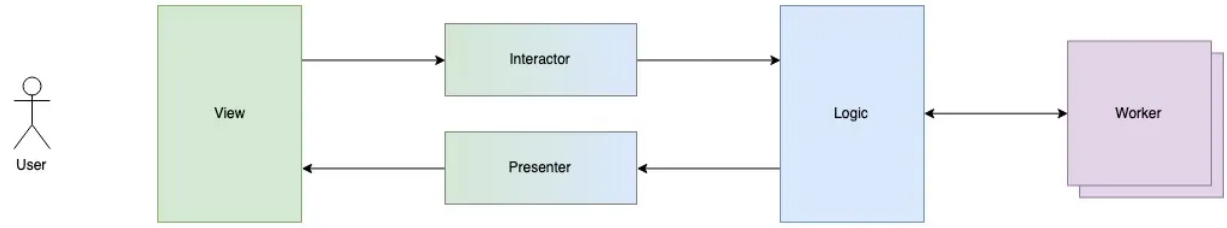


✦ Member-only story

PIECES OF A SCALABLE IOS APP ARCHITECTURE

Decoupling Display and Logic in iOS

The cycle of flow: View → Interactor → Logic → Presenter → View

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

In my article “The perfect iOS app architecture”, I present an architecture that divides a *Scene* among others in *View*, *Interactor*, *Logic* and *Presenter*. I would like to explain in a bit more detail why this division makes sense in my opinion.

The flow cycle would then be:

1. The *View* provides the view hierarchy.
2. The *Interactor* hooks itself into the *View* to get notified about user interactions.
3. These interactions are then mapped to methods of the *Logic*.
4. The *Logic* computes the inputs and manages any state changes.
5. The *Logic* tells the *Presenter* to update the *View* according to a new app state.
6. The *Presenter* updates the *View*.

Incidentally, this is a post from the "Pieces of a Scalable iOS App Architecture" series.

View

The task of a *View* is to show the user something. On a touchscreen-based device such as the iPhone or iPad, the *View* also receives all user input. Thus, the *View* represents the interface between user and app.

A *View* would traditionally have two tasks, the presentation and the interaction. Usually, this is then handled by a *UIViewController*, which listens via actions for user input, calculates something, then formats the result and fills the *View* with it.

However, if you want to respect the **Single Responsibility Principle** from the **SOLID** principles (nicely explained by Chidume Nnamdi 🔥💻🎵🎮 in his article "SOLID Principles every Developer Should Know"), then this belongs into their own classes. So, let's put these tasks into own classes and call them *Interactor* and *Presenter*.

What's then left for the *View*? 🤔

The *View* provides the static structure, the view hierarchy. It creates and manages the subviews, providing an interface to them. Other actors such as the *Interactor* can hook themselves into the subviews to intercept the user input, or the *Presenter* can directly fill subviews with data.

If the view hierarchy should change, e.g. because an embedded view should be inserted, then that would be in the task area of the *View*.

Initial values of the subviews can also be set by the *View* because the *View* creates these subviews. For example, the text and style of a button in the view hierarchy could be set when creating the *View* because these are static default values.

But, if a style or value should change at runtime, then that depends on a special state of the app and that would no longer be the task of the *View*, but rather the *Presenter*.

So, if the *Presenter* gets notified by the *Logic* that the login button should now show a red "Cancel", then it is the responsibility of the *Presenter* to adjust the button accordingly, not the *View*. Of course, the button itself can provide a corresponding

function that changes state and adjusts the look, but it remains up to the *Presenter* to call that function on the button.

The *View* does nothing except to provide the view hierarchy. 🙂

Views done programmatically

By the way, in my DemoApp Project (DAP), the *Views* are created programmatically and not via xib or storyboard.

Why? 😩

Dominik Veselý gives a nice comparison in his article "User interface programming on iOS", which I can only agree with. Writing views in code just means:

- Merge conflicts become solvable
- View-code can be reviewed
- View hierarchy and constraints can be commented on
- Related code becomes less fragmented

The only advantage of xibs and storyboards is that creating the views is faster and more intuitive. However, if you do not want to program a quick prototype but want to write professional, maintainable code in scalable teams, you really cannot avoid a programmatic approach.

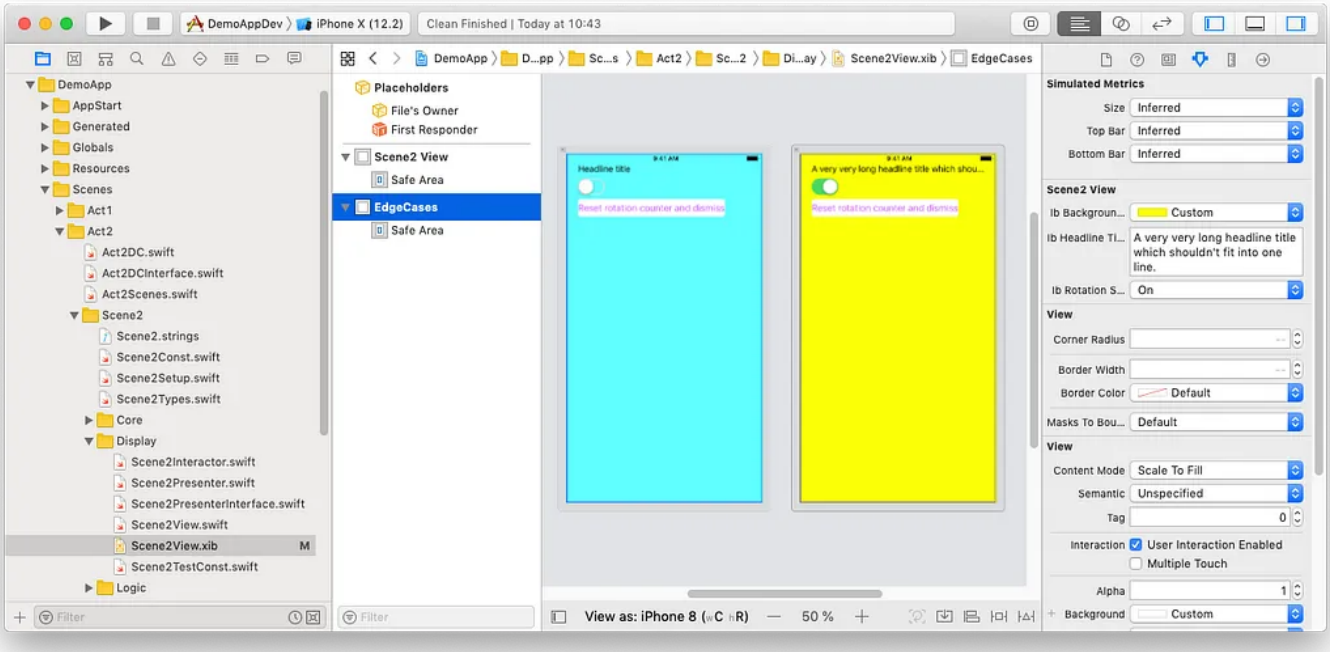
Give it a try. After a while you will surely love it. 😘

I can already hear you saying "But if you only have the code of the view, then it's difficult to design a visual image of it in your head!" 🤯

That's why I rely on *IBDesignable* and provide xib files. However, these xibs are not used to create the view and thus are not delivered in the app. They serve only as a visual aid in the development, because thanks to InterfaceBuilder and IBDesignable, you can easily visualize the code with xibs so that you can see relatively quickly what the view looks like.

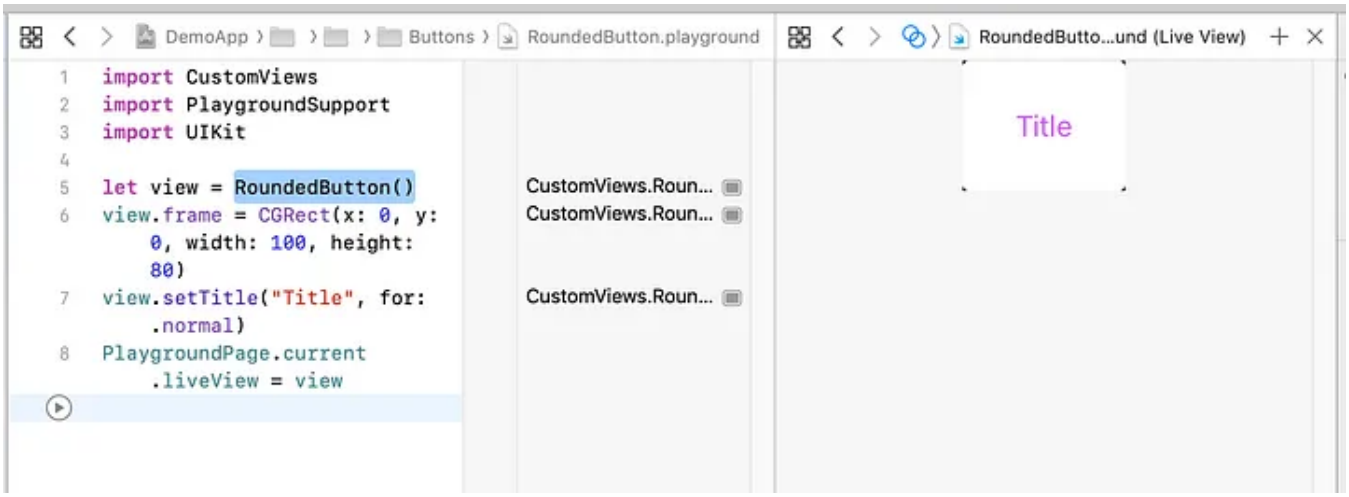
With *IBInspectable* you can even provide access to different states and test data, so that you can even display different representations of the view in the xib. Add

multiple view instances into the xib and provide different data for the inspectable properties, et voilà, you can see all edge cases at once and live.



If you outsource custom views into your own frameworks, then you can even embed them in *Playgrounds* and represent a live interactive view.

I did that with a “RoundedButton”, which you can find in the DAP under “EmbeddedFrameworks/CustomViews/Buttons”.



RoundedButton.playground

This has the advantage of seeing and even testing user interactions like button taps without having to run the entire app. Isn’t that cool? 🤩

Apple seems to be moving in a similar direction with the new *SwiftUI* framework and its declarative approach. That makes me very happy, and I'm sure I'm not the only one.

SwiftUI even does the *View* part with less code and a direct, manipulable, visual representation that will replace the xib approach. As soon as iOS 13 is sufficiently spread and the minimum iOS requirements of a project allows it, we should switch to *SwiftUI*. 😍

Interactor

As already briefly mentioned, the task of the *Interactor* is to map all user interactions to the business logic and to call appropriate use case methods at the *Logic*.

To do this, the *Interactor* must be able to access the *View*'s subviews directly in order to register for corresponding actions. You could also put the *View* behind an interface so that the *Interactor* does not directly depend on the *View*, but that only inflates the *View* unnecessarily, making it far more complex than necessary without any real benefit in the maintainability or testability.

Not sure? 😉

What would the *View*'s interface look like compared to the direct *View* class? You will have to expose all subviews somehow. Well, the View doesn't do much more than provide an interface to all subviews anyway. And what do you want to test? It makes no sense to test the *Interactor* without its *View*. You just will not replace the *View*'s class with a dummy. So why abstract the *View* away then?

More interesting is actually how this binding looks like between the *View* and the *Logic*. And when you think of *binding*, you certainly think of Rx.

In fact, in the DAP I use *RxSwift* and *RxCocoa* to couple behavior to button taps, switch changes, etc. True to the motto: If this happens, then that should be executed. Of course, Rx is not mandatory, but it makes it much easier, for example, if you want to install a throttle for textfield changes, so that a new server request is not sent with every keystroke.

The *Interactor*, like the *View*, could eventually be replaced by *SwiftUI* and *Combine*. Unfortunately, you'll have to wait here one to three years for a sufficient distribution of iOS 13. 😴

Logic

Ah, the *Logic*. It's the heart of an app. It's a collection of use cases. It contains the business logic. Until the awareness of the Massive View Controller problem, this code was still in the *UIViewController*. Now it lets the *Logic* get massive. Yippee! 🤠

So that doesn't happen, the code should be further modularized and outsourced, which is best done in frameworks and encapsulated by workers. So, instead of letting the *Logic* execute any server requests, it should be done by a *ServerWorker* and the *Logic* will only call the appropriate method on the worker.

This reduces the *Logic*'s code more or less to simple checks, state changes and delegation. Often common use case requirements can be mapped directly to something like: change this and if this and that is true, then *foo* should happen. But *foo* is then up to the *Worker*, *Presenter* or *Navigator*.

Of course, the *Logic* gets all dependencies passed via DI so that they can be mocked for *UnitTests*. The *Logic* is the *UnitTest* candidate par excellence. But that is also true for every worker.

The methods of the *Logic* are called mainly by the *Interactor*, most of the time because the user has made corresponding inputs, which lead to a state change. However, it may also be that the *UIViewController* needs to communicate something to the *Logic*, e.g. a rotation of the device.

The *Logic*, in turn, does not really know anything about the *View*, but merely instructs the *Presenter* to perform appropriate view updates. Through the use of *Interfaces*, the *Logic* is thus strongly decoupled from the other components.

Just try to keep the Logic as small as possible. 😅

Presenter

The responsibility of the *Presenter* is to update the *View*. So, if the *Logic* is transmitting data or a special state, then the *Presenter* must manipulate the view accordingly.

The *View* only provides the view hierarchy with the subviews. The *Presenter* directly accesses the subviews of the *View* and sets styles and values.

Accordingly, the code in the *Presenter* is also pretty straightforward: take this parameter and assign it to that subview. From time to time, a formatter may come into play to format values before assigning them, but there is rarely more logic in the *Presenter*.

However, there are still some peculiarities! 😡

Alerts

Alerts, meaning *UIAlertController,* are *UIViewController* and presenting them, one would expect, would be the task of the *Navigator* because it also presents all other *UIViewControllers,* right?

Not quite. ☹️

The *Navigator* is not there for presenting UIViewControllers, but rather to navigate to other scenes. An alert is not a scene. Therefore, alerts are not the responsibility of *Navigators*.

So, if the *Logic* wants to show something like a *serverError,* that's not a scene change, but a simple display of information within the scene. This would not necessarily have to happen via an alert, but could also be done via a simple text output in the existing view. Or it could be something complex, like as a custom view with a controller presented modally.

Aha, a modally presented *UIViewController* is almost a scene transition, isn't it? 🤨

If you write this *UIViewController* implementation as *Scene,* then yes, because then you pass the dependencies again. You have the *Core* with the *Navigator* to come back, and maybe the *Navigator* also leads to other scenes.

However, using a *UIAlertController* will never lead to other scenes or *UIViewControllers*, but always back to the current scene. You do not hand over any dependencies and you do not have all the architectural overhead. In principle, this alert does not differ from displaying a simple view or an embedded view in the current hierarchy.

Navigating to *Scenes* is the task of the *Navigator*, while presenting other stuff like *UIViewControllers* is the task of the *Presenter*. 😜

A popover would also be displayed via the *Presenter* and not via the *Navigator*.

Embedded Controller

What about an embedded ViewController? So, a view with its own *UIViewController*, which should be added via `addChild` to the current *UIViewController* of the Scene? Let's assume a custom error displaying controller for *serverErrors*. 🤔

It's the same principle as with the alert: It is not a *Scene*. The *Presenter* adds the controller via `addChild` to the *UIViewController* and presents the view.

Here, however, the controller's view should be inserted into the view hierarchy with `addSubview`, usually together with new constraints. That would then be the task of the *View*, e.g. using an `addEmbeddedView` method or similar, which is then called by the *Presenter*.

But who then creates the object of this *ViewController* class to embed and thus gets the dependency on this? 😐

Again, like for the alert, the concrete object is created by the *Presenter*. The *Presenter* therefore has a direct dependency on these controllers to be embedded, as well as the *UIAlertController*.

You could also consider calling the *Factory* by the *Presenter* to create these dependencies. However, that would be a considerable increase in complexity. What would be the benefit?

There would be not benefit, because you will not test the Presenter with unit tests. That would not be practical because of the needed view lifecycle. You will need to have the full *UIViewController* be running including animation, ViewController presentation, etc. If you want to have tests there, then UITests would be more appropriate.

So you do not necessarily have to replace the dependency on this embedded controller with a mock, because you won't use something different from the controller. Therefore, keep it simple, stupid! 😝

Conclusion

The Single Responsibility Principle states that a class should do only one thing. What this "thing" is, is somewhat subjective, but, in my opinion, it makes sense to outsource the interaction into an *Interactor* and the presentation into a *Presenter*. The *View* then only manages the view hierarchy and *Logic* just contains the business logic.

The *Presenter* and *Interactor* are usually not tested by *UnitTests*, but rather by *UITests*. The *View* as well, although you can also test it via unit tests using snapshot testing. But since the *View* is not dependent on anything else, that's not a problem either.

An abstraction through interfaces is only necessary for the *Presenter* because this has to be referenced by the *Logic*. Since the *Logic* should be placed under *UnitTest*, it is important to be able to mock the *Presenter* thanks to the interface.

What's left are *Workers*. Workers are the henchmen of the *Logic* and are, of course, also abstracted via interfaces. Thus, the *Logic* is not directly dependent on the *Workers*.

Decoupling is everything! 😎

IOS Architecture Decoupling Display Programming

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

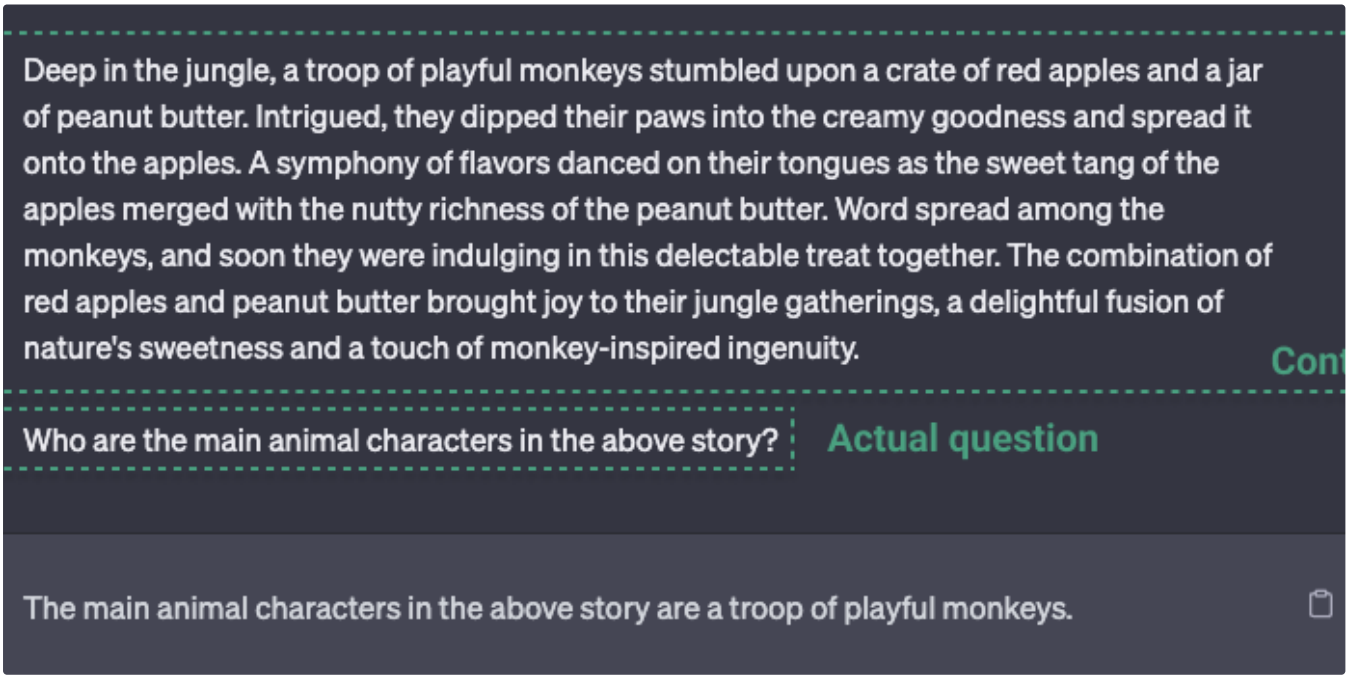
Benoit Ruiz in Better Programming

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

-- 260



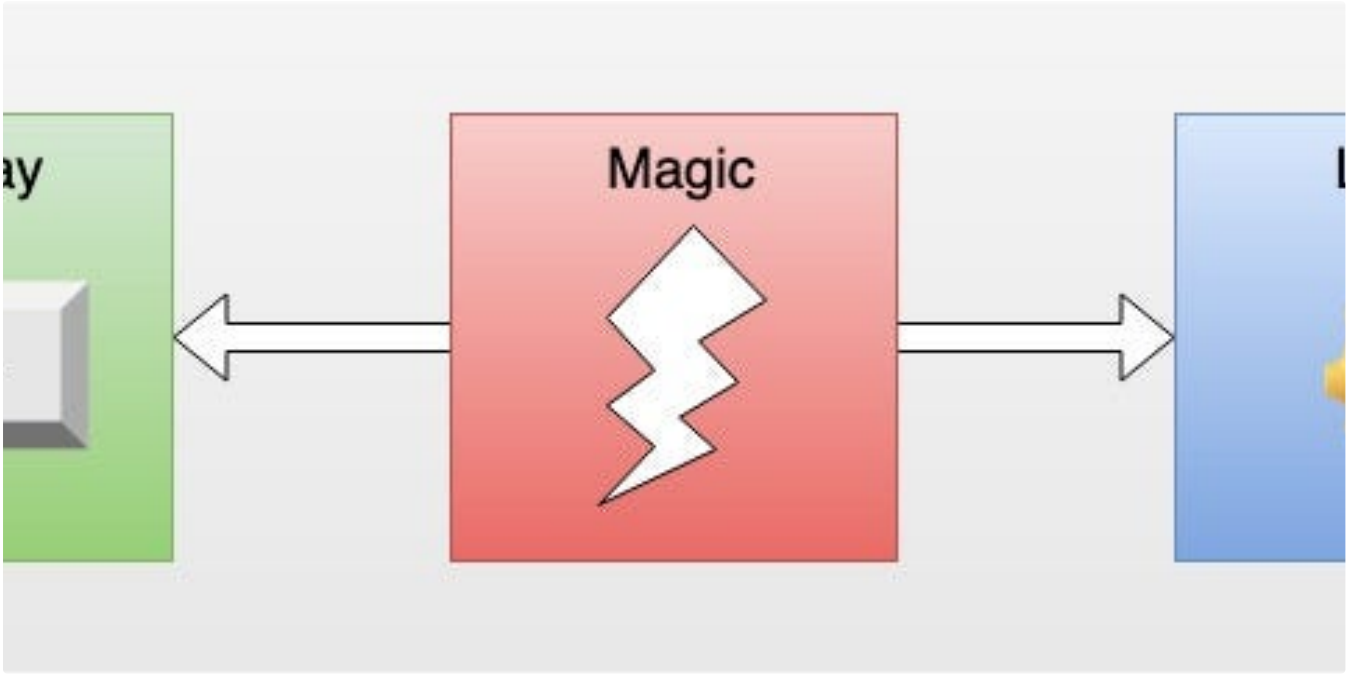
Sami Maameri in Better Programming

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!

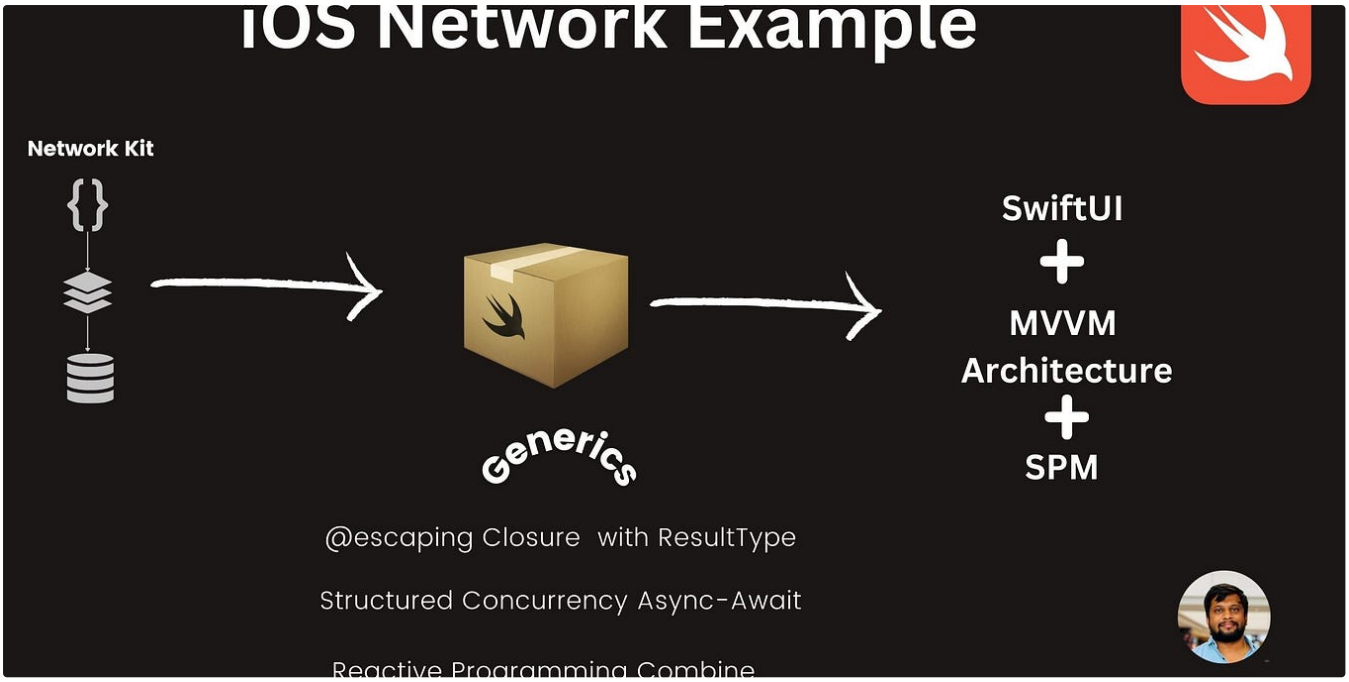
· 11 min read · Aug 16, 2019

-- 2

See all from Sven Korset

See all from Better Programming

Recommended from Medium



Kanagasabapathy Rajkumar

Building Modular iOS Apps: A Guide to SPM, MVVM, SwiftUI, and Combine/Async-Await

Seamless Integration: A Step-by-Step Guide to Incorporating Swift Package NetworkKit for Robust iOS App Development

6 min read · Jan 4

--

Wahyu Alfandi

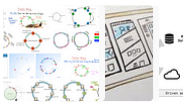
A Beginner's Guide to Clean Architecture in SwiftUI: Building Better Apps Step by Step

Photo by Lala Azizli on Unsplash

9 min read · Aug 24, 2023

-- 7

Lists

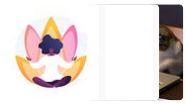


General Coding Knowledge

20 stories · 870 saves

Coding & Development

11 stories · 421 saves



Stories to Help You Grow as a Software Developer

19 stories · 776 saves

ChatGPT

21 stories · 435 saves

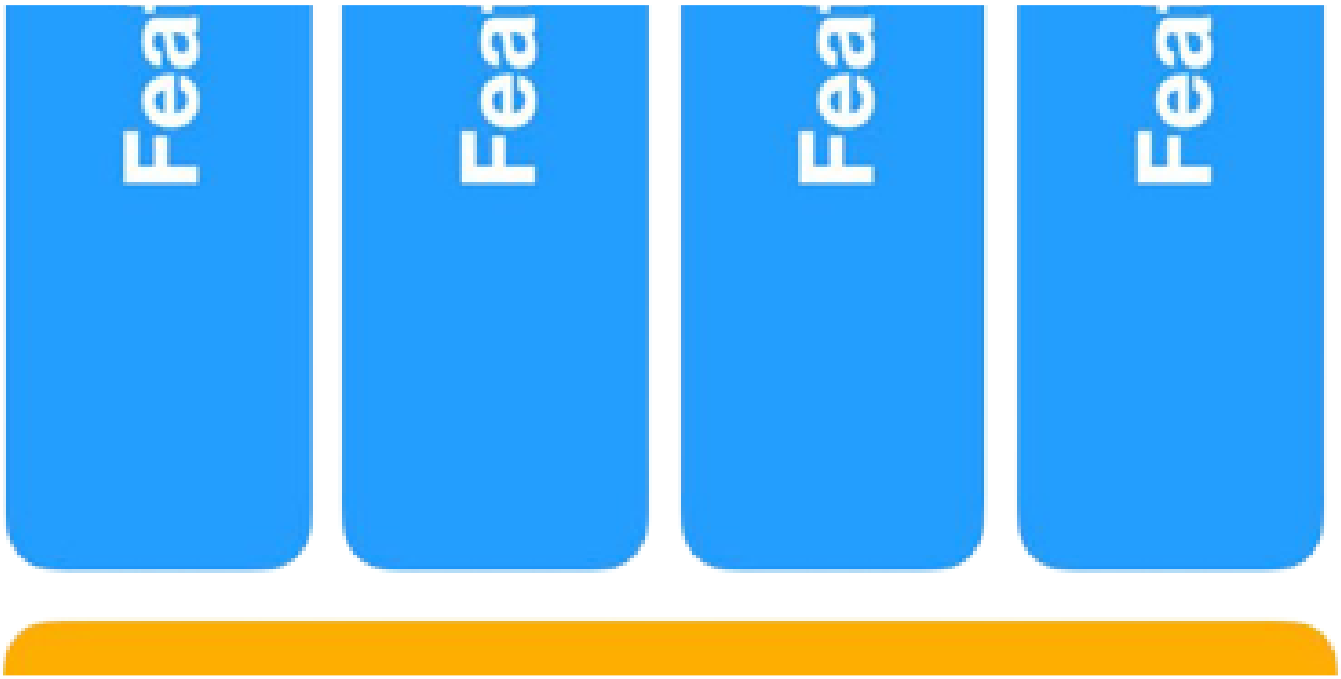
Jacob Bartlett in The Swift Cooperative

Async Unit Testing in Swift

Write robust and maintainable software using modern language features

· 47 min read · Aug 22, 2023

-- 2



Artem Kvasnetskyi

IOS Microapps Architecture—Part 2

This is the second part of a series of articles on Microapps architecture. If you missed the first part—you can read it here. It will...

17 min read · Oct 6, 2023

-- 1

Thomas Ricouard

Migrating a simple structure from UserDefault to SwiftData

The iOS 17 SDK introduced a powerful new tool to save and restore user data easily, SwiftData.

4 min read · Sep 22, 2023

 --  

Nindya Alita Rosalia

Introducing MVVM in SwiftUI: How to Easily Implement MVVM in SwiftUI 🧐

👋 Introduction/Overview

14 min read · Aug 27, 2023

-- 6

See more recommendations