✦ Member-only story

PIECES OF A SCALABLE IOS APP ARCHITECTURE

An Example of a Scalable iOS Project, Part 2

The DemoApp-Project's (DAP's) Scenes explained

Sven Korset · Follow

Published in Better Programming

10 min read · Aug 16, 2019

In my article "The perfect iOS app architecture", I introduced my own architecture and explained the concept of *Scenes*. With Part 1 of "An example of a scalable iOS project", I described the general project's structure. You might wish to read that, before proceeding with the details of how Scenes are implemented. 🧐

The DemoApp Project (DAP) exists only to present certain aspects and concrete solutions to common problems. In my opinion, it is more understandable if the code is as condensed as possible, reduced only to the problems and their solutions with as little boilerplate code as possible.

Tutorials that end up with a "useful" app will swell up the code and make it harder to find the specific "A-ha" moments. That's why I tried to compress those moments and reduce the overhead in the DAP. I hope I succeeded. 😅

Scene0

First, when you start the app, you get to *Scene0*. This is just a splash screen with a corresponding label.

Scene0

Scene0 is the simplest scene and shows the structural composition of a scene, including *Dependency Injection*. Start by investigating this scene to get a good overview of the architectural structure.

In *Scene0*, it is shown how best to update any *InternalSettings* at app start. For new app versions, it often happens that something changes internally, which then has to be updated, e.g. a database or Core Data schema or some internal settings.

The *Logic* not only has to manage the update in the background but also initiate a fade animation in the *View* via the *Presenter* and to time it off before automatically going to *Scene1*. This timing can be tricky, and a possible solution is presented here.

Let's dive into code! 🤠

First, the business logic gets triggered by the `viewDidAppear` in the *Scene0VC*.

Scene0VC.swift

The *Logic* is responsible for calling the `updateSettings` method, ensuring that the splash screen's minimum display time is met and the current scene then disappears with the initiation of the *Scene1* navigation.

Scene0Logic.swift

For the timing, a *DispatchGroup* is used, but, of course, there are also other possibilities for solving this.

More interesting might be lines 18 and 29 where `testFlags` come into play. These flags come from the *Config.plist* file, and the `noSplash` entry shows how to change the logic for testing purposes, e.g. when running UI tests, you normally don't want to wait many seconds for the splash screen to disappear. So, simply run your UI tests with such a config to save some time.

The *Presenter* is implemented straight forward. Simply hides the splash and informs the *Logic* when completed. So the *Logic* only starts the hiding process, but the

process itself is managed by the *Presenter*.

Scene0Presenter.swift

The *Navigator* is also simple. It creates the *NavigationController* and the next *Scene1* and presents them. However, creating the *Scene1VC* is done by the *Factory,* which is accessed via the `dependencies`.

Scene0Navigator.swift

That's all for this specific use case. 😊

The *Interactor* does nothing because this scene doesn't need to map any user interactions to business logic use cases.

The View is very simple with only a single `titleLabel` to set up and center on the screen. However, it's worth mentioning that the label is created programmatically, as are the constraints. For easy constraint creation, the 3rd-party library "Anchorage" is used. It's a neat little library I can highly recommend.

Scene0View.swift

The reasons why the programmatic approach is better can be read in my article "Decoupling Display and Logic in iOS". However, in this example, you can already see that a `let` can be used for the `titleLabel` instead of a typical *Outlet* `var` which leads to a more correct interface. The label styling and setup can be done in one place and in addition with constants. By the way, as you can see you can review the whole view because it's simple Swift and not cryptic XML. 😉

The scene is set up in the *ViewController*. So, let's have a closer look at *Scene0VC*.

Scene0VC.swift

The *VC*'s `init` method in line 14 requires the dependencies being injected. Classical dependency injection. 😄

The first parameter is a `SetupModel.Scene0`, which is a scene-specific parameter. In this case, it's an empty struct, but other scenes (like *Scene2*) might expect some values for being able to operate properly.

The second parameter is an `Act1DCInterface`. This is a dependency container holding references to all dependencies, like the *InternalSettings* or the *Factory* used later by the *Navigator*.

The *Navigator* instance is created in the `init` method. Again, any dependencies are injected right away.

The same is true for the *Logic*, but this time with an own dependency container wrapping the other scene components up.

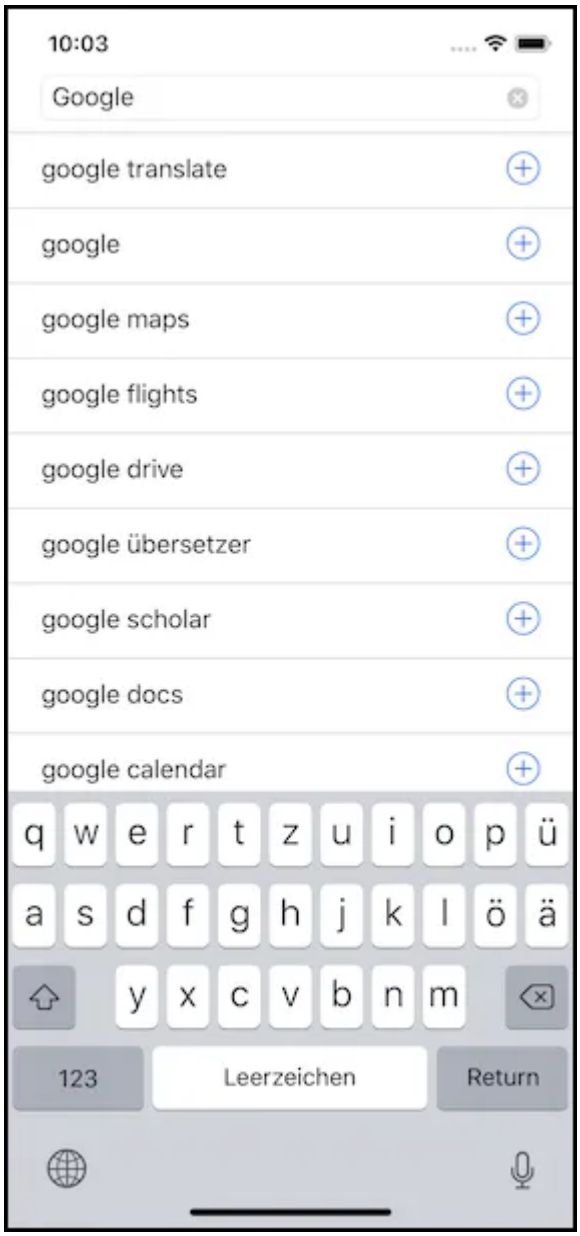
After the *VC*'s call to `super.init()`, the other components can receive a reference to the *ViewController*. This is done via property injection.

When the *View* then gets loaded via `loadView` in line 30, the `Scene0View` gets created and assigned. The presenter also gets a reference to the view, and the `Interactor` is created.

That's all the magic behind the scene. The other scenes work similarly. Do you already see the pattern? 😄

Scene1

The majority of apps use *TableViews* or *CollectionViews* and send any server requests. That's why this should be shown in *Scene1* as an example.



Scene1

Here the user can search for specific words and receive suggestions via Google's autocompletion feature.

If the user presses on the blue plus icon, the suggestion is taken over into the text field, which triggers a new search. On the other hand, if the user selects an entry, the app will navigate to *Scene2*.

The server requests are executed by a corresponding *ServerWorker*, which is located in an embedded framework and injected as a dependency.

In this scene, the interaction between *Display* and *Logic* via *Interactor* and *Presenter* is shown, as well as the transition via the *Navigator* and the communication with *Workers*. So, this actually shows a complete scene as a whole.

The *VC*, *Navigator* and *View* are self-explanatory. There is not much difference compared to *Scene0*. Yeah, a *TableView* is created in the *View* and its *TableController* in the *VC*, but that's really simple stuff. 😪

Scene1VC.swift

The *Logic* has some more use cases. Therefore, there are more methods available to call, but that's also nothing special. *LogicState* may be interesting.

Scene1LogicState.swift

It's a simple struct that holds some values and thus the state for the scene's *Logic*. It is initialized during *Scene1Logic*'s `init` and the use cases then read and manipulate these values.

The *Logic* itself doesn't need to have any further properties for holding state logic. It can be done in this struct. In addition, you could quickly persist the logic's state by simply serializing the struct if needed. Data and logic are now more separated.

The *Presenter* again does little more than show data on the view.

Scene1Presenter.swift

In the first case of the `suggestionList(suggestions:)` method, the `tableController` needs to be told to update its view. A *TableController*, therefore, works as a *Presenter*. It is responsible for updating its view, a *TableView* in this case. On the other hand, the *TableController* also works as an *Interactor* because it maps user actions to the *Logic*'s methods. However, its responsibility is clear: taking care of the *TableView* and nothing else.

Let's get back to the *Presenter*, which presents an alert in the `serverError` method. You can read why it's okay to do it here in "Decoupling Display and Logic in iOS".

The last method, `searchText`, is also self-explanatory. Just update that input field's text.

Now to the *Interactor*! 🤠

Scene1Interactor.swift

Yikes, Rx! 😱

Yes, *RxSwift* and *RxCocoa* are used here. You can, of course, write your own bindings, like registering for actions and then calling the appropriate logic method. However, Rx makes it simpler and gives you more power. At least most of the time. You know, power and responsibility and so forth. 😅

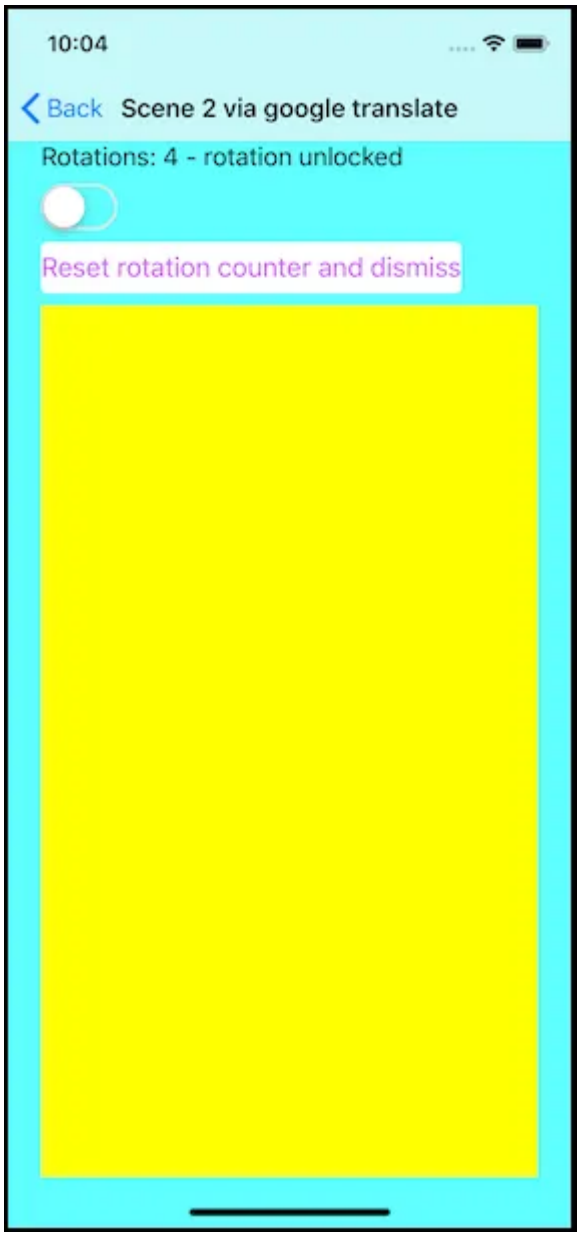
I don't want to go into any Rx details. I'm not a Rx pro. I think the code is commented well enough to understand what each call does.

In essence, the first block maps any text changes in the search input field to the logic's `searchForText` method and the second block maps any `tableView` scroll events to the logic's `dismissKeyboard` instruction. I bet you can guess what these logic methods do. 😊

Well, that's all for this scene. Feel free to investigate further into the *TableController*'s code and how to set up the cells.

Scene2

Scene2 is a typical detail view with some extras.



Scene2

If the user has selected an entry in *Scene1*, then this should be passed to *Scene2*. Passing parameters to scenes is usually fairly common.

How to pass values back to the previous *Scene1* is also shown with *Scene2*. Here the number of device rotations across all *Scene2* instances is summed up. The number is thus always passed to *Scene2*, modified and later returned back to *Scene1*.

A usual transition can be done via the *Back* button in the *NavigationBar* but also manually with the *Reset* button in the *View*.

The switch is there to prevent the view to rotate when turning the device. And then there is an embedded view from a child controller. That's the yellow area.

To have a better understanding it's best to examine the individual aspects directly in the code. 🧐

Let's start with the *VC*.

The setup is the same as in the other scenes. For specific *UIKit* events, the *VC* has to report them to the *Logic*, for example, to inform the *Logic* via the `displayRotated` method when `viewWillTransition(to:, with:)` has been called on the *VC*.

Scene2VC.swift

The same goes for the `viewWillDisappear`, where the *Logic* should update the parent scene because the scene is about to move back to the previous.

The computed property `shouldAutorotate`, on the other hand, needs a state to be returned. The *VC* should not hold any states. That's the responsibility of the *Logic*. However, the *VC* should also not call a method on the *Logic* only to get a value back and to propagate it here. Therefore, the *VC* can access the *Logic*'s `state` struct directly, which is defined as:

```
private(set) var state = Scene2Model.LogicState()
```

That's not ideal, but it's simpler than querying all values via methods and it's not really an issue because the `state` property is read-only and not part of the *Logic*'s interface. So, no other component can access it but the *VC* because it holds a reference to the *Logic* itself, not to the *LogicInterface*. 😬

Speaking of the *Logic*, it again only manipulates some state values and calls some *Presenter* and *Navigator* methods. One method to look closer at is the `updateParentScene`, which, well, updates the parent scene with data.

Scene2Logic.swift

This is how to pass values back to the previous scene. The Logic doesn't know anything about the previous scene. All it knows is that it has to call a `callback` method with some `Scene2Result` values.

The callback method comes from the `setupModel`, which is a struct of the type `Scene2`. So what does this look like?

Scene2Setup.swift

The struct provides values that are needed by *Scene2*. The `headline` and `numberOfRotations` are values to pass and present by this scene. However, there is also the `callback` property, which points to a closure that is then called for passing values of the type `Scene2Result` back.

`Scene2Result` is a simple struct that only holds the `numberOfRotations` to deliver back, but could also be more complex. So, one of the parameters for initializing *Scene2* is a closure for returning values back to the previous scene, whichever that is.

This is then prepared in *Scene1Logic*.

Scene1Logic.swift

Here the *Logic* of *Scene1* passes the values and the callback closure to *Scene2*. The closure then updates only the *Logic*'s state, but that's essentially how to pass any values back and forth. 😄

Last but not least, let's look at the embedded view controller.

Scene2Presenter.swift

This is an excerpt of the `updateView(model:)` method in the *Scene2Presenter*. When there is no child controller already added, a new *UIViewController* is created and added as a child to the scene's *ViewController*, and its view gets embedded into the scene's *View*.

Scene2View.swift

The *Scene2View* is responsible for manipulating the view hierarchy. That's why this `addEmbeddedView` method is in the view. It only adds the subview and anchors it.

But, wait a minute, why is the *Presenter* creating the *UIViewController* and not the *Factory*? 😉

Good point! I have already explained the reason for this in my article "Decoupling Display and Logic in iOS". Of course you're free to use the *Factory* to create this dependency. That would certainly be cleaner.

However, the *Presenter* will not be unit tested but UI tested. It will not be tested detached from the other components. You will not need to mock this

UIViewController dependency. The same goes for any formatter the *Presenter* might need or any other dependencies. You will rarely inject anything different from these exact instances. Why then inject at all? Only for the sake of injecting? 🤔

Like everything, it's a trade-off. Does the benefit justify the work? Sure, we want clean code, but we also want maintainable code, and less code means less to maintain.

In this particular case, I think we don't need to inflate the *Factory* and can put these dependencies directly into the *Presenter*. However, that has always to be weighed up depending on the situation, and I might change my mind in the future.

So, don't take it as an undeniable dogma.

> Don't take anything as undeniable. Time can change everything, even dogmas and best practices!

With that, everything is said. 😄

Conclusion

The DAP should represent common apps in a compressed form.

Maybe that does not look like much, but according to the Pareto principle, 20% of effort is enough to solve 80% of problems. These are, in my opinion, the most common problems a typical app faces and should cover the said 80%.

However, that's of course only the bare bone. The rest will then be your creative task as a developer. 😄

Btw, this is only one of many articles from the "Pieces of a scalable iOS app architecture" series. Maybe you want to read more of my undeniable wisdom? 😂

IOS Swift Demo Architecture Projects

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

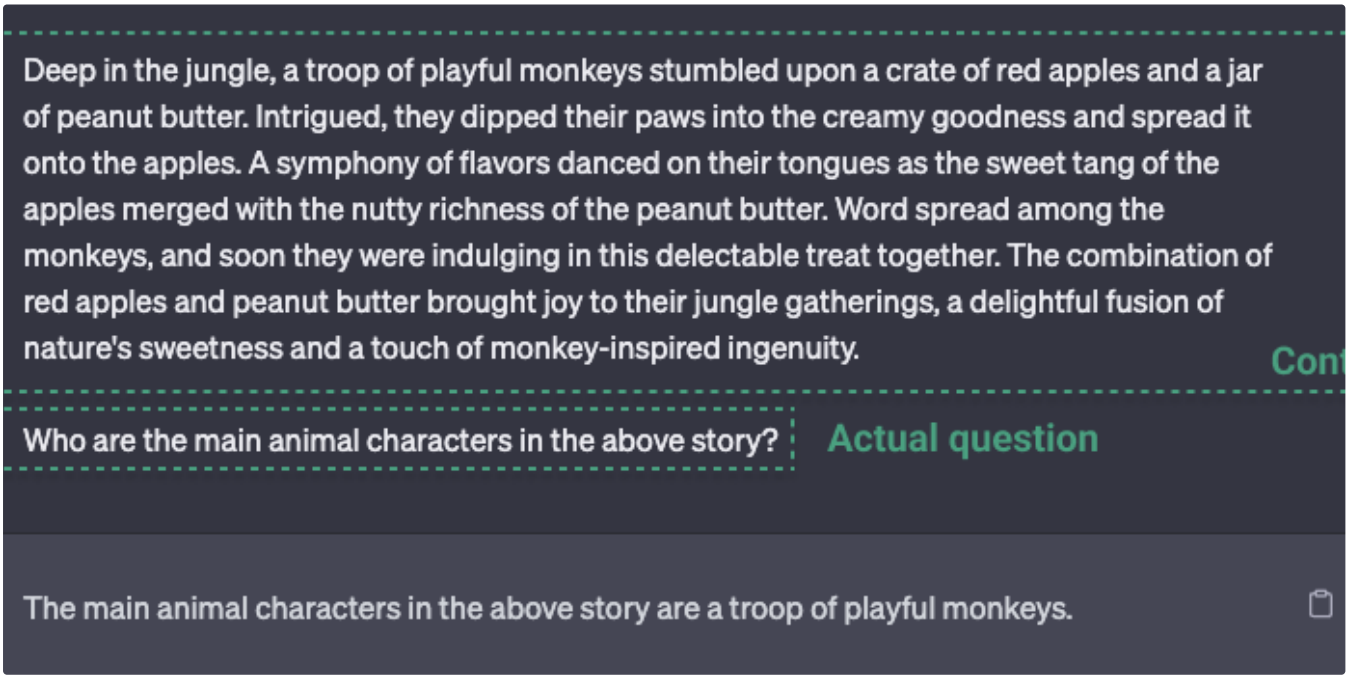
Benoit Ruiz in Better Programming

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



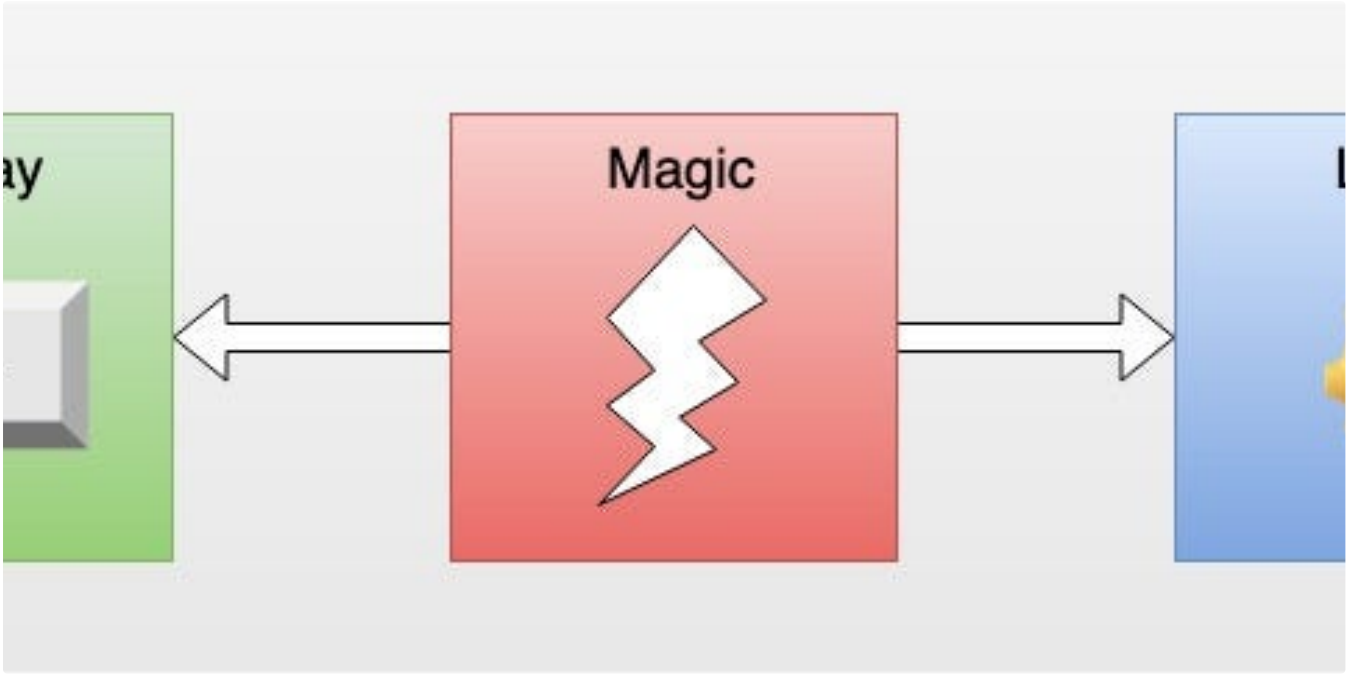
Sami Maameri in Better Programming

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

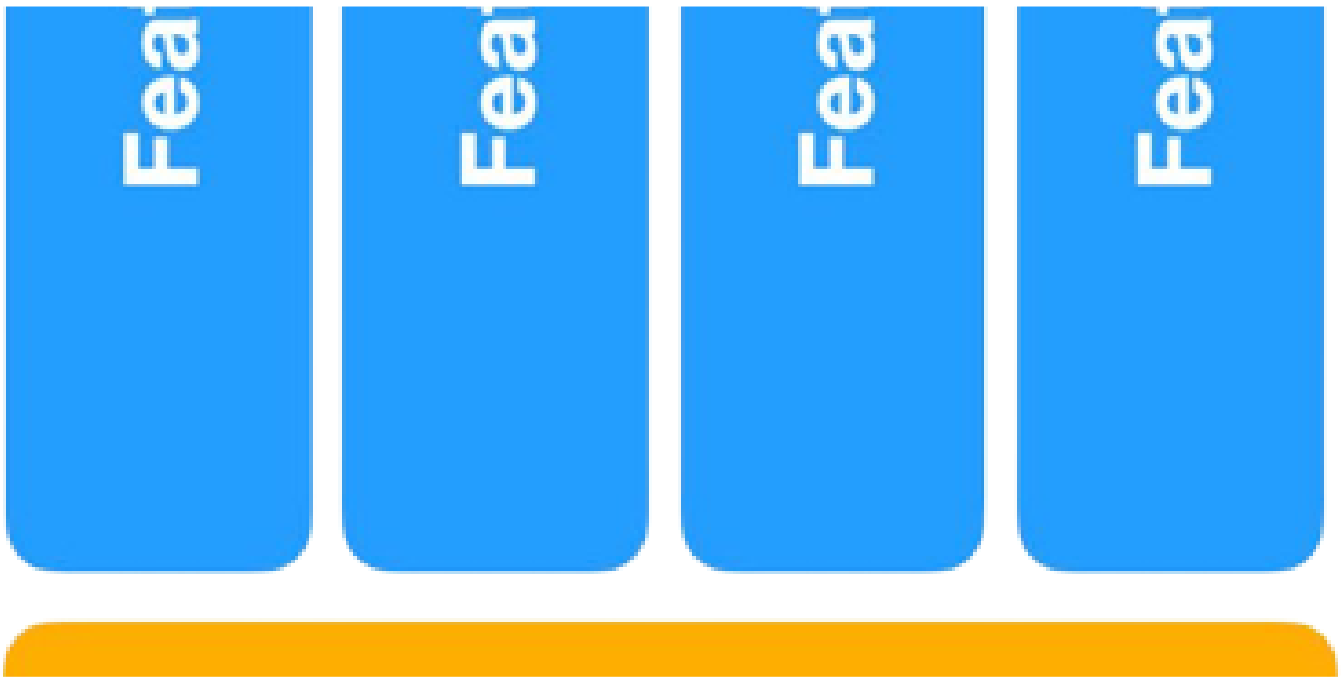
Ben Myers in Better Programming

Create a Scalable SwiftUI MVVM Project

Make a well-documented, organized, scalable SwiftUI project using the MVVM architecture that will make your code reviewers say “wow”.

7 min read · Sep 15, 2022

-- 3



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...

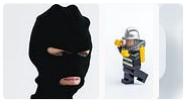
17 min read · Oct 6, 2023

-- 1

Lists

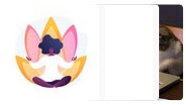
Apple's Vision Pro

7 stories · 51 saves



Tech & Tools

16 stories · 138 saves



Stories to Help You Grow as a Software Developer

19 stories · 776 saves

Icon Design

36 stories · 215 saves

Zeba

MVVM in iOS Swift

Model-View-ViewModel (MVVM) is a design pattern widely used in iOS app development to create clean, maintainable, and testable code. MVVM...

4 min read · Sep 6, 2023

-- 1

Ario Liyan

iOS Application Scene Delegate VS App Delegate(A talk about Life cycle)

Preface

6 min read · Aug 8, 2023

-- 1

Muhammad Rezky Sulihin

SwiftUI Push Notifications with Firebase: A Step-by-Step Guide

If you're a developer looking to enhance your app's feature and user engagement, push notification is one of the key.

8 min read · Aug 12, 2023

--

Ankur Kesharwani in LazyPay Engineering

How moving from SPM to CocoaPods improved our developer productivity 🚀

Code modularization is an essential aspect of software development that aims to improve code's maintainability, scalability, and...

11 min read · Aug 8, 2023

-- 5

See more recommendations