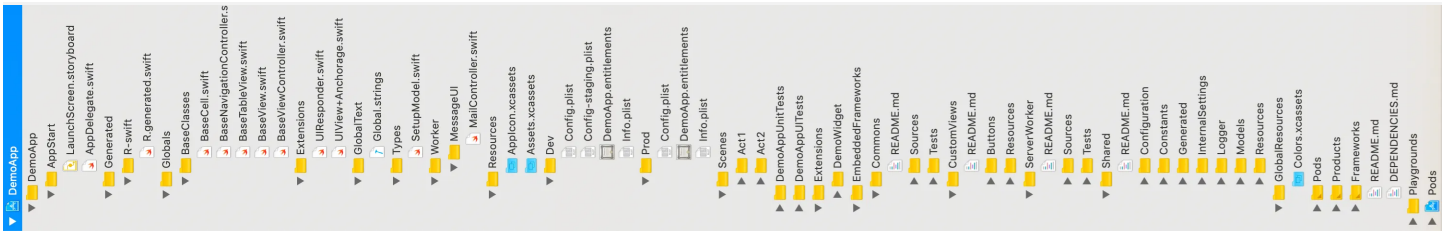


✦ Member-only story

PIECES OF A SCALABLE IOS APP ARCHITECTURE

An Example of a Scalable iOS Project

The DemoApp-Project (DAP) and some best practices (Part 1 of 2)

Sven Korset · Follow
Published in Better Programming
10 min read · Aug 16, 2019

Listen Share

In my article “The perfect iOS app architecture”, I introduced my own architecture and explained the concept of *Scenes*. I already explained the individual actors of the architecture on the basis of a more complex example project.

This article is about this more complex example project: The DemoApp-Project on GitHub. I also explain a few best practices related to general project setup based on that architecture, but also more widely applicable.

Incidentally, this is a post from the “Pieces of a scalable iOS app architecture” series.

Download the DemoApp-Project (DAP)

For this article to make sense, I highly recommend having the project in front of you. Therefore, please first download the DemoApp Project from GitHub. 😉

The project is opened via the “DemoApp.xcworkspace” in the “DemoApp” folder. There is nothing more to do because all necessary pods are checked in to be sure all dependencies are always available. By the way, this is already my first recommendation:

> Always check in all pods and other dependencies to the repository.

Pods are dependencies of the project. Should a dependency no longer be there, e.g. because a pod project has been deleted, then your own project is broken. Except the pods were saved within the repo. The few MB in the repo aren’t really driving up costs. In fact, **the risk of total failure is much greater than the cost of prevention**.

Of course, you should then take care of a pod failure and look quickly for a replacement, but at least you can still use the current state. Having this current state is often useful, because the client, of course, wants a critical app update with just a small text change done yesterday and does not understand why this should now take a week. 🙄

In the DAP, I use *CocoaPods* because it is simple and prevalent. I like it simple! 😄

Of course, you should regularly question the status quo and, if necessary, adapt it to the team and project. For example, Zalando uses a manual solution because it works better for them. I’m not sure if it’s really better to create the dependency manager from scratch with own scripts, but that’s something each team has to decide for themselves.

Anyway, there is a “Package Management Dilemma in Swift” as described by Noah Trupin. However, with the *Swift Package Manager* getting better integrated in Xcode 11, might this become the new *CocoaPods*? 🤔

Folder Structure

If you open the project in Xcode, you will find the main modules at the bottom level, like the app project itself, Unit and UITest folders, extensions, frameworks, etc.

Within these modules, a tree of groups and files further structures the project. It's important to note that:

> Folders on the HD should match the groups in Xcode.

Admittedly, that's not really so important because it does not prevent any merge conflicts or anything like that, but it's proof of professionalism if you keep your file system **neat**. Even Apple explicitly recommends this at WWDC19 in "Great Developer Habits". 😎

In any case, you should structure the code in groups. Hardly anything is worse than having all 1000 files in one folder. Who wants to scroll through every file every time to find what you need? 😖

If you open the "DemoApp" folder, you will find the groups:

- **AppStart**: Here is the AppDelegate and the start storyboard.
- **Generated**: Everything that is created automatically by scripts and tools.
- **Globals**: Some base classes, app-specific extensions, global text, special app types, and not in frameworks outsourced workers.
- **Resources**: AppIcon, App-only images, Info.plist, Entitlements, Configuration files.
- **Scenes**: All acts and scenes. No, we are not in a theater, and this is also not related to SceneKit or the new SceneDelegate introduced in iOS 13. 😜

In "Scenes" you will find the folders "Act1" with the subfolders "Scene0" and "Scene1", as well as "Act2" with the subfolder "Scene2".

An act groups several Scenes. However, the grouping criterion here is certain dependencies that can only be resolved over time. Most of the time this can be mapped to a special app phase.

For example, here "Act1" could be renamed to "Onboarding" to represent the phase in which the app has not yet been fully set up by the user. "Scene0" would be called "Splash" and "Scene1" as "Login".

After successful onboarding, the user should have logged in, so that a user object is available and can be resolved as another dependency for the following scenes. This phase could then be called “LoggedIn” rather than “Act2”, but that’s of course only an example.

In the DAP, this is sketched with a user-object. The more detailed approach along with the background is explained in my article “Dependency management done manually in Swift”. 🤓

How a scene is structured exactly is described in my article “The perfect iOS app architecture”. It’s important to mention that:

> All Scene-related dependencies should be close in Xcode.

Too often I’ve seen the files structured by type, i.e. a “VC” folder where all *ViewController* come in. A “view” folder for all views, one for all cells, etc. The result of such a structure, however, is that you have to constantly jump in Xcode over the entire project tree.

When working on a *ViewController*, you often edit the associated view, the cell, etc. It makes more sense if all these classes that belong to the *ViewController* are also in the same folder or in a subfolder. Then you have everything you need close together for a **better overview and shorter navigation**.

Scenes follow this principle. Everything that belongs only to this *Scene* is in the *Scene* folder or a subfolder. If you do not work on this *Scene*, you will not need to open this folder. Overview by suppression! 😋

Xcode Template

If you often create new *Scenes*, then it will be annoying to create each file manually and write all the skeleton code again and again. It’s better if you can do it only once and then automate.

That’s why there’s a “Templates” folder in the repo with an Xcode template. The included *Readme* explains how to install and use the template in Xcode.

> Always automate work when setting up the automatism and maintaining it costs less than it saves over time.

The tricky part is that you have to estimate how much time it will cost and how much time it will save. **Experience** and good time tracking help here. 🧐

Development Tools

The "Automate" advice also includes certain command line tools, such as "R-swift" (or "SwiftGen"), "SwiftFormat" and "SwiftLint". In the DAP, these are integrated via *CocoaPods* and configured in the corresponding *Build Phases* of the *Targets*.

Once properly configured, the tools will always run as soon as you compile the code. These then automatically create certain code, format all Swift files to comply with the coding guidelines and perform additional checks to find specific vulnerabilities and code smells.

That's three important points at once! 😳

> Use strong typed references, automate code formatting and fix any warnings.

Strong typed is, well, type-**safe**. Formatted code is just **prettier**. And who likes yellow warnings in their face? Fix the warnings and make the code more **robust**. 😅

With R-swift, you simply call `R.string.scene0.title()` to get back the localized string with the key `Title` from the `Scene0.strings` file. Non-safe would be a call like `NSLocalizedString("Title", tableName: "Scene0", comment: "")` where you might risk misspelling the key "Title" or "Scene0". Try to use strong typed references everywhere, from localized strings to images, colors and any constant values.

Why automatic code formatting? Because we are all human and don't always implement the guidelines 100%, even if we do our best and other people do the code

review. Plus, a machine does what it does best — so, why do it yourself? 😌

Compiler and linter warnings indicate possible problems. Maybe it's not an issue yet, but those are the things you should improve so that they don't turn into problems one day.

In the *Build Settings,* you can also set *Warnings* to be treated as *Errors*, but that's too sharp, I think. If you're developing something, it's not clean yet, but you still want to be able to run the code. However, you should have eliminated **all** warnings after feature completion, at the latest.

Generally speaking, just make the code pretty and write code you still love tomorrow! 😍

Modularization

The DAP shows that you can use embedded frameworks well to modularize code. As explained in my article "The perfect iOS app architecture", **modularization** is the key to a reasonable architecture. So, it's about splitting code into smaller units, structuring them and making them more **maintainable**.

Divide and Conquer!

As described in my blog post "Structure constants in iOS with enums" *enums* can be used in the form of namespaces to structure the code. However, using frameworks goes even a step further. The advantages of embedded frameworks especially are often underestimated. 😐

Code that you outsource to embedded frameworks automatically gets a better interface because you're forced to think about it. Frameworks can also be used in other targets and are usually heavily encapsulated.

In addition, parts of the *UnitTests* can be outsourced with frameworks. Code that is now in a framework can be tested there. Framework tests only have to be run if the code in the framework changes, not every time anymore.

With embedded frameworks, mental ballast is simply outsourced. 😎

Incidentally, Xcode 11 just introduced a better-integrated way of using the *Swift Package Manager,* so maybe this will become an alternative to embedded frameworks.

Configurations

In a project, one often develops in different environments. For example, all server requests are to go against the Dev server, sometimes the Staging server and later against the Live server. Or different test scenarios should be used, e.g. the database should be deleted with each app start and then filled with test data. For that, you don't want to change the code all the time. 😫

For these environments, it would be good if you don't have to change any code but just to load a different configuration file. This can be done in different ways, one is to use config files in Xcode as described in my article "Dev/Staging/Prod Configs in Xcode". 😌

Documentation

You open a project and find no *Readme* for a rough overview or some details. The code lacks any comments. What does this method do, what is this parameter for and what does that cryptic line mean? Now you ask yourself: "What the hell did I do there?" 😭

> Always write down the context and use simple English.

I know there are other opinions out there, like those of Brian Norlander who proclaimed to "Stop Writing Code Comments", because every programmer should write code expressive enough, so comments become unnecessary. He brings good arguments and even our saint Robert C. Martin a.k.a. Uncle Bob likes to write code without comments.

However, on the other hand, you have people like Christopher Laine, who explains why "Self-documenting code is (mostly) nonsense" and Cindy S. Cheung who goes more into detail into the benefits of comments with her article "Pleeeeaase Explain Your Code". I'm sure you've already guessed it, I'm more of the opinion of the latter. 😆

Code comments simply help to clarify what the code does. For a parameter named "title" just to write the comment "the title" is obviously not helpful. That's why you should write **context and not just text**, e.g. "the title text shown at the top of the view". I really don't want to name the parameter `titleTextShownAtTopOfView`.

The same goes for inline comments describing what a subsequent code block does. You no longer have to read and understand every line of code. Simply skim the comments to keep track.

Comments just save a lot more time than they take to write. Not only is it usually easy to write comments, but it will make things easier for anyone (including you) who reads your code later on. Comments are written once (and maybe updated a few times) but read often. They normally speed up things, especially for people new to your code.

Don't believe me? Then simply go through the code of my DemoApp Project. I'm sure you can figure out how it works without reading any comments, but it would certainly be faster with reading them. 😉

Long story short, pleeeeaase always comment your code! 🙏

The same applies to *Readmes* or other types of documentation. It's better to sit for a day on a document that explains the rough structure of the project, any key decisions, concepts and specific dependencies than to spend several weeks reverse engineering.

And don't say that will never happen. You know Murphy's Law! 😣

Documentation should basically be written in easily understandable English. That's normally self-evident. English is just the one language that the largest section of the world's population understands.

You never know how the company and project shifts and who will eventually have to maintain the project. If the documentation is written in an unknown language, it

could be missing altogether. Even bad English is better than no help. 😤

For documentation of dependencies, there are interesting suggestions on how to use markdown files for that. The article "Tracking dependencies with DEPENDENCIES.md" provides a nice solution I adopted for the DAP.

However, I find it error-prone to document pods in a separate file. All the pod dependencies have already been listed in the *Podfile*, so why not document them right there? Keeping an extra file only results in it not being maintained, and what is not maintained rots away over time.

Therefore, the *Podfile* is the place to document any pods and the *DEPENDENCIES.md* file for any other dependencies. Just don't forget to document them when adding non-pod dependencies! (I bet you will) 😜

What is the App Doing Anyway?

Nothing — at least nothing useful. 🤣

The project exists only to present certain aspects of the architecture and concrete solutions to common problems. I will go into that in Part 2.

An example of a scalable iOS project, Part 2

The DemoApp-Project's (DAP) Scenes explained

Swift IOS Projects Demo Best Practices

Follow

Written by Sven Korset

295 Followers · Writer for Better Programming

Just a guy with some thoughts 😊

More from Sven Korset and Better Programming

Sven Korset in Better Programming

Dev/Staging/Prod Configs in Xcode

Using .xcconfig and config.plist files for different environments

✨ · 8 min read · Feb 7, 2020

-- 5

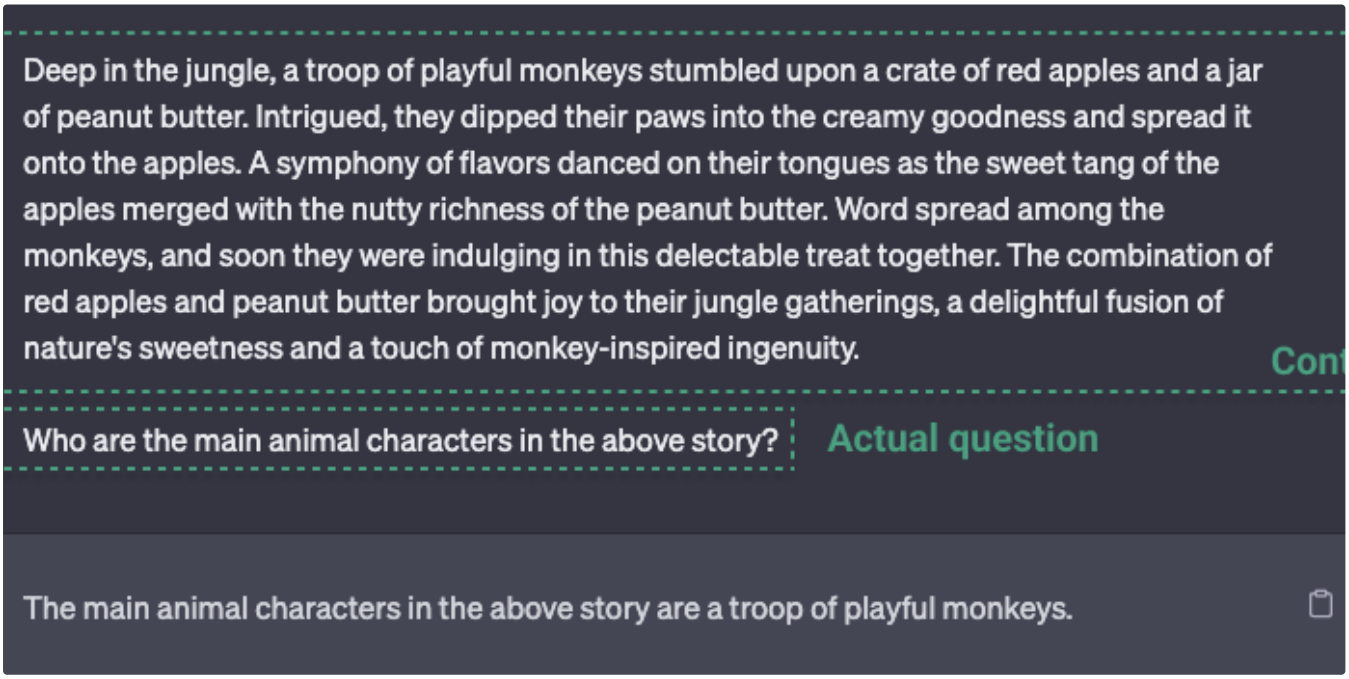
Benoit Ruiz in Better Programming

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

-- 261



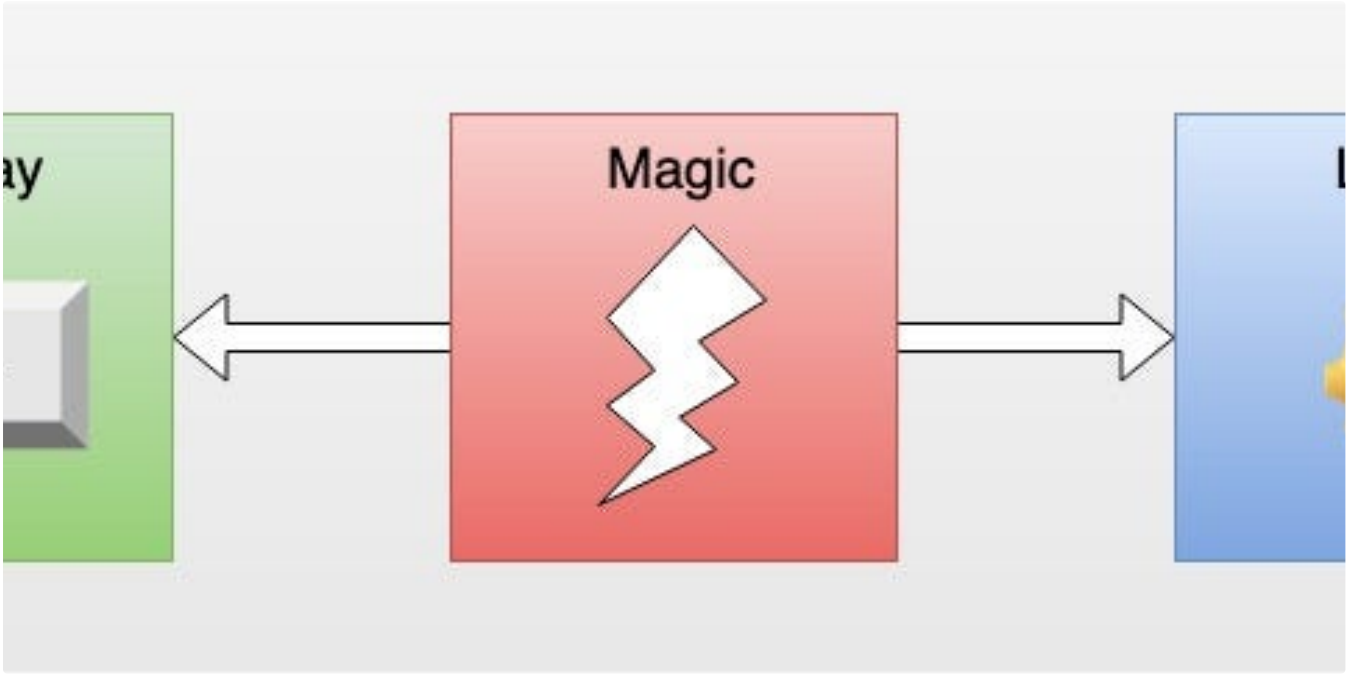
Sami Maameri in Better Programming

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

-- 14



Sven Korset in Better Programming

The perfect iOS app architecture

MVVM, Clean Swift, VIPER?—Just invent your own!

· 11 min read · Aug 16, 2019

-- 2

See all from Sven Korset

See all from Better Programming

Recommended from Medium

Kanagasabapathy Rajkumar

Getting Started with The Composable Architecture (TCA)

This article will help in understanding the Swift—Composable Architecture and how it benefits SwiftUI Apps efficiently.

9 min read · Oct 13, 2023

--

二	two
三	three
四	four
五	five
六	six

Itsuki

Localization in iOS/Swift

Localization is definitely a really important part for the success of an app, especially if you are targeting global market. I remember...

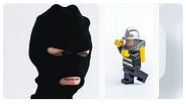
14 min read · Oct 1, 2023

-- 1

Lists

Apple's Vision Pro

7 stories · 53 saves

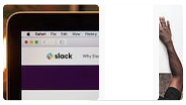


Tech & Tools

16 stories · 141 saves

Icon Design

36 stories · 221 saves



Productivity

237 stories · 321 saves

Batikan Sosun in Better Programming

Managing Navigation Between Multiple Modules in an iOS App Using Dependency Injection Containers

In modern iOS app development, breaking up an application's functionality into different modules or screens is common. This modular...

5 min read · May 2, 2023

-- 2

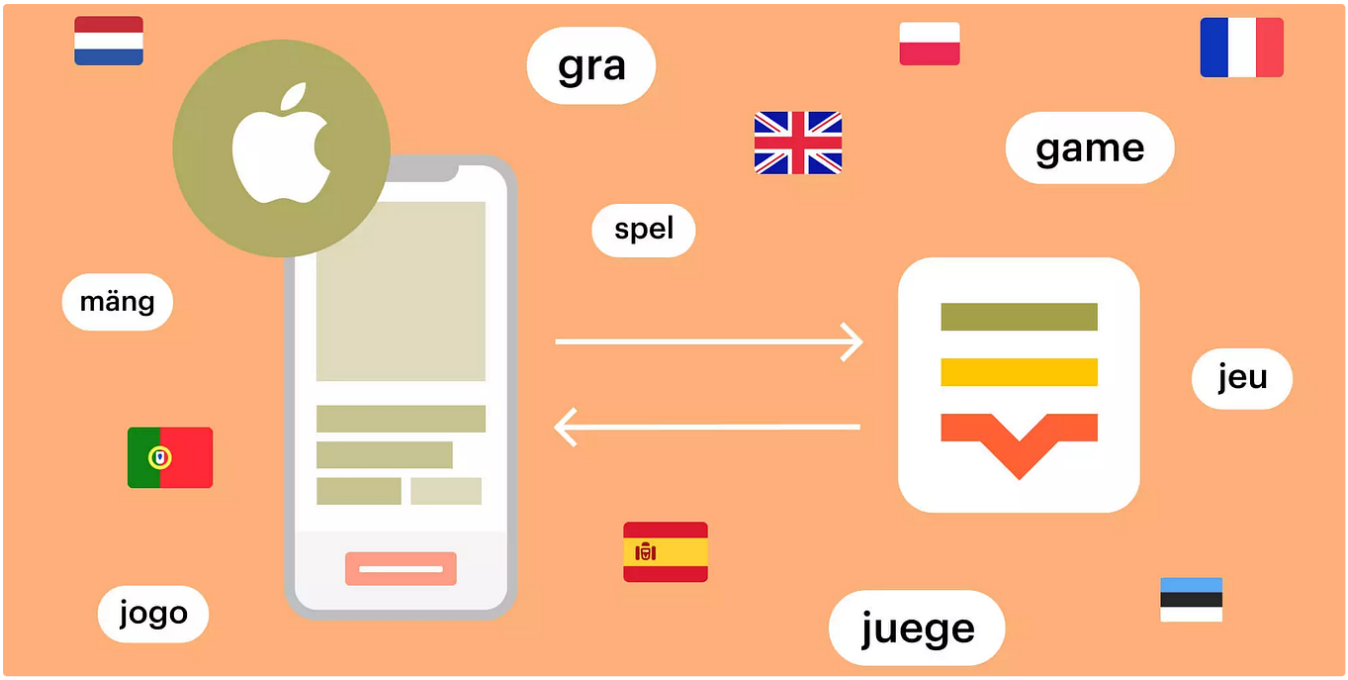
Roli Bernanda

Clean Architecture: Use-Case Centered Design in SwiftUI— MVVM

The idea is straightforward, Our design should not rely frameworks. Frameworks are tools to be used, they should not dictate how we build...

5 min read · Aug 25, 2023

-- 2



Muhammad Kashif

How to Localize Your iOS App in Swift Using Extensions and Localization Files

Localization is a crucial aspect of app development, as it allows you to reach a global audience by providing content in different...

4 min read · Nov 3, 2023

 --  


Wahyu Alfandi

A Beginner’s Guide to Clean Architecture in SwiftUI: Building Better Apps Step by Step

Photo by Lala Azizli on Unsplash

9 min read · Aug 24, 2023

-- 7