Architecting SwiftUI apps with MVC and MVVM

Matteo Manferdini

A SMALL PROMISE FROM YOU

It took me many years of research and experience, and hours of work to create this guide. But I am happy to offer it to you completely free of charge.

In exchange I ask you for a small promise.

If you will think I did a good job, I ask you to share this with someone you think can benefit from it.

That way, I can spend less time looking for people I can help and more producing great free material. You will find some links to share this guide at the end. Thank you.

Let’s start.

WHO IS THIS GUIDE FOR?

I wrote this guide for both beginners and experienced developers.

If you are just starting to make iOS apps, most of your questions probably revolve around how to structure your code. You might already know some of the features of iOS, but you can't bring them all together in a well-structured app. So, you are going to get the most value out of this guide.

And even if you already have some experience, I am still sure you can get a lot of value out of this guide as well. Online articles, books, and courses rarely approach iOS development from a structural point of view. App architecture is always an afterthought, and many developers get bad habits. This guide will hopefully correct those.

I have been teaching the concepts you'll find in this guide for years, and I have recently updated them for SwiftUI apps. I have noticed that many experienced developers don't know them. At the beginning of my career, I didn't know them either and made most of the mistakes I now see in many apps.

THE BLUEPRINT OF IOS APPS

WHY DO WE NEED DESIGN PATTERNS AT ALL?

Whether you are building an entirely new app or adding features to an existing one, the question is always the same: how should I structure my code?

When you learn to program in Swift, you get a plethora of tools: functions, structures, enumerations, classes, protocols, and many others. What you don't learn, though, is how to use them and for which purpose.

There are many tasks you need to consider when building an iOS app. The most common ones, which you find in pretty much any app, are:

- Representing data.
- Building the user interface.
- Implementing the app's logic.
- Storing data on the disk.
- Connecting to the network.
- Reading data from the device sensors.

There are many ways to address these points, using the constructs of the Swift language. But that does not mean that every approach is correct.

My favorite easy-to-understand metaphor is building a house. While there are many styles you can follow, all houses have the same structure: foundations, walls, doors, windows, and a roof. It does not matter if a house was built yesterday or 2.000 years ago. Its blueprint remains the same because it’s the only one that makes sense. If you deviate, your house will fall apart.

The same applies to software development. You can’t structure your code in any way you want. Only a set of predefined blueprints makes sense.

Some developers like to say that the architecture of your app depends on the specifics. I disagree. Sure, you have some flexibility, but if you take it too far, you will get a house that collapses under its weight.

In software development, these blueprints are called *design patterns.* They emerged because developers came to standard diagrams that work for any software. They did the work for you, so you don't need to go through the same process of trial and error.

There are many *architectural design patterns* that tell you how to structure your apps, all with different, esoteric names. Following any pattern is better than not following any pattern at all. But, if you look at them closely, you will notice they all have the same structure, like houses.

That's not a coincidence. The most popular pattern is the *Model-View-Controller* pattern or MVC. All other trends, including MVVM, MVA, MVP, and even VIPER, come from MVC.

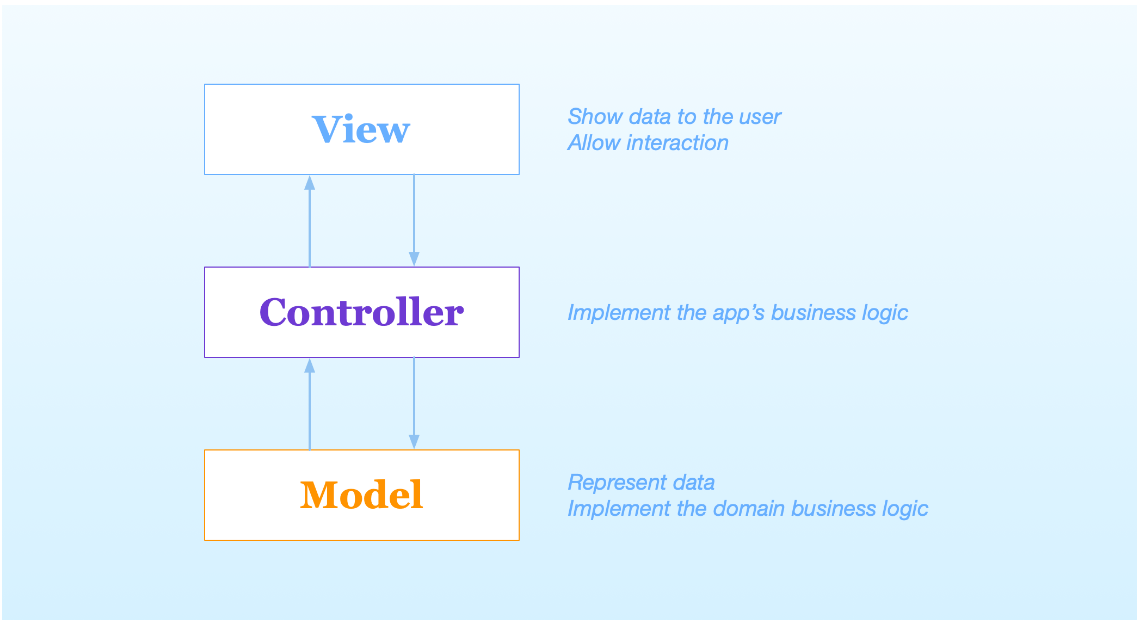
So, that’s where we will start.

THE VANILLA MVC PATTERN AND ITS IOS EVOLUTION

The MVC pattern divides the structure of any app into three layers, each with its precise responsibilities:

- The *model* layer represents the data of the app and encapsulates the domain business logic.
- The *view* layer is the user interface of the app. It shows data to the user and allows interaction.
- Finally, the *controller* layer acts as a bridge between the other two layers and implements the app’s business logic.

(I'll explain you later the difference between the *domain business logic* and the *app's business logic*).



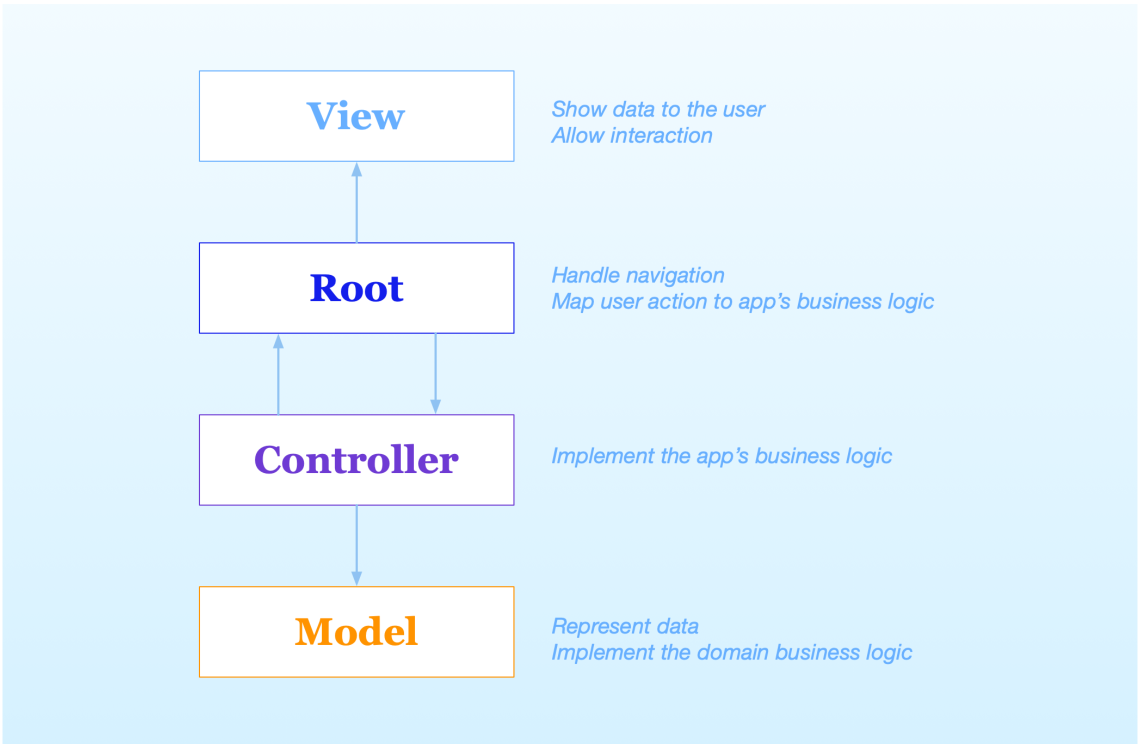
In just a paragraph, I gave you a blueprint you can follow to build any app. Don't worry about the details. We will spend the rest of this guide exploring code. What is important is that now you know where, approximately, each new piece of code goes. These roles will guide every structural decision you make.

What I showed you above is the "vanilla" version of the MVC pattern. For decades and to this day, it is used by developers on many platforms. So it's not a surprise that it works for SwiftUI too.

We don't need to stick to the vanilla version of MVC, though. We can make slight modifications and, indeed, the diagram above needs some refinement.

iOS apps are made of many "screens" through which the user can move. That requires code to handle navigation and to map the user actions into the app's business logic. As an app grows, it becomes clear that all that code concentrates at the meeting point of the view and controller layers.

To avoid large intricate code and keep our app well structured, we can add an extra layer to take those responsibilities. In UIKit, that was the role of *view controllers*. In SwiftUI, that code does not magically disappear, so we need a similar layer. I named it the *root layer* because it's composed of views that sit at the root of the view hierarchy for an entire screen.


View
Show data to the user
Allow interaction
Root
Handle navigation
Map user action to app's business logic
Controller
Implement the app's business logic
Model
Represent data
Implement the domain business logic

Why the arrows follow those directions will be clearer when we will look at the code for our app. What is important, for now, is the four layers and their responsibilities.

SOME WORDS ABOUT MVVM

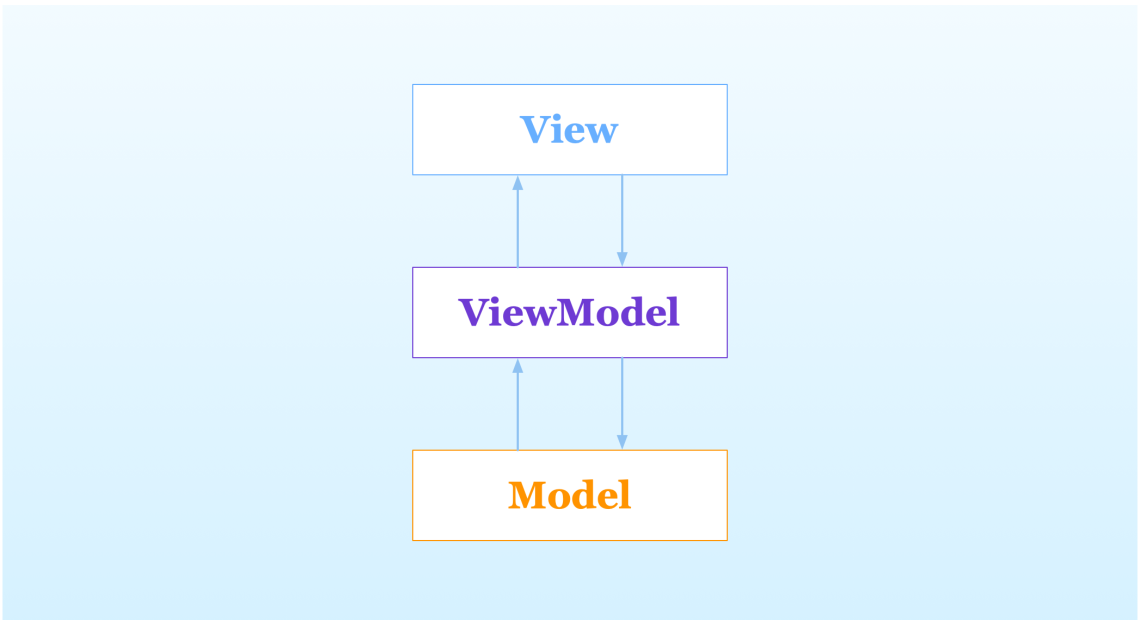
Before we go on and see how to build a SwiftUI app following the MVC blueprint, I have to spend a couple of words on the *Model-View-ViewModel* pattern and address some misconceptions.

In short: the MVC and MVVM have always practically been the same design pattern, with a few nuanced differences. In SwiftUI, those go away.

That's is all you need to know, so you can skip this section.

For those who are unconvinced, or that want an explanation, here are some more words.

MVVM is a design pattern created by Microsoft architects to simplify event-driven programming. If you look at its diagram, you will see that it has the same structure of MVC.



You don't need to be a genius to see that the *view model* layer of MVVM is the *controller* layer of MVC. The only difference is that the view and view model layers are connected through a *binder,* using some declarative data binding technology instead of standard programming practices like it happens in MVC.

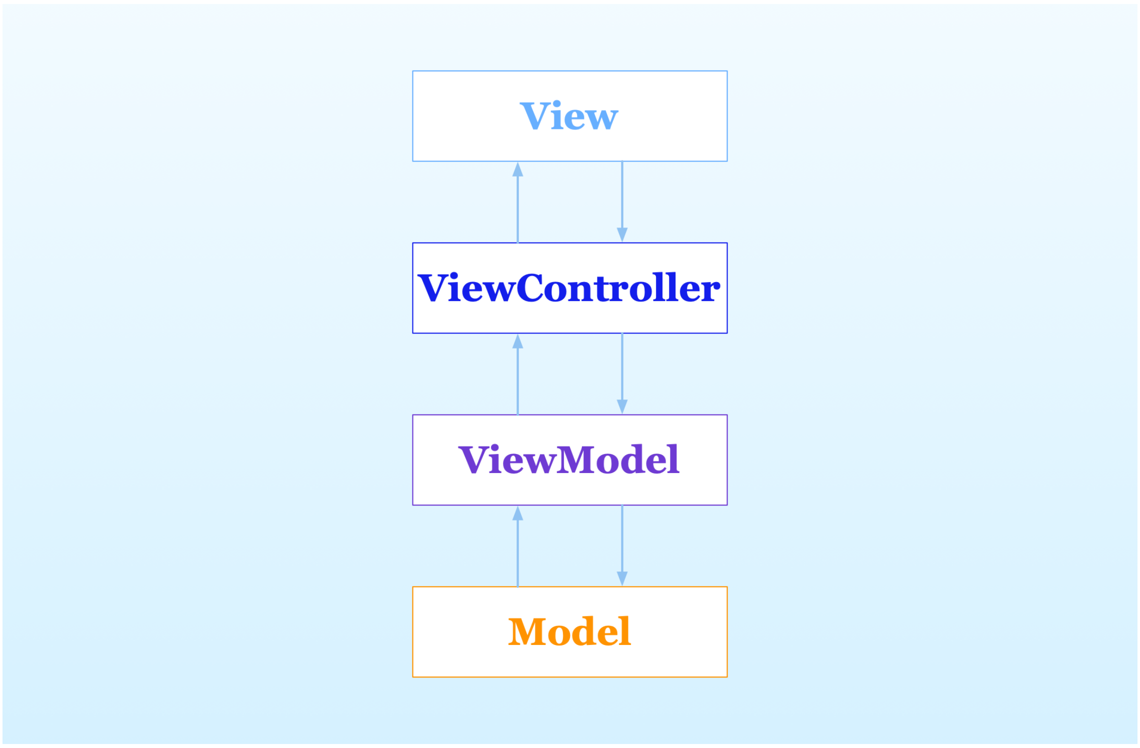
That is, or better, was, the only difference. In UIKit, MVC apps used the lifecycle events of view controllers, delegation, and callback closures to transfer data between the view and controller layers. MVVM apps, instead, relied on open-source event-driven *functional reactive programming* frameworks like *RxSwift*.

That difference goes away in SwiftUI. The only way to transfer data to the view layer is by using *observable objects* with **@Published** properties. SwiftUI views connect to these objects using the **@ObservedObject** and **@EnvironmentObject** property wrappers.

This is an event-driven mechanism. Every time a published property is updated, the user interface is automatically refreshed, removing the only difference between the MVC and MVVM patterns.

But there is more.

UIKit apps were managed by view controllers, which were an unavoidable part of the framework. So, the proponents of MVVM swept view controllers under the rug, placing them inside the view layer and pretending they didn't exist.



We have come full circle back to the diagram of the extended MVC pattern. The arrows are slightly different, but is also depends on which chart you read. In any case, they are far less important than people think.

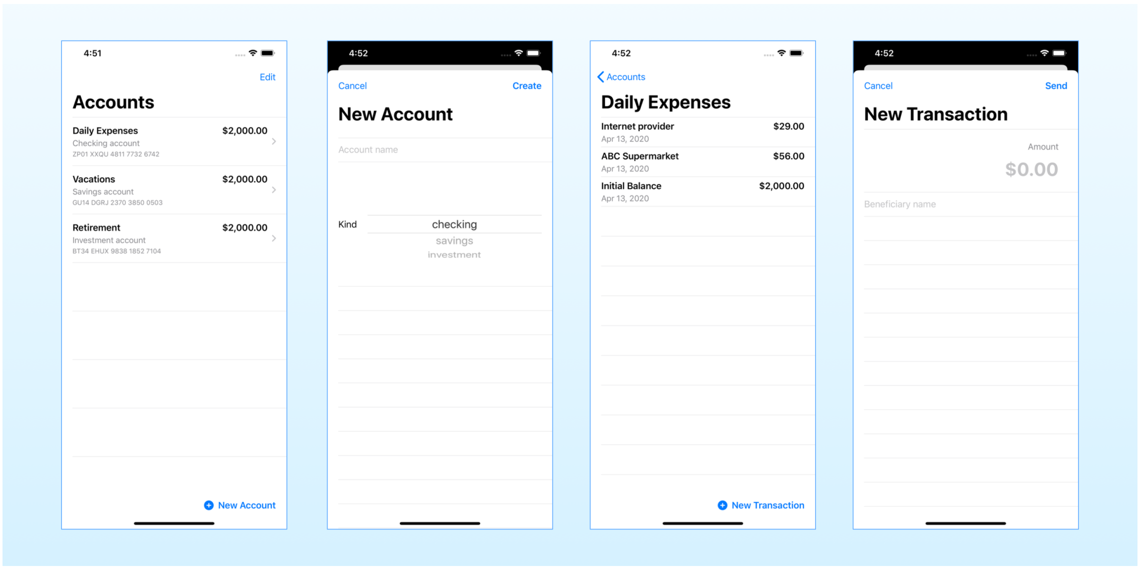
For these reasons, I will only talk in terms of MVC, which is the original pattern. Its variations are irrelevant, and sometimes, even worse.

MANAGING DATA AND BUSINESS LOGIC IN MODEL TYPES

We will now go through a concrete example to show you how to use the abstract ideas of the MVC in a real app.

We will look at a simple banking app that manages multiple accounts. The user will be able to create new accounts in the app and perform transactions in each account.

This is how the final app looks like.



Daily Expenses $2,000.00
Checking account
ZP01 XXQU 4811 7732 6742

Vacations $2,000.00
Savings account
GU14 DGRJ 2370 3850 0503

Retirement $2,000.00
Investment account
BT34 EHUX 9838 1852 7104

Internet provider $29.00
Apr 13, 2020

ABC Supermarket $56.00
Apr 13, 2020

Initial Balance $2,000.00
Apr 13, 2020

I didn't spend too much time on a pretty design because it's not essential. This will also keep the code as small as possible, so we will be able to focus on the app's structure without extra distractions.

I won’t explain the code in the app line by line either, as I usually do in my articles and courses. Instead, I will show you the relevant sections of the complete code, which you can download here. The point of this guide is to learn how to structure apps, not learning every little detail of Swift and SwiftUI.

If you are interested in in-depth coverage of SwiftUI and more advanced techniques for creating full-fledged apps with data storage and networking, I have a full course called The Confident iOS Professional that covers all that, and much more. I open the course regularly only to the subscribers of my email list, so be sure to stay in it (you can join here).

REPRESENTING THE APP'S DATA

The model layer is the foundation of our app. Many developers like to start from the app's user interface, and that's a fine approach. But I often prefer to start with model types.

The model layer is independent from the rest of the app. Model types only focus on data and don't need to worry about storage, networking, or the user interface. This makes them easy to write and test.

Despite their simplicity, a lot of the code what powers a well-structured app resides in the model layer. A bug in the model layer can influence any or all parts of your app. On the other hand, a solid model layer will help you significantly reduce the code in different layers of MVC. So it's important to get model types right.

Our app needs to deal with bank accounts and transactions, so we can create a couple of types to handle those.

```swift
struct Transaction {
	let amount: Int
	let beneficiary: String
	let date: Date
}

struct Account {
	let name: String
	let iban: String
	let kind: Kind
	var transactions: [Transaction]
}

extension Account {
	enum Kind: String, Codable, CaseIterable {
		case checking
		case savings
		case investment
	}
}
```

The model layer of an app is better represented using Swift's *value types*, i.e., structures and enumerations. Value types get copied, so each piece of code can deal with a local value that does not get affected by anything else. Parallel code that accesses the same resources is the most significant source of bugs in any software. With value types, we keep our code as independent as possible.

(Side note: I used **Int** for the **amount** property of the **Transaction** type instead of a **Float** because you should never use floating-point numbers for precise numbers, like in financial transactions. Money amounts are usually expressed in cents, using integers).

IMPLEMENTING THE DOMAIN BUSINESS LOGIC IN THE MODEL LAYER

The most significant mistake developers make is putting the *domain business logic* in the other layers of MVC.

The domain business logic is any logic that deals with the domain covered by our app and its rules. In our case, the field is banking. So, all the code that deals with the laws of banking belongs to the domain logic.

Many developers stop at the code above, leaving the model layer too thin. They use model types to only represent the data in their apps. But in MVC, the model layer should take care of much more. Model types should also:

- encapsulate the domain business logic;
- transform data from one format to another.

There is much more to bank accounts and transactions than what we expressed in the **Account** and **Transaction** structures. For example:

- A customer can only open an account by putting into it a €2.000 deposit.
- The balance of an account needs to match the sum of all its transactions.
- Each transaction is final, and it should not be possible to delete it from an account.

Since these rules belong to the domain business logic, their code should also go into the model layer.

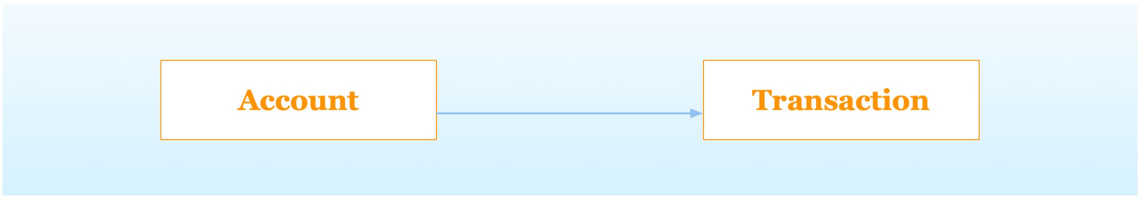
```swift
struct Transaction: Identifiable, Codable {
	let id = UUID()
	let amount: Int
	let beneficiary: String
	let date: Date
}

struct Account: Codable, Identifiable {
	let name: String
	let iban: String
	let kind: Kind
	private(set) var transactions: [Transaction]

	var id: String { iban }

	var balance: Int {
		var balance = 0
		for transaction in transactions {
			balance += transaction.amount
		}
		return balance
	}

	init(name: String, iban: String, kind: Kind) {
		self.name = name
		self.kind = kind
		self.iban = iban
		transactions = [
			Transaction(amount: 2000_00,
				beneficiary: "Initial Balance", date: Date())
		]
	}

	mutating func add(_ transaction: Transaction) {
		transactions.append(transaction)
	}
}

extension Account {
	enum Kind: String, Codable, CaseIterable {
		case checking
		case savings
		case investment
	}
}
```

Our model types are now richer and enforce all the rules I listed above. For starters, the initializer of the **Account** structure sets the initial transaction to €2.000. (I'm simplifying here. The request to create an account would usually go through the bank where the customer would physically bring the sum).

The **balance** is a computed property that sums all the transaction amounts. And the **transactions** property is read-only and can be modified only by the **Account** structure through its **add(_:)** method. This prevents any external code from removing existing transactions.

Our model types also handle data transformation. All the types conform to the **Identifiable** and **Codable** protocols, which allow us to organize, store, and send data. If we had more complex rules for encoding and decoding our data, those would also end inside the **Transaction** and **Account** structures.

Often, all this logic ends inside controllers or, worse, views. That's a mistake. It spreads the business logic across the entire app, often duplicating functionality and making it unmanageable. Keeping it inside model types makes it simpler to write and to test. And since our full app will rely on this business logic, this approach produces fewer bugs.



IMPLEMENTING THE APP'S BUSINESS LOGIC FOR STATE AND STORAGE IN CONTROLLERS

Moving up the MVC pattern, the next layer we meet is the controller layer. Here we also find many responsibilities that developers often put inside SwiftUI views.

While the model layer contains the *domain business logic*, the controller layer contains the *app's business logic*, which includes:

- keeping the state for the entire app;
- storing data on disk;
- transmitting data over the network;
- dealing with data and events coming from the device sensors (GPS, accelerometer, gyroscope, etc.)

STORING DATA AND KEEPING THE APP'S LOGIC ISOLATED INSIDE CONTROLLERS

We will start with saving and reading data on disk, which many apps need. For that, we will use the iOS file system, saving our data in the documents directory.

```
class StorageController {
	private let documentsDirectoryURL = FileManager.default
		.urls(for: .documentDirectory, in: .userDomainMask)
		.first!

	private var accountsFileURL: URL {
		return documentsDirectoryURL
			.appendingPathComponent("Accounts")
			.appendingPathExtension("json")
	}

	func save(_ accounts: [Account]) {
		let encoder = JSONEncoder()
		guard let data = try? encoder.encode(accounts) else { return }
		try? data.write(to: accountsFileURL)
	}
	func fetchAccounts() -> [Account] {
		guard let data = try? Data(contentsOf: accountsFileURL) else { return [] }
		let decoder = JSONDecoder()
		let accounts = try? decoder.decode([Account].self, from: data)
		return accounts ?? []
	}
}
```

The first thing to notice here is that we used a class and not a struct. Controllers need to be shared across the entire app and provide a unique access point, so they need to be objects.

The core functionality of the **StorageController** class is in its **save(_:)** and **fetchAccounts()** methods. The former encodes the data in our model type and stores it in a file. The latter reads that file and decodes its data to restore the model types every time the user runs the app. Since each account contains its respective transactions, this is enough to save all our data.

While this class does not do much, it is crucial to keep its code separated from the rest. In a real-world app, things are rarely this simple. Any code in your app is likely to grow larger with time, so keep responsibilities separated from the start.

KEEPING THE APP'S STATE TOGETHER WITH ITS BUSINESS LOGIC

Pretty much any app needs to store its global state somewhere.

Holding the app's state is another of those tasks that require more code than it might be evident at first glance. Many developers just store the data representing the current app state in an observable object and call it a day.

```
class StateController: ObservableObject {
	@Published var accounts: [Account]
}
```

Whenever you see a type with stored properties but no code, you should be suspicious. While that can sometimes happen, it's not destined to last for long anyway.

Recall that the controller layer must contain the app's business logic, which defines, among other things:

- how to get an IBAN for new accounts;
- where to place new accounts relatively to existing ones;
- setting the current date for new transactions;
- when to save and read data.

Remember: if you don't put the app's logic in a controller, it will spread into your SwiftUI views. That's not the right place. These are global rules that affect the behavior of the whole app. Putting such code inside views would lead to duplication, polluting views with responsibilities they should not have.

First of all, we need a way to generate IBAN codes for new accounts. This is another of those tasks that would typically be carried by the bank. A real app would fetch it from the network, but for simplicity, we will generate IBAN codes in our app.

```
extension String {
	static func generateIban() -> String {
		func randomString(length: Int, from characters: String) -> String {
			String((0 ..< length).map { _ in characters.randomElement()! })
		}

		let letters = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
		let digits = "0123456789"
		return randomString(length: 2, from: letters)
			+ randomString(length: 2, from: digits)
			+ randomString(length: 4, from: letters)
			+ randomString(length: 12, from: digits)
	}
}
```

IBAN codes must follow a precise algorithm, but we don't care here, so I just generated a random string that looks like an IBAN. I put it into a **String** extension because this is still part of the domain business logic.

We can now extend the state controller to perform all the tasks I listed above.

```
class StateController: ObservableObject {
	@Published var accounts: [Account]

	private let storageController = StorageController()

	init() {
		self.accounts = storageController.fetchAccounts()
	}

	func addAccount(named name: String, withKind kind: Account.Kind) {
		let account = Account(name: name, iban: String.generateIban(), kind: kind)
```

```
        accounts.append(account)
        storageController.save(accounts)
    }

    func addTransaction(withAmount amount: Int, beneficiary: String,
        to account: Account) {
        guard let index = accounts.firstIndex(where: { $0.id == account.id })
            else { return }
        let transaction = Transaction(amount: amount, beneficiary: beneficiary,
            date: Date())
        accounts[index].add(transaction)
        storageController.save(accounts)
    }
}
```

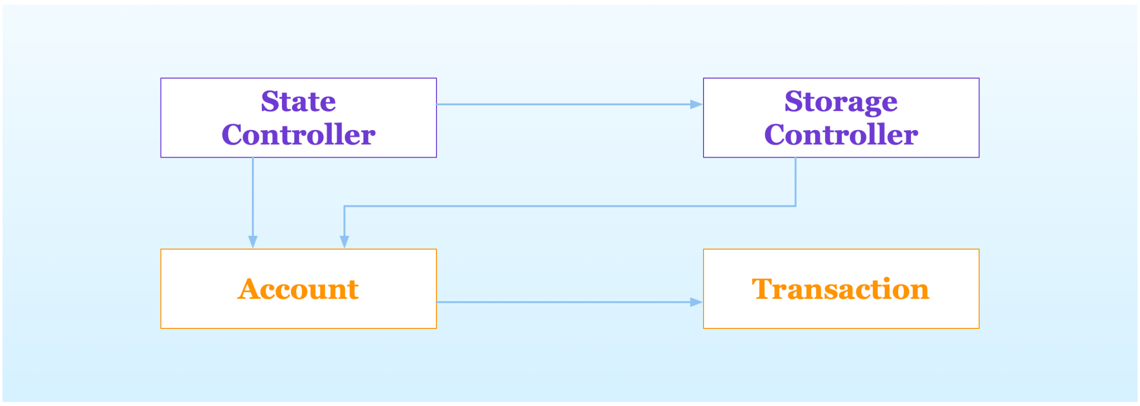
Notice how much code we have now. This is code that would have ended inside views if we didn't gather it here.

The **addAccount(named:withKind:)** and **addTransaction(withAmount:beneficiary:to:)** methods accept parameters with simple types and create **Account** and **Transaction** values internally. This keeps the app's business logic hidden from views, which, in turn, are left managing only the user inputs.

The other important part here is that the **StateController** manages the **StorageController**. That's because the former holds the state of the app, so it is its responsibility to decide when to save or read data.

For simplicity, that happens every time our data changes. This works for small amounts of data since access to the disk can remain relatively quick. In an app with a more significant amount of data, a more sophisticated policy might be required, for example, dispatching disk access to an asynchronous background thread.

Slowly but surely, we are building our house, placing each new type on solid foundations.



SHOWING INFORMATION TO THE USER AND ENABLING INTERACTION THROUGH VIEWS

We will skip the root layer for now, and jump to the view layer. The reason is simple: root views are a bridge between controllers and views, so it's easier to implement them when the other two exist.

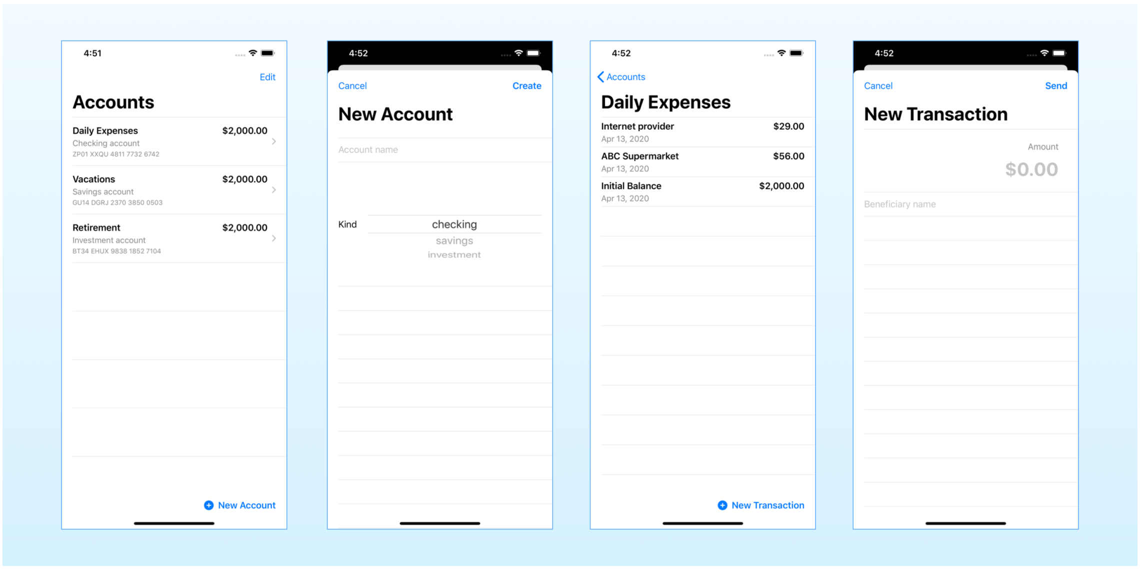
The view layer has only two responsibilities:

- showing data to the user;
- allowing user interaction.

Unlike what happened in other layers, here it's not important how many, but how few responsibilities there are in views. The mistake here is putting too much code inside views.

We relegated any considerations about the app's navigation structure and interaction with its data to the root layer. That simplifies our view code a lot.

Let's look again at the final result we want to achieve:



Before we go on, we need to create some data for our Xcode previews. It’s a good practice to place such data in a separate structure, so we don’t need to recreate it in every preview.

```swift
struct TestData {
	static let account =
		Account(name: "Test account", iban: String.generateIban(), kind: .checking)
	static let transaction =
		Transaction(amount: 123456, beneficiary: "Salary", date: Date())
}
```

ORGANIZING VIEW CODE INTO SMALL MODULAR TYPES

While views have few general responsibilities, their code tends to grow pretty quickly. An essential part of structuring SwiftUI views is keeping them small and reusable. Nothing prevents you from putting the code for an entire screen inside a single view. But that code would soon become unmanageable.

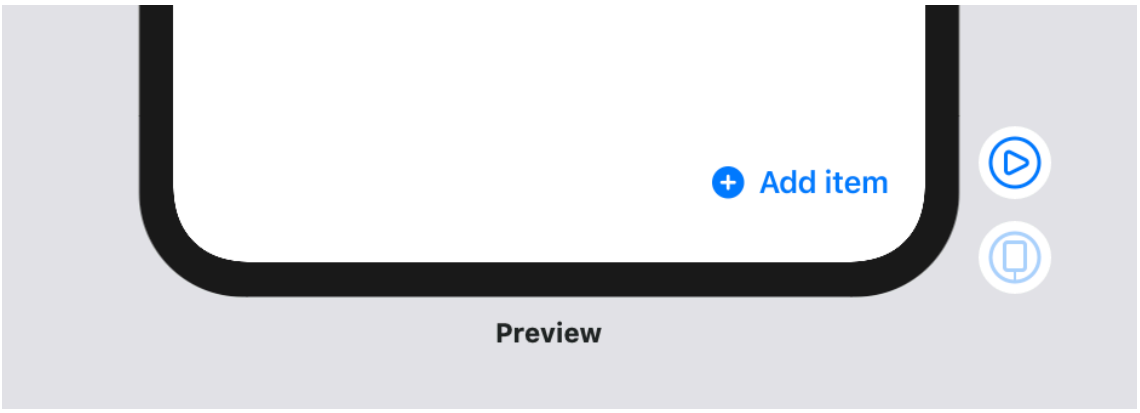
That applies to any code, not only to SwiftUI specifically. Whether you write imperative or declarative code, extended functions are hard to read and easy to get wrong.

When looking at how we should split our view code, some cases are more obvious than others. In general, any view that repeats more than once is a good candidate.

A simple example is the two buttons to add accounts and transactions at the bottom of the respective screens. Since the only difference is in their title, it's clear that their code should be in a single, reusable view.

```swift
struct AddButton: View {
	let title: String
	let action: () -> Void
	
    var body: some View {
		HStack {
			Spacer()
			Button(action: action) {
				HStack {
					Image(systemName: "plus.circle.fill")
					Text(title)
				}
				.font(.headline)
			}
			.padding(.trailing, 20.0)
		}
    }
}
```

```
struct AddButton_Previews: PreviewProvider {
    static var previews: some View {
        VStack {
            Spacer()
            AddButton(title: "Add item", action: {})
        }
    }
}
```



Here we find the most common way for views to communicate top-down with their children: stored properties. The **title** property of the **AddButton** structure has a simple **String** type, while the **action** property takes a function to pass to the **Button** view.

The rows of a table are also another glaring case of reusable views. We find one instance of that in the *Accounts* screen.

```
extension AccountsView.Content {
    struct Row: View {
        let account: Account

        var body: some View {
            VStack(alignment: .leading, spacing: 4.0) {
                HStack {
                    Text(account.name)
                        .font(.headline)
                    Spacer()
                    Text(account.balance.currencyFormat)
                        .font(.headline)
                }
                Text("\(account.kind.rawValue.capitalized) account")
                    .font(.subheadline)
                    .foregroundColor(.secondary)
                Text(account.iban.ibanFormat)
                    .font(.caption)
                    .foregroundColor(.secondary)
            }
            .padding(.vertical, 8.0)
        }
    }
}
```

I like to namespace my views inside their parent view using Swift extensions to avoid excessively long type names. The full name of this view is **AccountsView.Content.Row**. But you don't have to follow this convention. For example, you can call this view **AccountRow** and place it outside the Swift extension.

Formatting data is another of those tasks that are repeated across screens, so its code should also be isolated. Data transformation code goes inside model types, but formatting data for the user is a task the view layer. We solve this puzzle by placing formatting code inside Swift extensions for model types.

```
extension Int {
	var currencyFormat: String {
		let formatter = NumberFormatter()
		formatter.numberStyle = .currency
		return formatter.string(from: NSNumber(value: Float(self) / 100 )) ?? ""
	}
}

extension Date {
	var transactionFormat: String {
		let formatter = DateFormatter()
		formatter.dateStyle = .medium
		return formatter.string(from: self)
	}
}

extension String {
	var ibanFormat: String {
		var remaining = Substring(self)
		var chunks: [Substring] = []
		while !remaining.isEmpty {
			chunks.append(remaining.prefix(4))
			remaining = remaining.dropFirst(4)
		}
		return chunks.joined(separator: " ")
	}
}
```

We also have a table to show the list of transactions for an account, where we find another reusable row.

```swift
extension TransactionsView.Content {
	struct Row: View {
		let transaction: Transaction
		
		var body: some View {
			VStack(alignment: .leading, spacing: 4.0) {
				HStack {
					Text(transaction.beneficiary)
						.font(.headline)
					Spacer()
					Text(transaction.amount.currencyFormat)
						.font(.headline)
				}
				Text(transaction.date.transactionFormat)
					.font(.subheadline)
					.foregroundColor(.secondary)
			}
		}
	}
}
```

While reusable views are the most obvious candidate for separating code, they are not the only ones. Often, a single screen has complex sections with extended code. Even if you don't need to reuse such code anywhere else, it's still good to put it inside a separate type.

For example, in the *New Transaction* screen, we have a section to enter the amount for a transaction that requires several lines of code. We can isolate that in a modular view as well.

```swift
extension NewTransactionView.Content {
	struct Amount: View {
		@Binding var amount: String
		
		var body: some View {
			VStack(alignment: .trailing) {
				Text("Amount")
					.font(.callout)
					.foregroundColor(.secondary)
				TextField(0.currencyFormat, text: $amount)
					.multilineTextAlignment(.trailing)
					.keyboardType(.decimalPad)
					.font(Font.largeTitle.bold())
			}
			.padding()
		}
	}
}
```

KEEPING VIEWS SEPARATED FROM THE APP BUSINESS LOGIC

Thanks to the modular views we just created, the code for the screens of our app remains limited and readable. Let's start with the list of transactions for an account.

```swift
extension TransactionsView {
	struct Content: View {
		let account: Account
		let newTransaction: () -> Void

		var body: some View {
			VStack {
				List(transactions) { transaction in
					Row(transaction: transaction)
				}
				AddButton(title: "New Transaction", action: newTransaction)
			}
			.navigationBarTitle(account.name)
		}

		var transactions: [Transaction] {
			account.transactions.sorted(by: { $0.date >= $1.date })
		}
	}
}
```

What is important to notice here is how the **TransactionsView.-Content** view respects the two responsibilities of the view layer while leaving the rest outside.

First of all, it lays out the user interface for the entire screen and also sets the navigation bar title. It also sorts the transactions by date so that the most recent ones are at the top. This is part of the *user interface logic* that belongs to the view layer. This does not influence the global state of the app in our **StateController**, which can sort and store transactions in any order that makes sense there.

And finally, the **Content** view enables user interaction through the *New Transaction* button. Here, it is also essential what our view does not do. The **Content** structure does not decide what happens when the user taps the *New Transaction* button. That's part of the navigation structure of the app, which belongs to the root layer. Instead, the action for the **AddButton** is provided to the view by an ancestor through the **newTransaction** property of the **Content** type.

The same happens for all other views of our app.

```
extension NewTransactionView {
	struct Content: View {
		@Binding var amount: String
		@Binding var beneficiary: String

		let cancel: () -> Void
		let send: () -> Void

		var body: some View {
			List {
				Amount(amount: $amount)
				TextField("Beneficiary name", text: $beneficiary)
			}
```

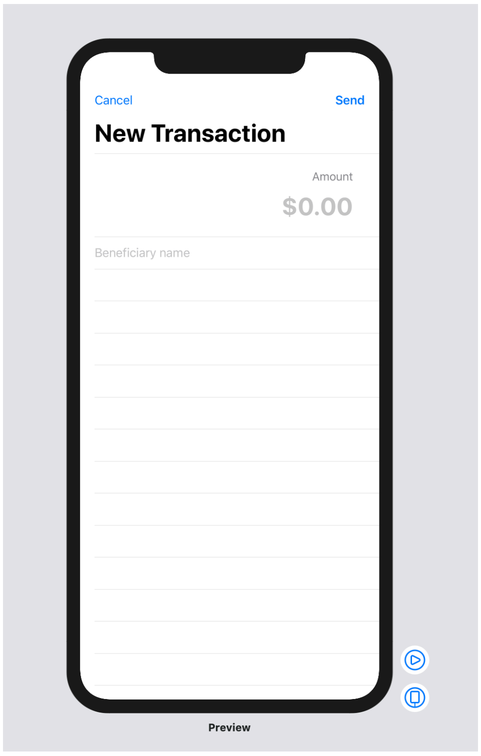
```
				.navigationBarTitle("New Transaction")
				.navigationBarItems(leading: cancelButton, trailing: sendButton)
		}

		var cancelButton: some View {
			Button(action: cancel) {
				Text("Cancel")
			}
		}

		var sendButton: some View {
			Button(action: send) {
				Text("Send")
					.bold()
			}
		}
	}
}
```

Again, the **NewTransactionView.Content** lays out the entire user interface, providing the title and the buttons for the navigation bar. But it does not implement the logic for canceling or sending a transaction. It merely passes the data entered by the user to an ancestor through two **@Binding** properties. The actions for the **cancelButton** and **createButton** come, again, through two properties that store functions.

This view is not concerned with creating new transactions and does not interact with the **StateController** in any way. It also does not dismiss the screen when the user acts.

Keeping view code separate from the controller layer makes it also more straightforward to create previews.

```
struct NewTransactionView_Previews: PreviewProvider {
      static var previews: some View {
            NavigationView {
                  NewTransactionView.Content(amount: .constant(""), beneficiary: .con-
stant(""), cancel: {}, send: {})
            }
      }
}
```

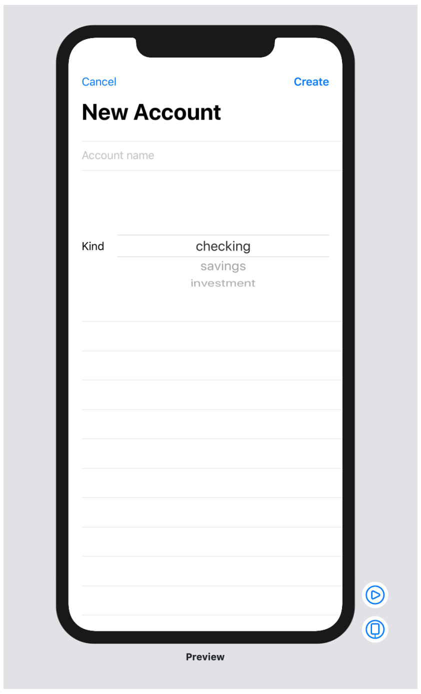


Since the **Content** view does not interact with objects, we can pass simple values as parameters in the preview. Adding a **Navigation-View** makes the navigation bar visible, so we can check that the whole screen looks as desired.

The view to create new accounts works the same way.

```
extension NewAccountView {
      struct Content: View {
            @Binding var name: String
            @Binding var kind: Account.Kind

            let create: () -> Void
            let cancel: () -> Void

            var body: some View {
                  List {
                        TextField("Account name", text: $name)
                        Picker("Kind", selection: $kind) {
                              ForEach(Account.Kind.allCases, id: \.self) { kind in
                                    Text(kind.rawValue).tag(kind)
                              }
                        }
                        Spacer()
                  }
                  .padding(.top)
                  .navigationBarTitle("New Account")
```

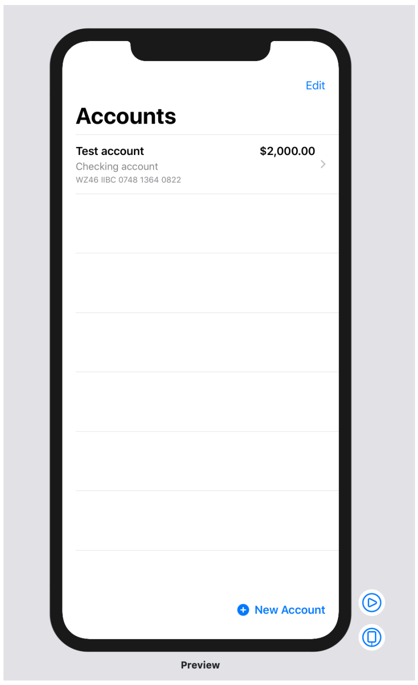
```
                .navigationBarItems(leading: cancelButton, trailing: createButton)
            }

            var cancelButton: some View {
                Button(action: cancel) {
                    Text("Cancel")
                }
            }

            var createButton: some View {
                Button(action: create) {
                    Text("Create")
                        .bold()
                }
            }
        }
    }

struct NewAccountView_Previews: PreviewProvider {
    static var previews: some View {
        NavigationView {
            NewAccountView.Content(name: .constant(""), kind: .constant(.checking),
                create: {}, cancel: {})
        }
    }
}
```



Again, this view contains interactive views like **TextField** and **Picker** and sets the navigation bar buttons, but does not handle any of the app's business logic.

And finally, we have the list of accounts.

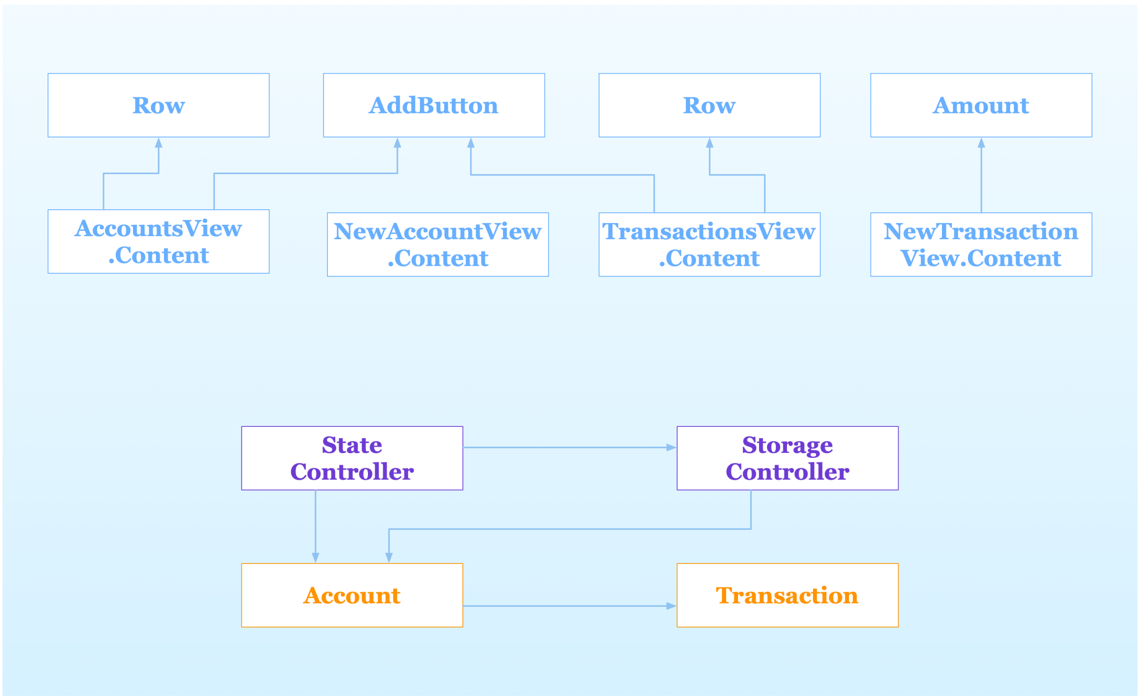
```
extension AccountsView {
    struct Content: View {
        @Binding var accounts: [Account]
        let newAccount: () -> Void

        var body: some View {
                VStack {
                    List {
                        ForEach(accounts) { account in
                            NavigationLink(destination:
                                TransactionsView(account: account)) {
                                    Row(account: account)
                            }
                        }
                        .onMove(perform: move(fromOffsets:toOffset:))
                    }
                    AddButton(title: "New Account", action: newAccount)
                }
                .navigationBarTitle("Accounts")
                .navigationBarItems(trailing: EditButton())
            }

            func move(fromOffsets source: IndexSet, toOffset destination: Int) {
                accounts.move(fromOffsets: source, toOffset: destination)
            }
        }
    }

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        NavigationView {
            AccountsView.Content(accounts: .constant([TestData.account]),
                newAccount: {})
        }
    }
}
```



This view contains some more user interface logic and allows the user to reorder the accounts on the screen. But again, it does not interact with the **StateController** where those accounts are stored. Instead, it takes an array of accounts through a **@Binding** property, which it uses to send data changes to an ancestor view.

You could even make the view receive the **move(fromOffsets:toOffset:)** through a function property if you wanted a more sophisticated approach, but this will suffice for our app.

This is what we have created until now.



The lower and upper layers of our app are still disconnected. What brings them all together is the root layer, which we will cover in the next, last chapter of this guide.

BRINGING THE WHOLE APP TOGETHER IN THE ROOT LAYER

We have finally come to the last layer of the MVC pattern: the root layer. Beware that the naming is mine, so you probably won't find any other developer refer to it with this name. That is unless this idea catches on.

Here we find all the remaining responsibilities for our app:

- structuring the navigation flow;
- connecting views to controllers;
- interpreting user action.

MANAGING THE APP'S NAVIGATION STRUCTURE WITH ARCHITECTURAL VIEWS AND MODAL PRESENTATION

First of all, we need a single instance of the **StateController** that can be accessed by all the screens in our app. We create that in the **SceneDelegate** of our app and then add it to the environment for our user interface.

```
class SceneDelegate: UIResponder, UIWindowSceneDelegate {
	var window: UIWindow?
	let stateController = StateController()

	func scene(_ scene: UIScene, willConnectTo session: UISceneSession,
		options connectionOptions: UIScene.ConnectionOptions) {
		let contentView = AccountsView()
			.environmentObject(stateController)
		if let windowScene = scene as? UIWindowScene {
			let window = UIWindow(windowScene: windowScene)
			window.rootViewController = UIHostingController(rootView: contentView)
			self.window = window
			window.makeKeyAndVisible()
		}
	}
}
```

Once the **StateController** instance is in the environment, it can be accessed by any root view using the **@EnvironmentObject** property wrapper.

Let's start from the first screen in our app.

```
struct AccountsView: View {
      @EnvironmentObject private var stateController: StateController
      @State private var addingAccount = false

      var body: some View {
            NavigationView {
                  Content(accounts: $stateController.accounts,
                        newAccount: { self.addingAccount = true })
            }
            .sheet(isPresented: $addingAccount) {
                  NavigationView {
                        NewAccountView()
                  }
                  .environmentObject(self.stateController)
            }
     }
}
```

This is our first root view. Its code is entirely structural. It binds the **accounts** property of the **Content** view to the **accounts** property of **StateController**.

It also presents the *New Account* screen modally and provides two navigation views. The first one enables the main drill-down navigation for the app, which leads to the list of transactions for the selected account. The second navigation view is only needed to show a navigation bar in the *New Account* screen, even though navigation does not proceed further.

```
struct TransactionsView: View {
      let account: Account

      @EnvironmentObject private var stateController: StateController
      @State private var addingTransaction: Bool = false

      var body: some View {
            Content(account: account, newTransaction: { self.addingTransaction = true })
                  .sheet(isPresented: $addingTransaction) {
                        NavigationView {
                              NewTransactionView(account: self.account)
                                    .environmentObject(self.stateController)
                        }
            }
      }
}
```

The **TransactionView** structure works the same. It connects its **Content** view to the shared instance of the **StateController** and manages the modal presentation of the **NewTransactionView**.

MAPPING USER ACTION INTO THE APP'S BUSINESS LOGIC

All we have left are the views to create new accounts and transactions.

```
struct NewAccountView: View {
	@EnvironmentObject private var stateController: StateController
	@Environment(\.presentationMode) private var presentationMode
	@State private var name: String = ""
	@State private var kind: Account.Kind = .checking

	var body: some View {
		Content(name: $name, kind: $kind, create: create, cancel: dismiss)
	}

	func create() {
		stateController.addAccount(named: name, withKind: kind)
		dismiss()
	}

	func dismiss() {
		presentationMode.wrappedValue.dismiss()
	}
}
```

Here we find a first example of mapping the user action into the app's business logic. The **NewAccountView** structure collects the data it receives from its **Content** view into two **@State** properties. Then, when the user taps on the *Create* button, it creates a new account by calling the **addAccount(named:withKind:)** method of the **StateController**.

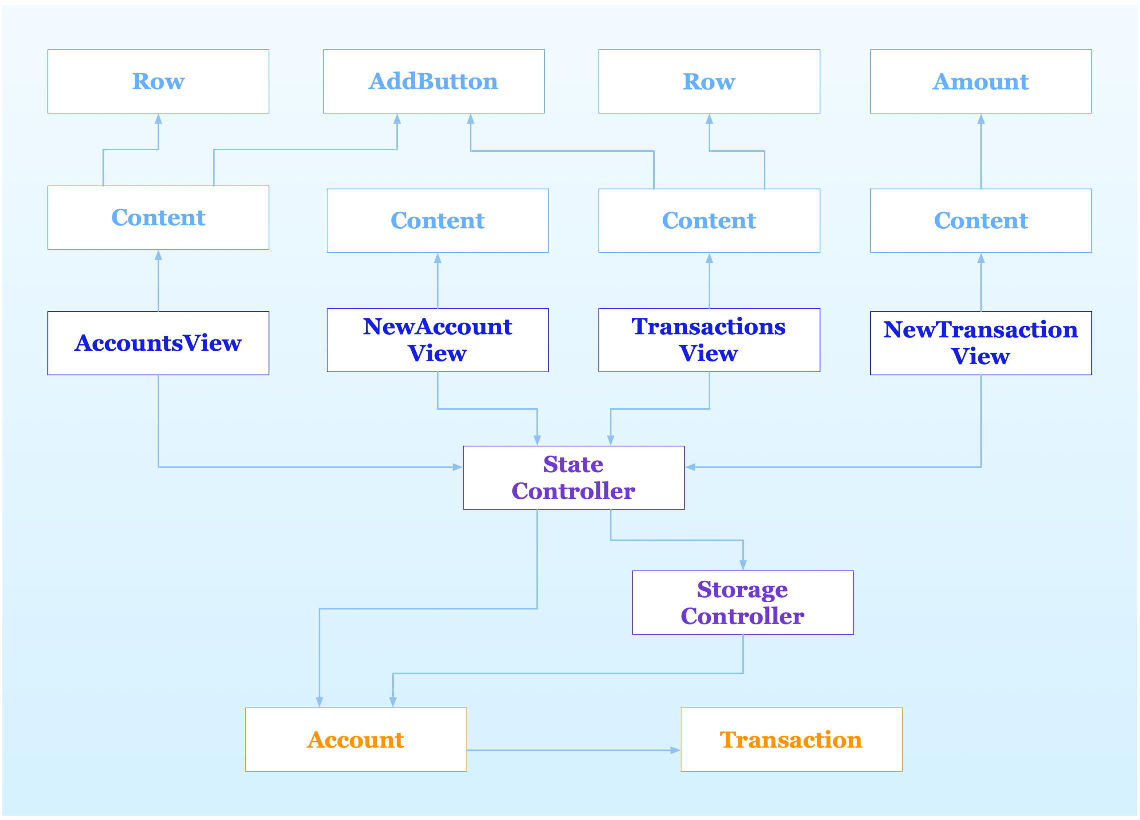
Notice that we are accessing the **StateController** directly instead of passing data up the view hierarchy using bindings. Creating a new account is a responsibility of the *New Account* screen. If we passed data up the view hierarchy, we would move the applicable code to the **AccountView,** which should have nothing to do with creating accounts.

And finally, our **NewAccountView** also manages navigation, dismissing the presented screen when the user creates an account or cancels.

The same happens in the **NewTransactionView**, which also adds new transactions to an account through the **StateController**, which is the final repository of all the app's business logic.

```
struct NewTransactionView: View {
	let account: Account

	@EnvironmentObject private var stateController: StateController
	@Environment(\.presentationMode) private var presentationMode

	@State private var amount: String = ""
	@State private var beneficiary: String = ""

	var body: some View {
		Content(amount: $amount, beneficiary: $beneficiary, cancel: dismiss,
			send: send)
	}

	func send() {
		let amount = (Int(self.amount) ?? 0) * 100
		stateController.addTransaction(withAmount: amount, beneficiary: beneficiary,
			to: account)
		dismiss()
	}

	func dismiss() {
		presentationMode.wrappedValue.dismiss()
	}
}
```

We finally covered the whole app. With the root layer, we provided all its navigation flow and connected all screens to the central state.



Thanks to the MVC pattern, we have a well-structured and understandable app. Each type, be it a model type, a controller, or a view, is small and manageable and contains only a limited amount of responsibilities.

Moreover, we have a map that allows us to find any specific code we are looking for quickly. And, finally, we can add new features and screens to our app, knowing exactly where to place any new code, keeping the project manageable as it grows.

PLEASE SHARE THIS GUIDE

I hope you enjoyed this guide and it helped you improve your understanding of iOS architecture in SwiftUI apps. It took me many hours of work to put it together, but I am happy to offer it to you free of charge.

If you found it useful, please take a moment to share it with someone that could also find it useful. In this way, I can dedicate more time into creating more free articles and guides to help you in your iOS development journey.

Think about colleagues and friends that could find this guide useful and send them this link through email, on forums, or through a private message, so that they can get it together with all the other material I only share with my email list:

https://matteomanferdini.com/architecting-swiftui-apps-with-mvc-and-mvvm/

You can also use one of these links to share the guide on your favorite social network.

Click here to share it on Facebook

Click here to share it on Twitter

Click here to share it on LinkedIn