

Section #3 removes old builds results, if any. Then it creates a necessary folder and copies `libNetworking.a` files to one folder and runs the `lipo` command. That is the heart of this article — `lipo` creates the final universal binary. Then it deletes all unnecessary files.

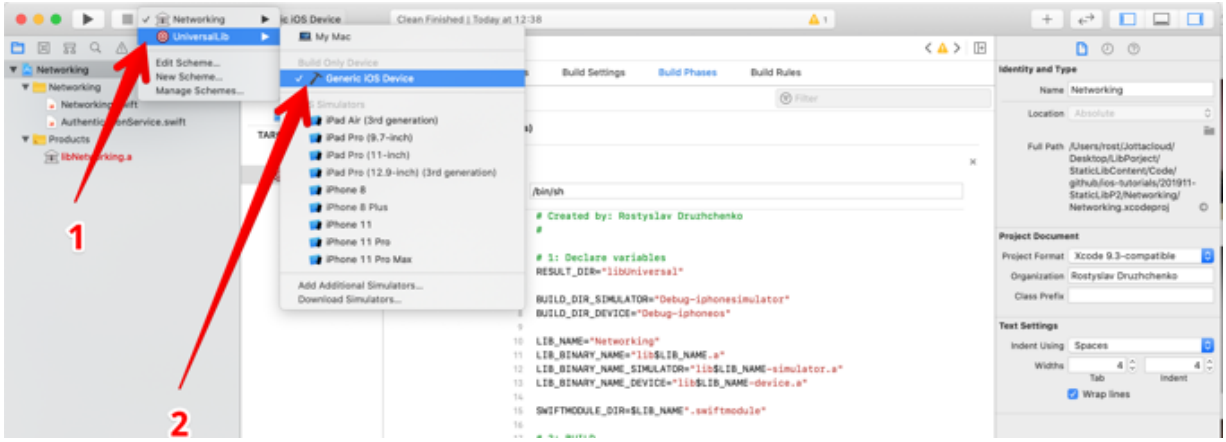
Section #4 combines `*.swiftmodule` and `*.swiftdoc` files for both architectures to one folder.

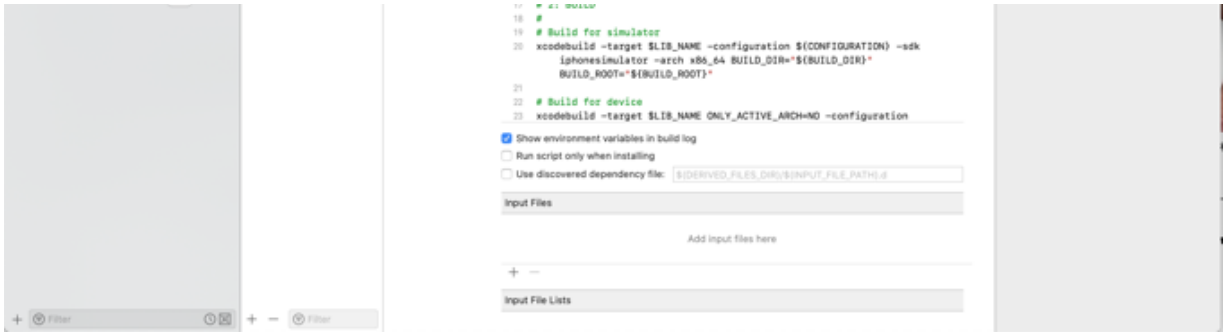
You may not change anything in this script; it should work for any library project well — the only thing that must be changed in another project is the `LIB_NAME` variable's value.

Build the library

This is an important step. In the scheme selection menu, select the UniversalLib target (1), and select `Generic iOS Device` (2).

Selecting `Generic iOS device` is mandatory. Without that, the library build won't be correct. You may find more information about it in this SO post.





Then press Cmd+B to build the project and ensure the project compiles without errors.

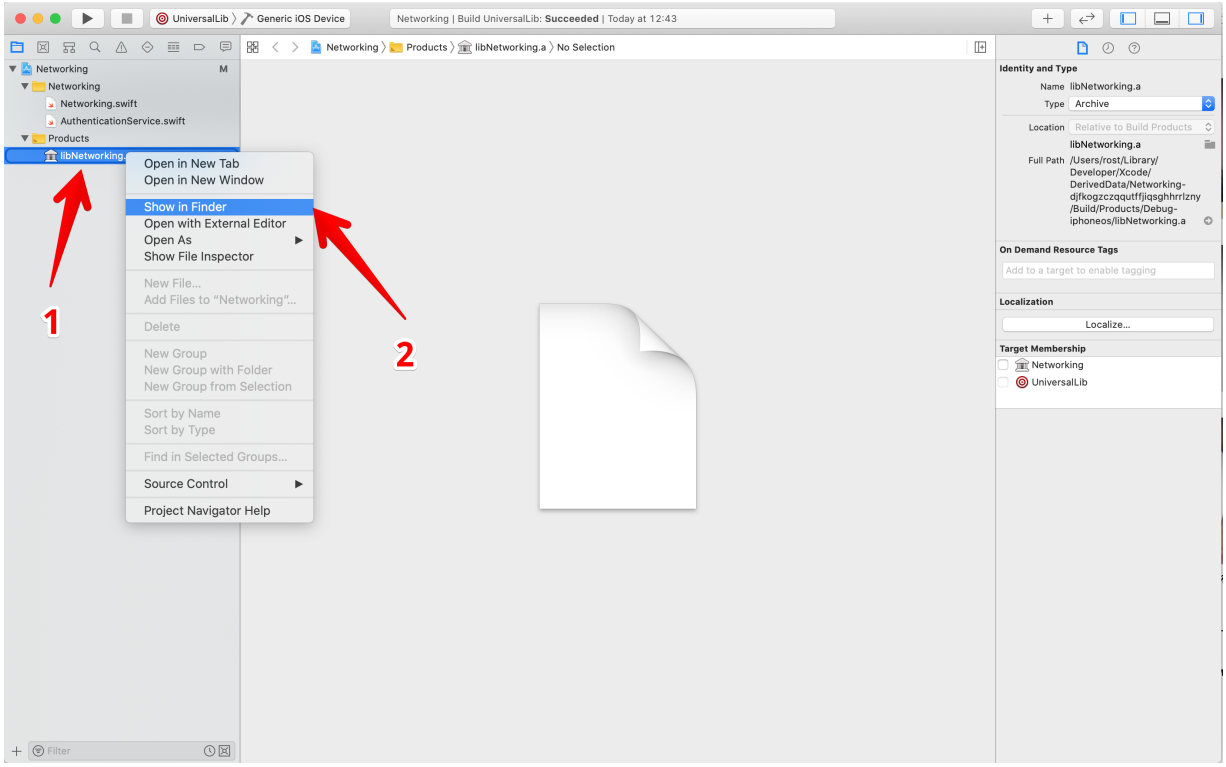
That's all we need to change in the static library's project … so not much. In the next section, we'll discover what we've compiled.

. . .

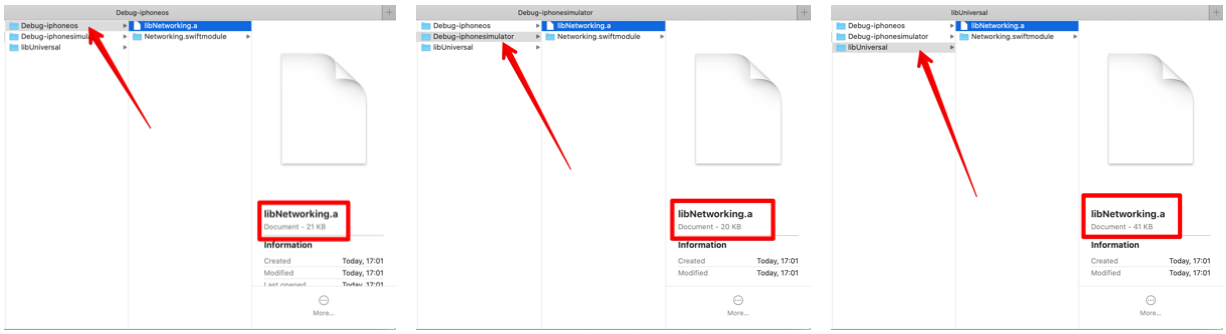
Investigate the Universal Binaries

In this section, we'll consider what we created and discuss some details. If you are interested only in the practical aspect of the tutorial, you may skip this section.

In the project navigator open Products, select libNetworking, right-click on it, and select Show in Finder.

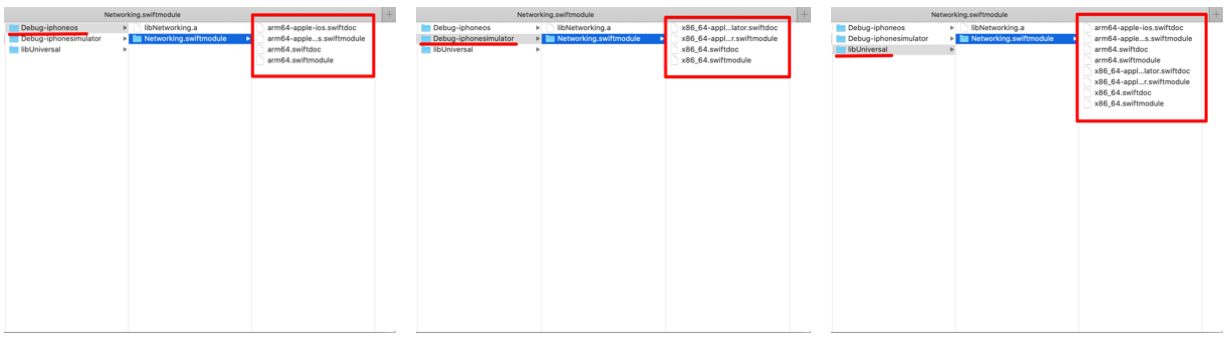


It will open a new finder's window, but it'll open the folder named `Debug-iphoneos`. There will be the binary for mobile devices only, and it won't contain code for the simulator.



If we compare these three folders' content, we'll find all of them contain a `libNetworking.a` file and a `Networking.swiftmodule` folder.

`Debug-iphoneos` and `Debug-iphonesimulator` folders contain binaries for both platforms, and the file size is almost the same — about 20k. But the `libUniversal` folder has the binary sized at 41k. It's exactly what we need — it's the binary that contains compiled code for both platforms, and we must copy the content of the `libUniversal` folder and paste it to your project later to make it build for both platforms.



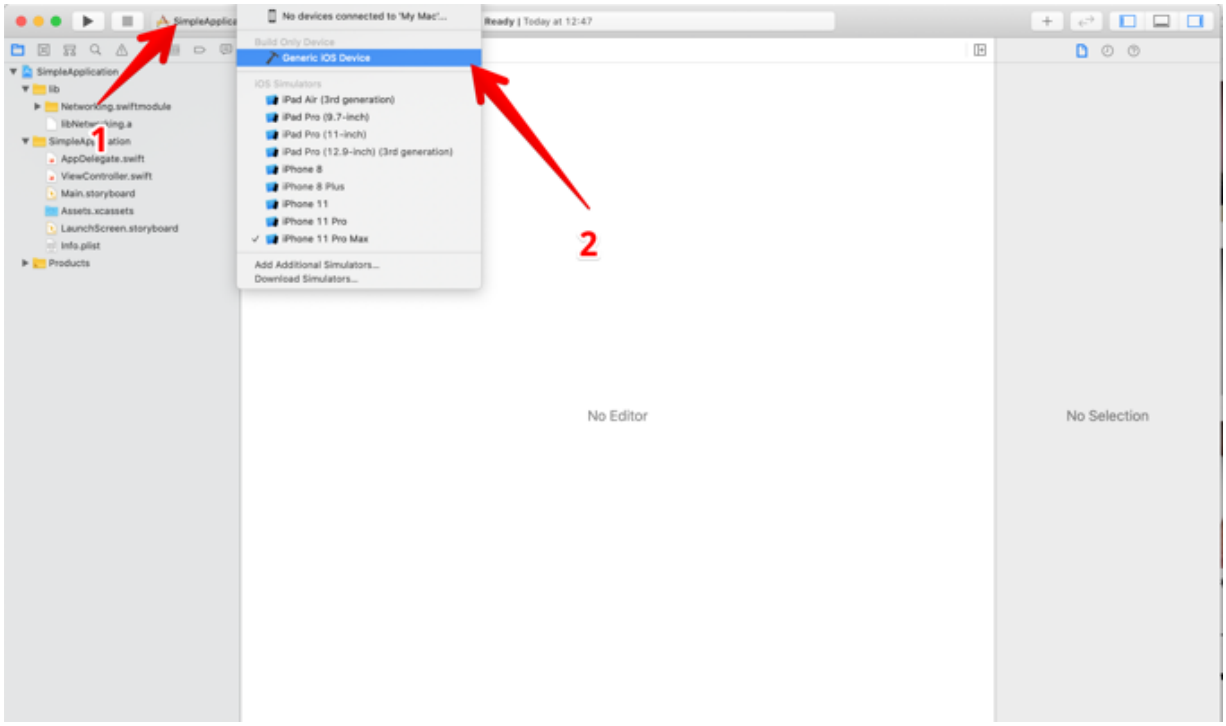
The `Networking.swiftmodule` folder contains `*.swiftmodule` and `*.swiftdoc` files that contain the library's interface description and documentation if it has been provided. We placed all the files generated by the compiler into one folder and will use them in our project.

. . .

Integration to Another Project

Prepare the project for integration

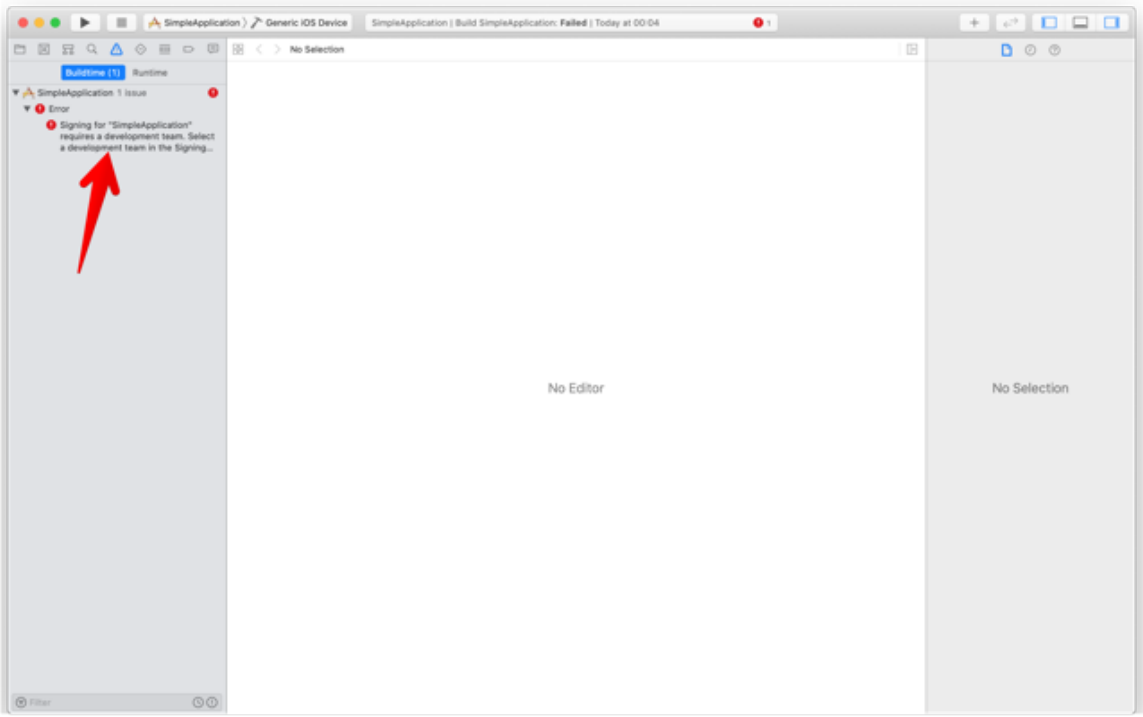
Then open the SimpleApplication project from the Starter folder. Select Generic iOS device in the scheme navigator, and press Cmd+B to build the project.



The project won't build. Instead, you'll see an error:

> *Signing for "Simple Application" requires a development team. Select a development team in the Signing & Capabilities editor.*

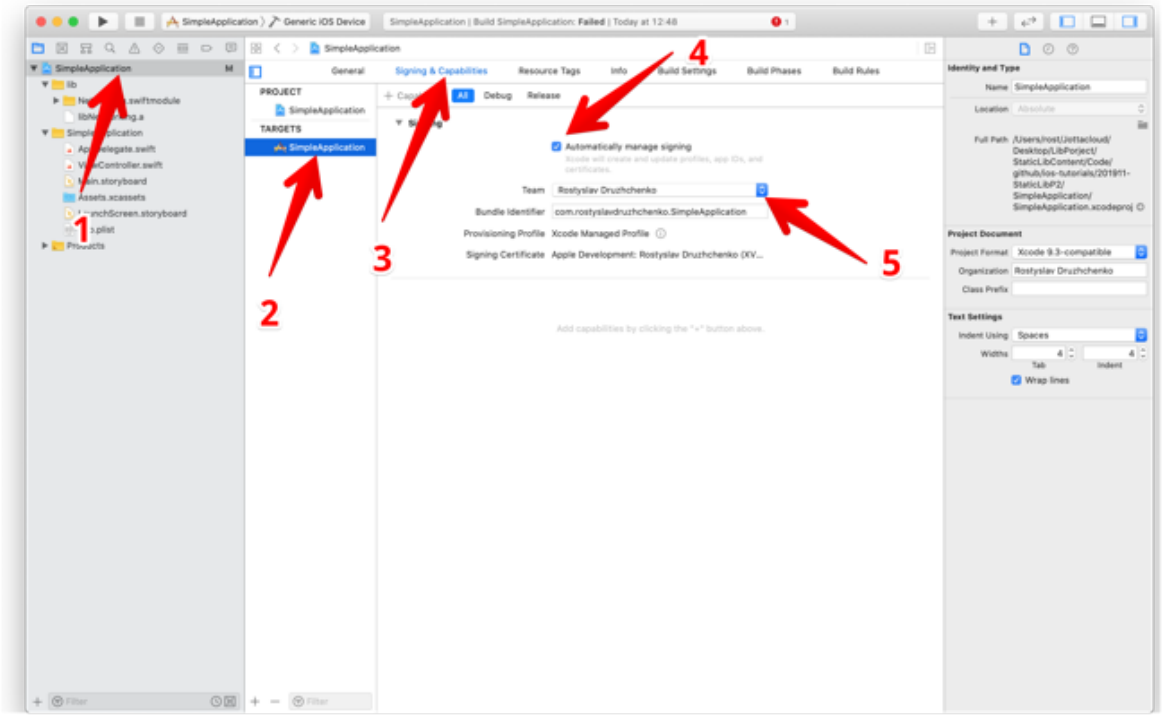
It's because building for a mobile device requires the final application's binary to be signed. It doesn't relate to the library-integration process directly, but it's a necessary step to run the application on a mobile device.



Signing applications is a huge and tangled process with a lot of possible scenarios, and in this tutorial, we're not going to dig

deep into it.

You may investigate this process on your own if you need it — we assume you're familiar with it and will demonstrate the simplest way to sign your application. This means you have a setup and valid Apple developer's account.



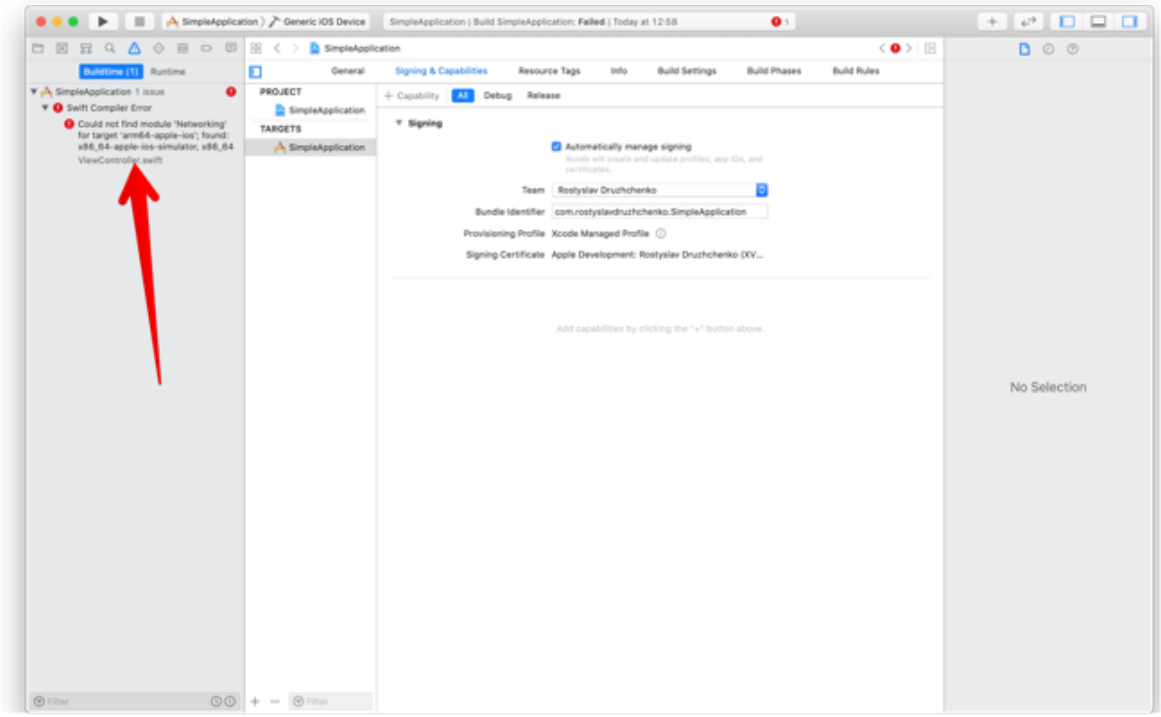
Select project in the project navigator (1), then select the target for the application you want to build (2).

Then select the Signing & Capabilities tab (3).

Check "Automatically manage signing" (4), and select a valid team (5).

Then try to build once again pressing Cmd-B. As a result, you'll have to see another error:

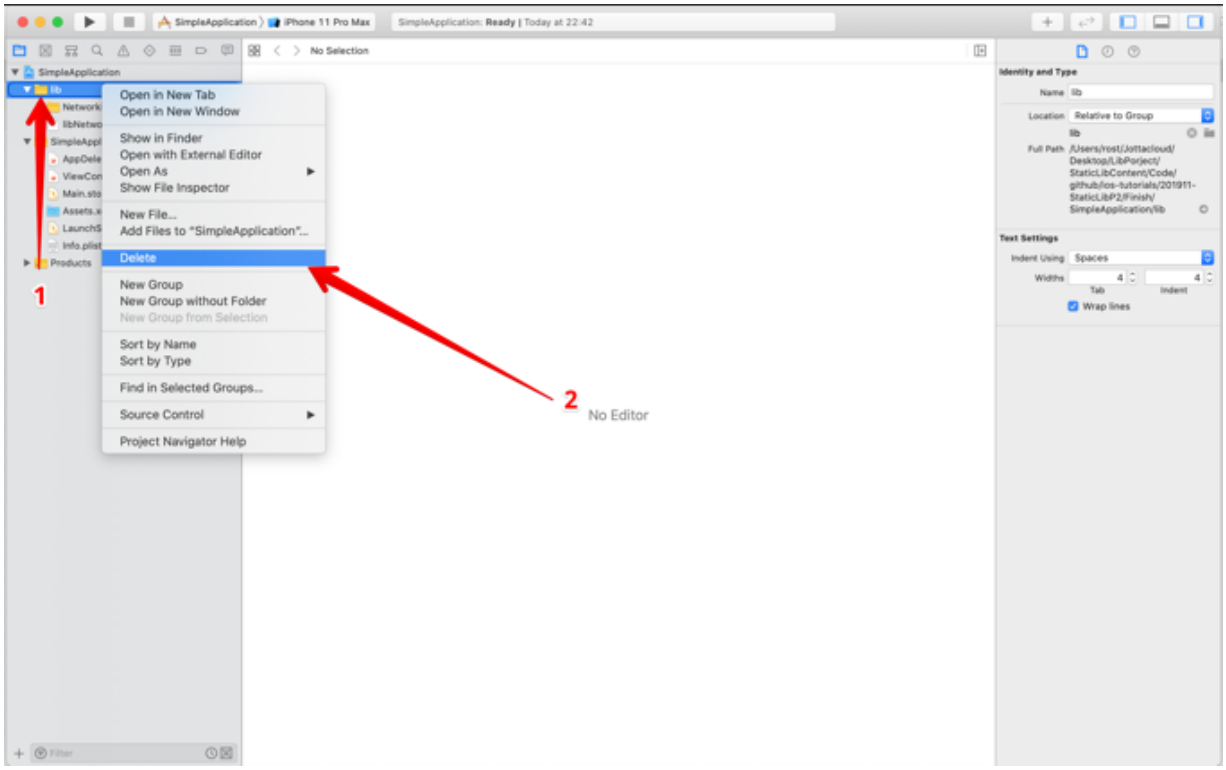
> *Could not find module 'Networking' for target 'arm64-apple-ios'; found: x86_64-apple-ios-simulator, x86_64*



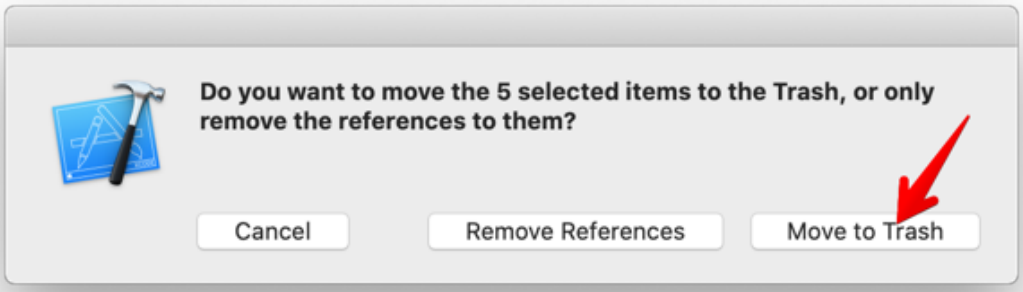
That's entirely predictable because we still haven't added appropriate binaries for the static library. So now we're ready to update the library's binaries.

Update library's files

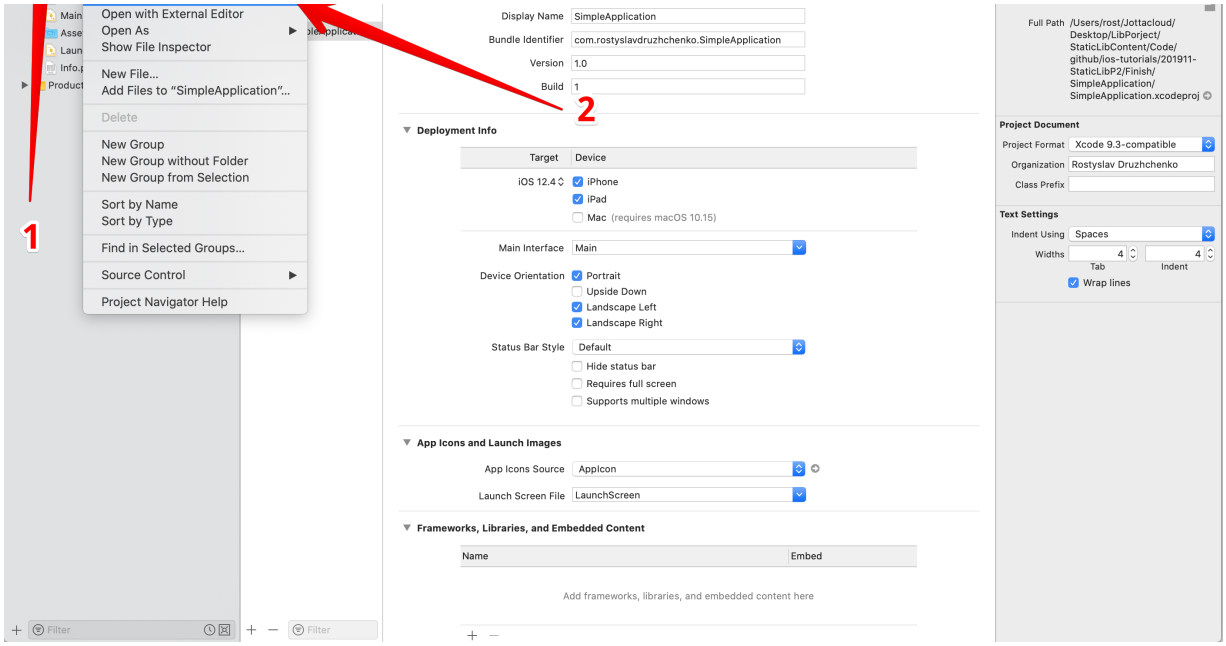
In the SimpleApplication project, open the project vavigator, find the `lib` folder that we builtin the previous tutorial (it contains binaries only for the simulator), and delete it:



Press Move to Trash in the window that appears:

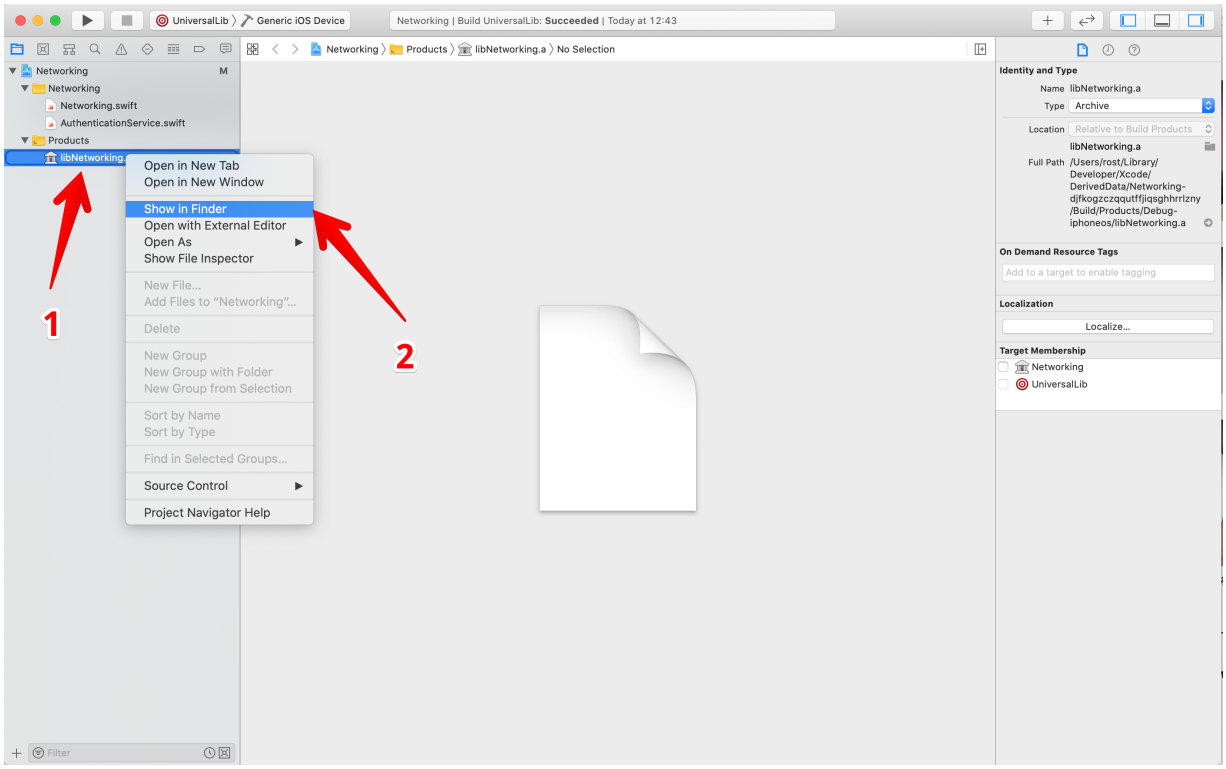


Than select the project's file in the project navigator. Right-click it, and select Show in Finder:

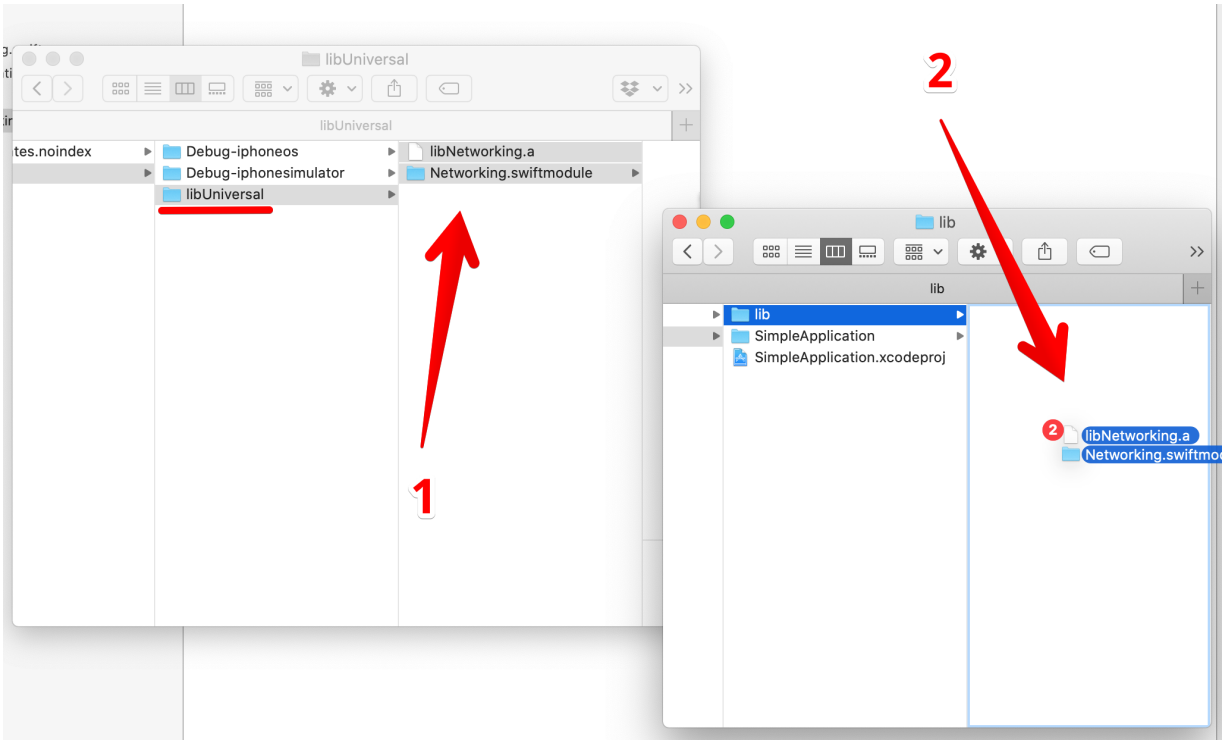


Navigate to the `lib` folder, and delete all the files there, if any.

Then go back to the Networking project, select `libNetworking.a`, and press Show in Finder.

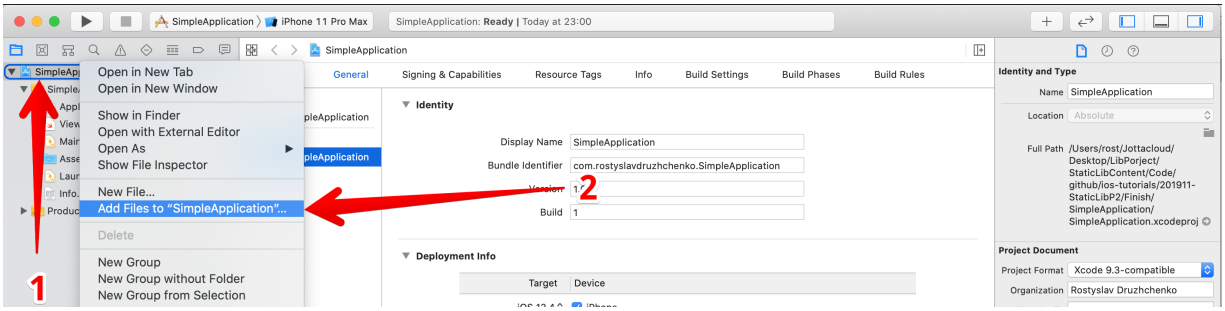


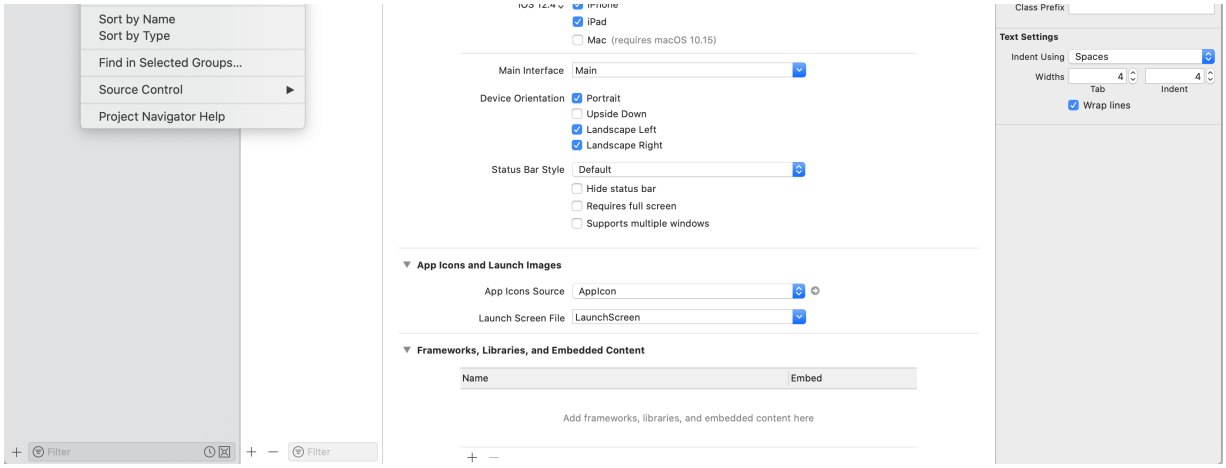
Navigate to the folder that contains the universal binary and `Networking.swiftmodule` folder. Drag and drop them to the `SimpleApplication/lib` folder:



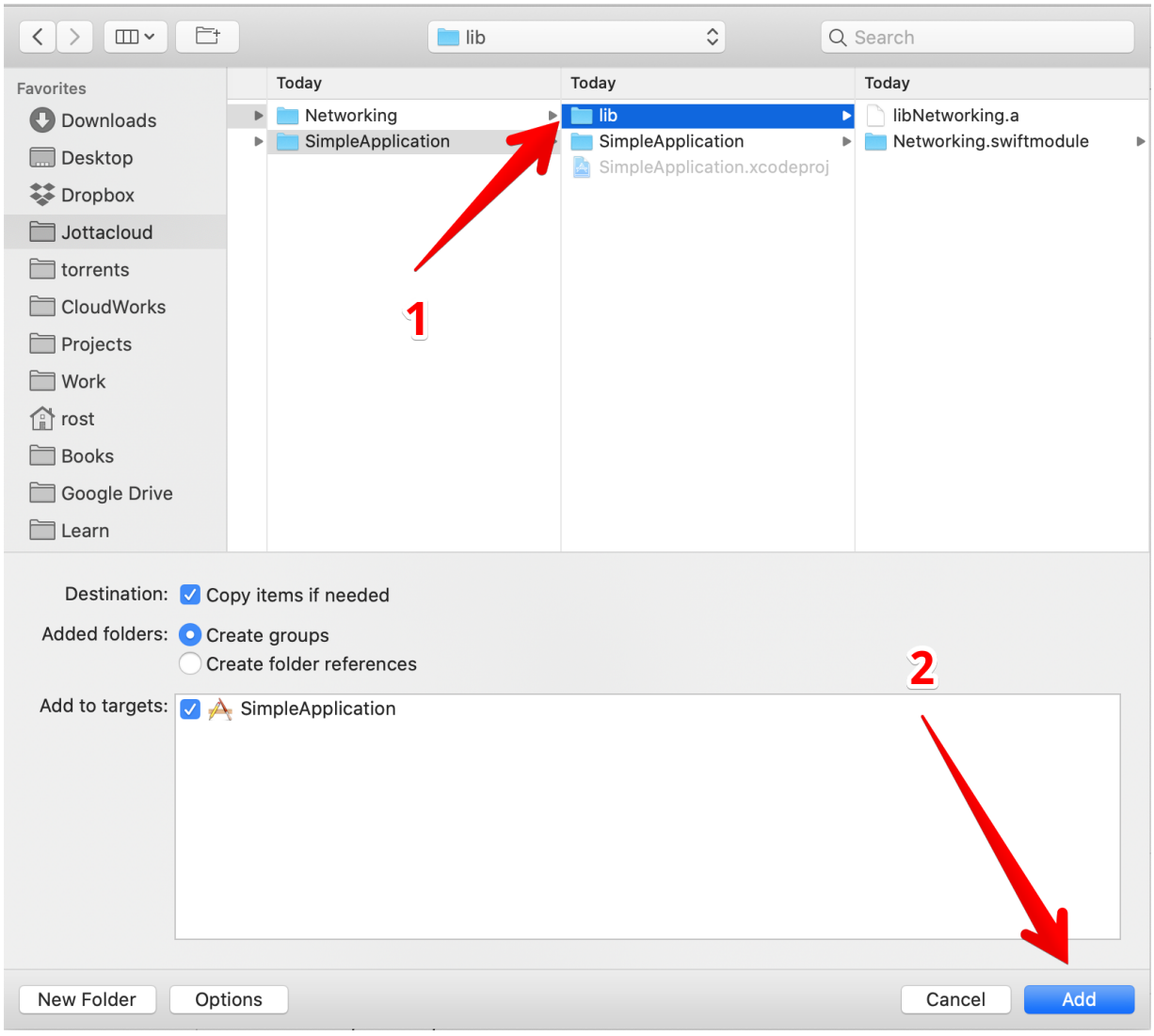
Make sure you dragged the files from the `libUniversal` folder, not from `Debug-iphoneos` or `Debug-iphonesimulator`.

Then open the `SimpleApplication` project again. Select the project file in the project navigator. Right-click, and select Add Files to "SimpleApplication"...:



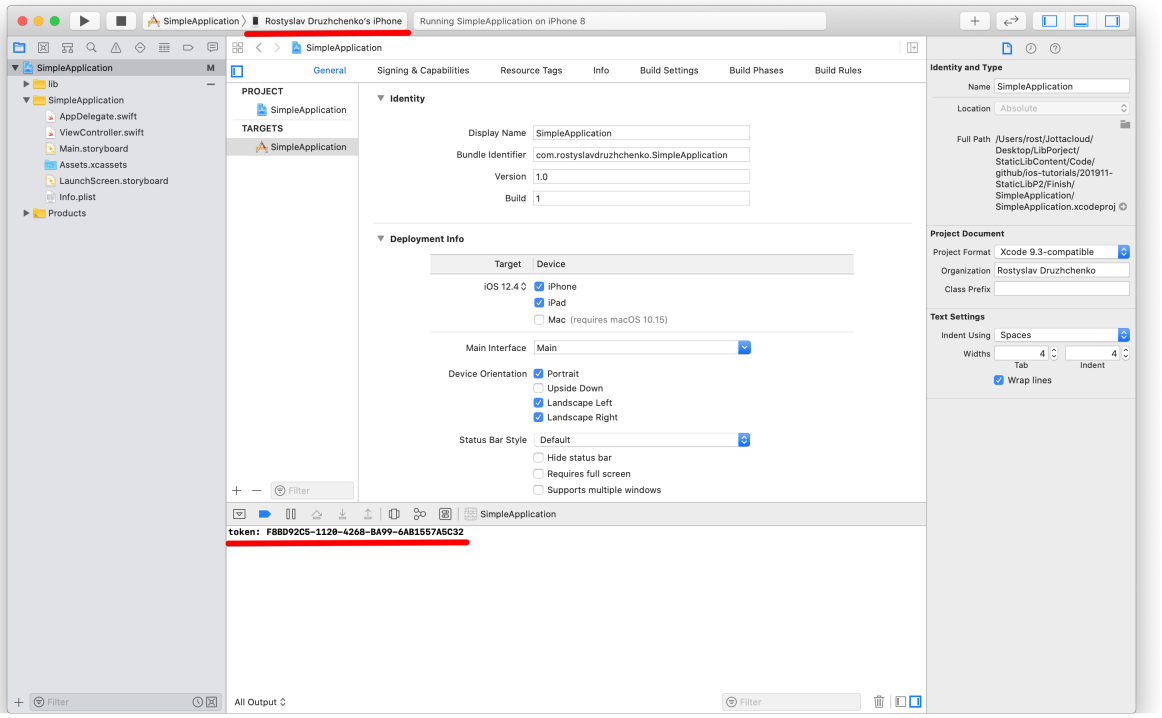


Then in the window that appears, select the `lib` folder, and press Add:



Press Cmd-opt-shift-K to clean up the project. Then, press Cmd-B, and the build should be succeeded.

You may run the application on a real device or on a simulator to make sure it works properly. In the Xcode's output, we should see something like this:



• • •

Conclusion

In this article we did and considered several things, namely:

- Made a universal binary for a static library written in Swift
- Considered a build script that can be easily reused in any project
- Integrated the universal library to another project and ran it on a simulator and on a mobile device

. . .

What's next?

In spite of the fact we created a static library and made its code available for use in other applications, there is plenty of space to improve:

Automation

Copy/pasting the library's binary can be automated. To do that, the run script should be slightly changed, and it also requires some changes in the environment.

Unit tests

One of the most important advantages of modularity is isolation and dependencies braking. It prepares a good ground for unit and integration testing.

. . .

References

- https://github.com/drrost/ios-tutorials/tree/master/201911-StaticLibP2 — full code version on GitHub that you may download and investigate
- https://medium.com/better-programming/create-swift-5-static-library-f1c7a1be3e45 — Part 1 of this tutorial
- https://stackoverflow.com/questions/51558933/error-unable-to-load-standard-library-for-target-arm64-apple-ios10-0-simulator
- https://github.com/apple/swift/blob/master/docs/Serialization.rst — a Swift document that sheds some light on `*.swiftmodule` and `*.swiftdoc` files

iOS Swift Programming Xcode Mobile

Discover Medium

Welcome to a place where words matter. On Medium, smart voices and original ideas take center stage - with no ads in sight. Watch

Make Medium yours

Follow all the topics you care about, and we'll deliver the best stories for you to your homepage and inbox. Explore

Become a member

Get unlimited access to the best stories on Medium — and support writers while you're at it. Just $5/month. Upgrade

About Help Legal