SwiftUI Architecture Blueprint - A Practical Guide to Scalable iOS Applications

If you've ever asked:
- "Is this a ViewModel or a Service?"
- "Should I use EnvironmentObject or pass data down?"
- "What's the difference between a Repository and a Service?"

You're not alone. Architecture debates in SwiftUI are everywhere—and often full of confusing or conflicting terminology. This glossary doesn't aim to settle every argument or define absolute truths. It's simply a guide to help us communicate clearly, so we can focus more on building great apps and less on fighting about names. This is not about rules. It's about clarity.

Let's Start With The Basics

Model

A Model is a simple type (usually a struct) that represents your raw data. It's the single source of truth for a particular piece of information in your app. Models should be independent of your UI - they don't know anything about how they'll be displayed.

```swift
struct User: Identifiable, Codable {
    let id: UUID
    var name: String
    var email: String
    var birthDate: Date
}
```

Think of a Model as a digital representation of a real-world concept. Just like a blueprint for a house contains measurements and specifications, a User model contains all the essential information that defines a user.

They often conform to:

- Codable for data persistence or networking
- Identifiable, Hashable for SwiftUI view spedifics like NavigationStack, List

> Note: if you use the Observation feature (iOS 17+) you models should be class annoted with the @Observable macro

View

A View in SwiftUI is a blueprint for a piece of your user interface. It's not the actual UI elements you see on screen - it's a description of how things should look based on your current state.

```swift
struct UserProfileView: View {
    let user: User

    var body: some View {
        VStack(alignment: .leading, spacing: 12) {
            Text(user.name)
                .font(.title)
            Text(user.email)
                .font(.subheadline)
        }
    }
}
```

Important points about Views:
- They are value types (structs)
- They are lightweight and composable
- Automatically update when their dependencies change
- They should be simple and focused on presentation
- They rebuild themselves when their data changes

Think of a View like a restaurant order ticket: it describes what should be made, but it's not the actual meal.

View Dependencies / UI State

SwiftUI has a simple rule: **When data changes, UI updates automatically**. But this only works when SwiftUI knows about your data dependencies.

What is State?

State is data that can change and should trigger UI updates when it does.

```swift
struct ContentView: View {

   @State private var isOn = false

   var body: some View {
    VStack {
       Button("Setting") {
          isOn.toggle() // Change the state
       }

      Rectangle().fill(isOn ? Color.green : Color.clear)// UI responds to state
    }
  }
}
```

When you tap the button, `isOn` changes, and SwiftUI automatically redraws the rectangle with the new color. This is SwiftUI's **reactive data flow** in action.

Dependencies: The Foundation of UI Updates

Every view has dependencies - data it needs to render correctly. When these dependencies change, SwiftUI re-renders the view.

```
struct ContentView: View {
   @State private var isOn = false

   var body: some View {
    VStack {
       Button("Setting") {
          isOn.toggle()
       }

      StatusIndicator(isActive: isOn)  // Passing dependency down
    }
  }

struct StatusIndicator: View {
   let isActive: Boolean // This view depends on this data
   var body: some View {
      Rectangle()
        .fill(isActive ? Color.green : Color.clear)
        .animation(.easeInOut, value: isActive) // Animate when dependency changes
  }
}
```

What's happening here:

1. `ContentView` owns the state (`isOn`)
2. `StatusIndicator` **depends** on that state (through `isActive`)
3. When `isOn` changes in `ContentView`, SwiftUI automatically updates `StatusIndicator`
4. The data flows **down** from parent to child

Why Dependencies Matter

Dependencies define your **data flow**. Poor dependency management leads to:

- Views that don't update when they should
- Performance issues from unnecessary re-renders
- Hard-to-debug UI glitches

The Dependency Chain

In complex apps, dependencies flow through multiple view layers:

```swift
struct AppView: View {
    @State private var userIsLoggedIn = false

    var body: some View {
        MainContentView(isLoggedIn: userIsLoggedIn)  // Dependency flows down
    }
}

struct MainContentView: View {
    let isLoggedIn: Bool  // Depends on parent's state

    var body: some View {
        if isLoggedIn {
            DashboardView()
        } else {
            LoginView(onLogin: { /* update parent state */ })
        }
    }
}

struct LoginView: View {
    let onLogin: () -> Void
    @State private var username = ""  // Local state
    @State private var password = ""  // Local state

    var body: some View {
        VStack {
            TextField("Username", text: $username)  // Bound to local state
            SecureField("Password", text: $password)  // Bound to local state
            Button("Login") {
                onLogin()  // Calls back to parent to change its state
            }
        }
    }
}
```

Key insight: Each view clearly declares what it depends on. When any dependency changes, only the views that actually depend on that data get re-rendered.

Dependency Best Practices

1. **Be explicit about dependencies** - Make it clear what data each view needs
2. **Keep dependencies minimal** - Only pass down what a view actually uses
3. **Consider the data flow** - Data flows down, events flow up

```swift
// GOOD: Clear, minimal dependencies
struct UserProfileView: View {
    let user: User  // Only depends on user data
    let onEdit: () → Void  // Only needs this callback

    var body: some View {
        VStack {
            Text(user.name)
            Text(user.email)
            Button("Edit", action: onEdit)
        }
    }
}

// AVOID: Too many dependencies
struct OverComplexView: View {
    let user: User
    let settings: Settings
    let networkManager: NetworkManager
    let analyticsTracker: AnalyticsTracker
    // ... this view depends on too much!
}
```

Remember: **Dependencies are SwiftUI's way of knowing when to update your UI**. Make them explicit, keep them minimal, and your app's data flow will be predictable and performant.

> ⚠️ **Warning: Hidden Dependencies**
> A convenient way of passing dependencies is the SwiftUI environment. All connected views will get an implicit/hidden dependency. It makes code harder to reason about in large view compositions.
> Use this with care and only for global state (user authentication, app theme, locale settings) and not as a default go to solution.

```swift
// BETTER for feature-specific data
struct ProductDetailView: View {
    let product: Product  // Clear what this view needs
    let onPurchase: (Product) → Void  // Clear what it can do

    @EnvironmentObject var appTheme: ThemeSettings // hidden dependency
}
```

> 💡 **Best Practice for Large Apps:**
> Use Swift Packages to create feature modules. Each module should manage its own environment objects and only expose what other modules actually need. This prevents the "everything is global" problem and keeps dependencies clear and manageable.

Local State - When to Use It

For simple views with minimal logic, SwiftUI's built-in state management is often sufficient.

```
struct LoginForm: View {

    @State private var username = ""
    @State private var password = ""
    @FocusState private var focusedField: Field?

    enum Field {
        case username, password
    }

    var body: some View {
        VStack {
            TextField("Username", text: $username)
                .focused($focusedField, equals: .username)

            SecureField("Password", text: $password)
                .focused($focusedField, equals: .password)

            Button("Login") {
                // Login logic here
            }
            .disabled(username.isEmpty || password.isEmpty)

            Button("Focus on Username") {
                focusedField = .username
            }
        }
        .padding()
    }
}
```

Local state works well when:

- The state is only used by a single view
- The logic is simple and self-contained
- There's no need to share this state with other views

In the above example the focusField state is only used in the current form view. When the user press enter it is used to programmatically change the focus of the textfield and move to the next entry. Other examples for local state include

- gesture states
- state that is used only for animations

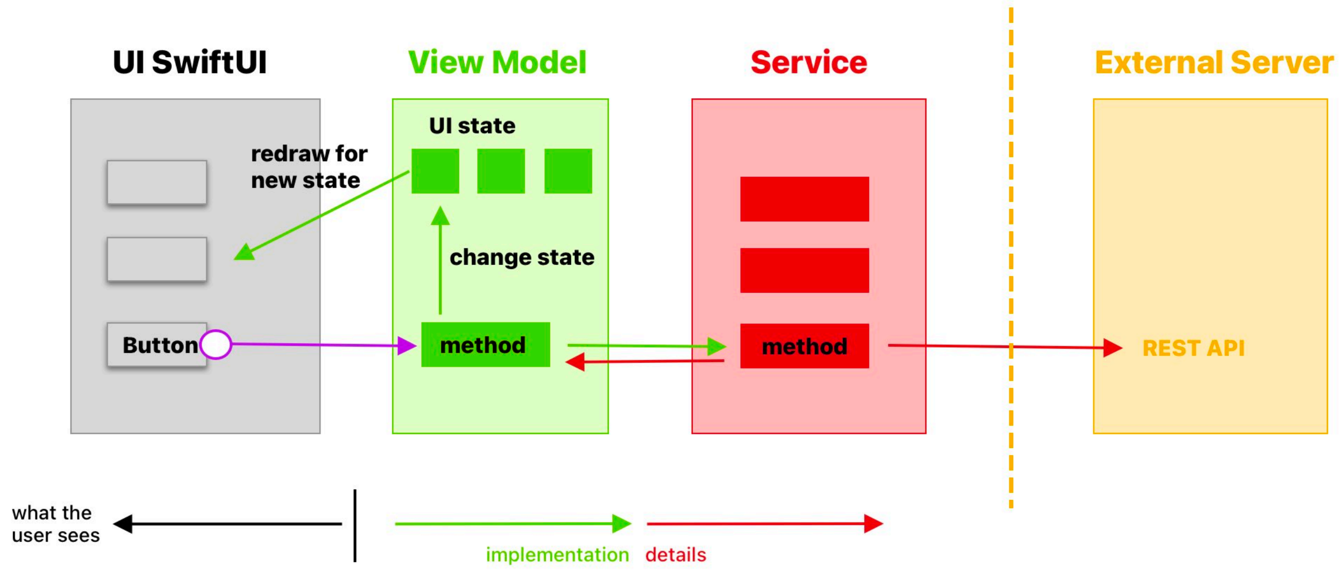
But as you might want to extend the view. For example, in the above form view, you might want to evalueate the entries to check e.g. the email is in a vaild formate. Instead of keeping this logic in the view you might consider upgrading and placing it in a separate layer.

Things Get More Complicated

As your app grows, you'll encounter challenges with local state:

1. **Larger Views**: Views become bloated with both UI and logic
2. **Complex Logic**: Business rules, validations, and transformations make views hard to understand
3. **State Sharing**: Multiple views need access to the same state
4. **Testability**: UI logic mixed with business logic is difficult to test

The solution? Separate your concerns by introducing another layer: *the view model*.



MVVM: Separating Logic from UI

MVVM (Model-View-ViewModel) separates your UI (View) from your business logic and state (ViewModel).

ViewModel

A ViewModel acts as a bridge between your Model and View. It prepares data for display and handles user interactions. ViewModels are where you put your presentation logic.

```
import Combine

class UserProfileViewModel: ObservableObject {
    // Published properties automatically notify the View when they change
    @Published private(set) var user: User
    @Published var isEditing = false
    @Published var editingName: String = ""

    // Computed properties format data for the View
    var formattedBirthDate: String {
        let formatter = DateFormatter()
        formatter.dateStyle = .medium
        return formatter.string(from: user.birthDate)
    }

    init(user: User) {
        self.user = user
    }

    // Functions that handle user actions
    func toggleEditMode() {
        isEditing.toggle()
        editName = user.name
    }

    func updateName() {
        // You might add validation here
        user.name = newName
        // You migh save to disk
    }
}
```

or with the new Observation feature (iOS 17+):

```
import Observation

@Observable
class UserProfileViewModel {
    private(set) var user: User
    var isEditing = false

   ...
}
```

Think of a ViewModel as a personal assistant for your View:

- It manages the state the View needs
- It handles user interactions
- It formats data for display
- It communicates with other parts of your app (like services)

Here's how they all work together:

```swift
struct UserProfileScreen: View {

    @StateObject private var viewModel = UserProfileViewModel()

    var body: some View {
        VStack {
            // Display formatted data from ViewModel
            Text(viewModel.formattedBirthDate)

            // Handle user interaction through ViewModel
            Button(viewModel.isEditing ? "Done" : "Edit") {
                viewModel.toggleEditMode()
            }

            if viewModel.isEditing {
                TextField("Name", text: $viewModel.editingName)
                  .onSubmit { newName in
                      viewModel.updateName()
                  }
            }
        }
    }
}
```

What is a ViewModel

In recent years, there has been a discussion to stop using ViewModels and MVVM. People developed new patterns. You might see layers called "Store", "Service" or "Manager". When you look at these layers, its a good idea to go through the following questions:

- Does this layer contain state which is used for the UI?
- Does this layer contain methods that change state of the layer?

If the answer to both of these questions is yes, that the layer is in its core a ViewModel. This might be all a question of definition. But I would like to emphasis the importance of clearly communicating what layers and concepts you introduce to your codebase so your collegues understand you.

Here is the definition of a ViewModel that I like to use:

> **A ViewModel contains UI state and methods that change UI state**. It is thus the layer directly behind the UI layer. Its main purpose is to back up the business functionality of the UI.

Advantages of MVVM

1. **Separation of Concerns**: UI code stays in the View, business logic in the ViewModel
2. **Reusability**: ViewModels can be reused across different views
3. **Testability**: You can unit test your ViewModel without testing the UI
4. **Maintainability**: Changes to UI don't affect business logic and vice versa
5. **Collaboration**: One developer can work on Views while another works on ViewModels

When to Use MVVM (and When Not To)

Use MVVM When:

- Your view contains complex logic
- You need to share state between multiple views
- You want to make your code more testable
- Your app is likely to grow in complexity
- You're working in a team where responsibilities are divided

Consider Simpler Approaches When:

- Creating a very simple view with minimal logic
- Working on a small app that won't grow significantly
- The overhead of creating ViewModels doesn't provide enough benefit

Remember, architecture is a tool to manage complexity. Start simple, and add layers of abstraction only when they solve real problems you're facing.

Beyond MVVM: Understanding Services and Repositories

When Do You Need These?

As your app grows, you'll notice ViewModels becoming too large and handling too many responsibilities. You might also find yourself duplicating networking code or wanting to add caching. This is when these patterns become valuable.

Services: Your App's Workers

A Service is like a specialized worker in your app. It's responsible for one specific type of operation, such as making network calls or handling file storage. Services are intentionally simple and stateless - they don't remember anything between operations.

- A **Service is a stateless component (typically a struct) that performs operations**
- Think of it as a "worker" that does one specific type of task
- Contains no state - just methods that perform operations
- Examples: API calls, file operations, device features

Here's what a typical Service looks like:

```swift
// Services are typically protocol-based
protocol UserService {
    func fetchUser(id: UUID) async throws -> User
    func updateUser(_ user: User) async throws
}
```

```swift
// Implementation is stateless
struct APIUserService: UserService {
    // No stored properties (stateless)

    func fetchUser(id: UUID) async throws → User {
        let url = URL(string: "api/users/\(id)")!
        let (data, _) = try await URLSession.shared.data(from: url)
        return try JSONDecoder().decode(User.self, from: data)
    }
}
```

Think of a Service like a delivery person - they just perform tasks (delivering packages) but don't keep track of what's been delivered. They get a request, do the work, and return the result.

Benefits of Services:

- Separates concerns (networking logic isn't in ViewModels)
- Easier to test (can mock network calls)
- Reusable across different ViewModels
- Can switch implementations (e.g., mock vs real API) easily

Repository Pattern

A Repository is like a smart manager that coordinates data access in your app. Unlike Services, Repositories can remember things (contain stored properties) and make smart decisions about where to get data from.

- A **Repository** maintains state and encapsulates data access logic
- Acts as a single source of truth for a type of data
- Coordinates between different services and local storage
- Can maintain caches or handle offline functionality

Here's what a Repository typically looks that performs and caches networrking request:

```swift
// Repositories are typically protocol-based
protocol UserRepository {
    func getUser(id: UUID) async throws → User
    func saveUser(_ user: User) async throws
    func getCachedUsers() → [User]
}
```

```swift
class DefaultUserRepository: UserRepository {
    // Has state
    private var cache: [UUID: User] = [:]

    // Owns services
    private let apiService: UserService
    private let databaseService: DatabaseService

    init(apiService: UserService, databaseService: DatabaseService) {
        self.apiService = apiService
        self.databaseService = databaseService
    }

    func getUser(id: UUID) async throws → User {
        // Check cache
        if let cachedUser = cache[id] {
            return cachedUser
        }

        // Try database
        if let localUser = try? await databaseService.fetchUser(id: id) {
            cache[id] = localUser
            return localUser
        }

        // Fallback to API
        let user = try await apiService.fetchUser(id: id)
        cache[id] = user
        try? await databaseService.saveUser(user)
        return user
    }
}
```

Think of a Repository like a library manager. When you request a book, they might:

1. Check if it's on the front desk (cache)
2. Look in the local shelves (database)
3. Order it from another library (API)

The Repository makes these decisions for you, and you don't need to know the details.

When to use Repositories:

- When you need caching
- When data comes from multiple sources
- When you want to abstract data source implementation details
- When you need offline support

How They Connect Service/Repository to MVVM

In a traditional MVVM implementation for a simple app, you can have the following view model:

```swift
class UserViewModel: ObservableObject {
    @Published var user: User?

    // Bad: Direct API calls in ViewModel
    func fetchUser() async {
        let url = URL(string: "api/user")!
        let (data, _) = try? await URLSession.shared.data(from: url)
        self.user = try? JSONDecoder().decode(User.self, from: data)
    }
}
```

You can add a Service layer to MVVM like so:

```swift
class UserViewModel: ObservableObject {
    @Published var user: User?

    private let userService: UserService

    init(userService: UserService) {
        self.userService = userService
    }

    func fetchUser(id: UUID) async {
        user = try? await userService.fetchUser(id: id)
    }
}
```

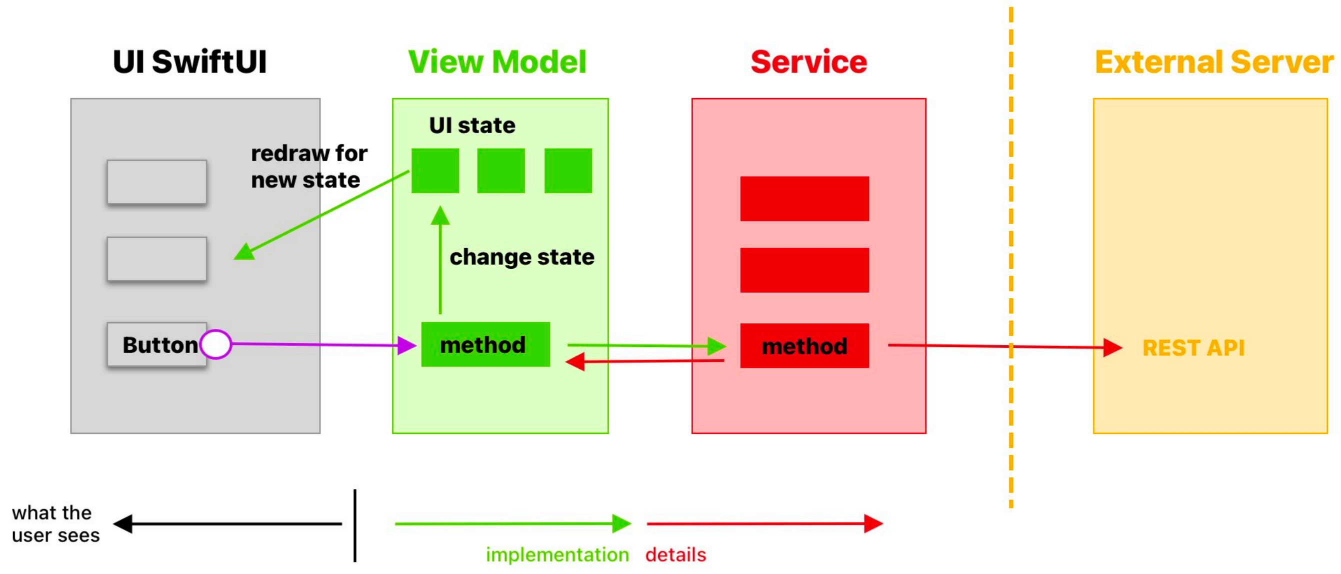
When things get more complex, you can extend MVVM with Repository (Large App):

```swift
class UserViewModel: ObservableObject {
    @Published var user: User?
    @Published var cachedUsers: [User] = []

    private let userRepository: UserRepository

    init(userRepository: UserRepository) {
        self.userRepository = userRepository
        self.cachedUsers = userRepository.getCachedUsers()
    }

    func fetchUser(id: UUID) async {
        user = try? await userRepository.getUser(id: id)
    }
```

In this setup:

1. The ViewModel talks to the Repository
2. The Repository manages the data access strategy
3. The Repository uses Services to perform specific operations
4. The View remains unchanged, only talking to its ViewModel

Think of it this way:
- Services are like individual tools (hammer, screwdriver)
- Repositories are like a toolbox that organizes and manages those tools
- ViewModels use the toolbox (Repository) rather than reaching for individual tools (Services)



When to Add These Layers

Start with MVVM, and add these layers when:

1. Add Services when:
- Your ViewModels contain lots of network or file operation code
- You're repeating the same operation logic in multiple ViewModels
- You need to make testing easier
2. Add Repositories when:
- You need to cache data
- You're dealing with multiple data sources (API, database, cache)
- You need offline support
- Your data access logic is getting complex

Remember, these patterns are tools to help manage complexity. Don't add them until you need them - a simple app might work perfectly fine with just MVVM. The goal is to make your code more maintainable and easier to understand, not to make it more complicated.

How not to write spaghetti code with many layers of abstraction

The more layers/components you add, the more work you need to put into making them work together. One strategy that can help is to have clear rules who talks to whom. You should make sure that these layers are separated by how close/far away they are from the UI.

For a service with ViewModel implementation:

> View ⟷ ViewModel ⟷ Service

or with a repository + service composition:

> View ⟷ ViewModel ⟷ Repository ⟷ Service(s)

That means the view can only talk to the ViewModel, not layers like the repository or service that are 'behind' the ViewModel. You should have clear 'paths' of logical flow

For example, a view is shown that depends on a boolean property which is owned by a view model:

> View ⟵ ViewModel **state**

pressing a button in the view will pass the action to the ViewModel:

> View button ⟶ ViewModel method ⟶ Service method

The ViewModel method internally calls a method from the service layer. It uses the return value to change its state:

> View update ⟵ ViewModel update state ⟵ Service method return value

As you see there is one clear path that defines what logic is called and what state is modified.

Why it matters: When data can be changed from multiple places, bugs happen. Having one clear owner prevents sync issues. This represents the *single source of change* for the UI state.

Testing Your Business Logic

Once your app grows, you'll want to make your code testable. This is where protocols come in:

```swift
// First, define the contract
protocol UserServiceType {
    func fetchUser(id: UUID) async throws → User
    func updateUser(_ user: User) async throws
}

// Real implementation
struct UserService: UserServiceType {
    func fetchUser(id: UUID) async throws → User {
        // Real API implementation
    }
```

In the above sevice and repository examples, I already defined the protocols.

Testing Services

Services are the easiest to test because they're stateless and have clear inputs and outputs (pure functions):

```swift
class MockUserService: UserService {
    var mockUser: User?
    var error: Error?

    func fetchUser(id: UUID) async throws → User {
        if let error = error {
            throw error
        }
        return mockUser ?? User(id: id, name: "Test User")
    }
}

func testUserService() async throws {
    // Arrange
    let mockService = MockUserService()
    mockService.mockUser = User(id: UUID(), name: "John")

    // Act
    let user = try await mockService.fetchUser(id: UUID())

    // Assert
    XCTAssertEqual(user.name, "John")
}
```

Testing Repositories

Repositories require more setup because they maintain state and coordinate multiple services. You would typically use a MockService (instead of actually making API calls which can be brittle. The following tests the repository function getUser:

```
class TestUserRepository {
    func testCaching() async throws {
        // Arrange
        let mockService = MockUserService()
        let repository = DefaultUserRepository(service: mockService)

        // Act
        let user1 = try await repository.getUser(id: UUID())
        let user2 = try await repository.getUser(id: user1.id)

        // Assert
        XCTAssertEqual(user1.id, user2.id) // Should hit cache
    }
}
```

Testing ViewModels

With proper Services and Repositories, testing ViewModels becomes straightforward. You can inject the mock service/repository to the view model and write clean test like so:

```
class UserProfileViewModelTests: XCTestCase {
    func testLoadUser() async {
        // Arrange
        let mockRepository = MockUserRepository()
        let viewModel = UserProfileViewModel(repository: mockRepository)

        // Act
        await viewModel.loadUser(id: UUID())

        // Assert
        XCTAssertNotNil(viewModel.user)
    }
}
```

Best Practices for Testable Architecture

1. Always code to protocols (interfaces):

```
protocol UserService { ... }
protocol UserRepository { ... }
```

2. Use dependency injection:

```
class UserProfileViewModel {
    init(repository: UserRepository) { ... }
}
```

3. Keep Services stateless:

```swift
struct APIService { ... }  // Use struct to enforce statelessness
```

4. Make Repositories responsible for state:

```swift
class Repository {
    private var cache: [UUID: User] = [:]  // State belongs here
}
```

This separation makes it clear what to test and how:

- Services: Test the operation logic
- Repositories: Test the caching and coordination logic
- ViewModels: Test the presentation logic

Remember: Good architecture naturally leads to testable code. If something is hard to test, it might indicate a design that needs improvement.

Coordinators: Separating Navigation Logic

After organizing your business logic with MVVM and data management with Services and Repositories, you might notice that your Views still handle a lot of navigation decisions. This mixing of concerns can make your app harder to maintain, especially when navigation depends on business rules or API responses.

What is a Coordinator?

Think of a Coordinator as a traffic controller for your app's screens. While ViewModels handle "what to show" on a screen, Coordinators handle "which screen to show next" and "how to get there."

Key differences from other patterns:
- ViewModels manage screen state and business logic
- Services handle external operations
- Repositories manage data access
- Coordinators manage navigation flow and screen creation

Here is at traditional approach (navigation mixed in View):

```
struct ProductListView: View {
    @StateObject private var viewModel: ProductViewModel
    @State private var navigationPath = NavigationPath()

    var body: some View {
     NavigationStack(path: $path) {
        List(viewModel.products) { product in
            Button {
                // Navigation logic mixed with UI
                if product.isAvailable {
                    navigationPath.append(product)
                } else {
                    // Show alert
                }
            } label: {
                ProductRow(product: product)
            }
        }
      }
    }
}
```

Which you can change to use the coordinator approach (separated navigation):

```
class ProductCoordinator: ObservableObject {
    @Published var navigationPath = NavigationPath()
    private let productService: ProductService

    func showProductDetail(_ product: Product) async {
        // Can perform checks before navigation
        let isAvailable = await productService.checkAvailability(product.id)

        if isAvailable {
            navigationPath.append(Destination.productDetail(product))
        } else {
            // Handle unavailable product
            presentAlert(.productUnavailable)
        }
    }
}
```

I added more complex logic in the showProductDetailView where i first make a network request and depending on the result change what is shown in the navigation stack. This is a more complex task because I am mixing networking (typically in view models) and navigation logic. The coordinator is a separate layer that isolates and handles these navigation related logic. In the view you can then use it like:

```
struct ProductListView: View {
    @StateObject private var viewModel: ProductViewModel
    @StateObject private var coordinator = ProductCoordinator()

    var body: some View {
     NavigationStack(path: $coordinator.path) {
        List(viewModel.products) { product in
            Button {
                coordinator.showProductDetail(product)
            } label: {
                ProductRow(product: product)
            }
        }
    }
}
```

The key insight here is that Coordinators are fundamentally different from ViewModels because they:

1. Focus on screen flow, not screen content
2. Often span multiple screens
3. Can contain complex navigation logic
4. Manage dependencies for new screens
5. Can handle deep linking and complex navigation states

This separation makes your app's navigation flow more maintainable and testable, especially in larger applications.

Decision Framework

"Where does this code belong?"

- **Is it formatting data for display?** → ViewModel
- **Is it a network call or external API interaction?** → Service
- **Is it saving/loading data locally?** → Repository
- **Is it complex business logic used by multiple ViewModels?** → Service
- **Is it simple view state (show/hide, selections)?** → View with @State
- **Is it data that multiple views need?** → Consider EnvironmentObject or shared Service

---

"Should I create a new Service or extend existing?"

Ask yourself:

1. **Single Responsibility**: Does the existing service have a clear, single purpose?
2. **Dependencies**: Would adding this require new dependencies that don't fit?
3. **Testing**: Would this make the service harder to test?
4. **Size**: Is the service getting unwieldy?

If you answered "yes" to any of these, create a new service.

"EnvironmentObject or property passing?"

Use EnvironmentObject when:

- Data is needed by views at different hierarchy levels
- It's app-wide state (user session, theme, settings)
- You want to avoid prop drilling

Use property passing when:

- Dependencies should be explicit for testing
- Data is specific to a view hierarchy
- You want to be clear about what each view needs

Glossary

Source of Truth: The single place where a piece of data is owned.

Source of Change: The single place where a piece of data is modified.

ViewModel: Object that prepares data for views and handles view-related business logic. It is holds UI state and has methods to change the UI state.

Service: Layer that handles business logic or external communication. Does not own properties. Only contains pure functions. Typically a protocol with value type/struct implementations for default and mock behavior.

Repository: Abstraction over data access, whether from network, database, or files. Holds stored properties. Typically a reference type/class.

Coordinator: Pattern for managing navigation and flow between screens.

Dependency Injection: Providing dependencies from outside rather than creating them internally.

@ObservedObject: The iOS 13+ way to observe changes to reference types.

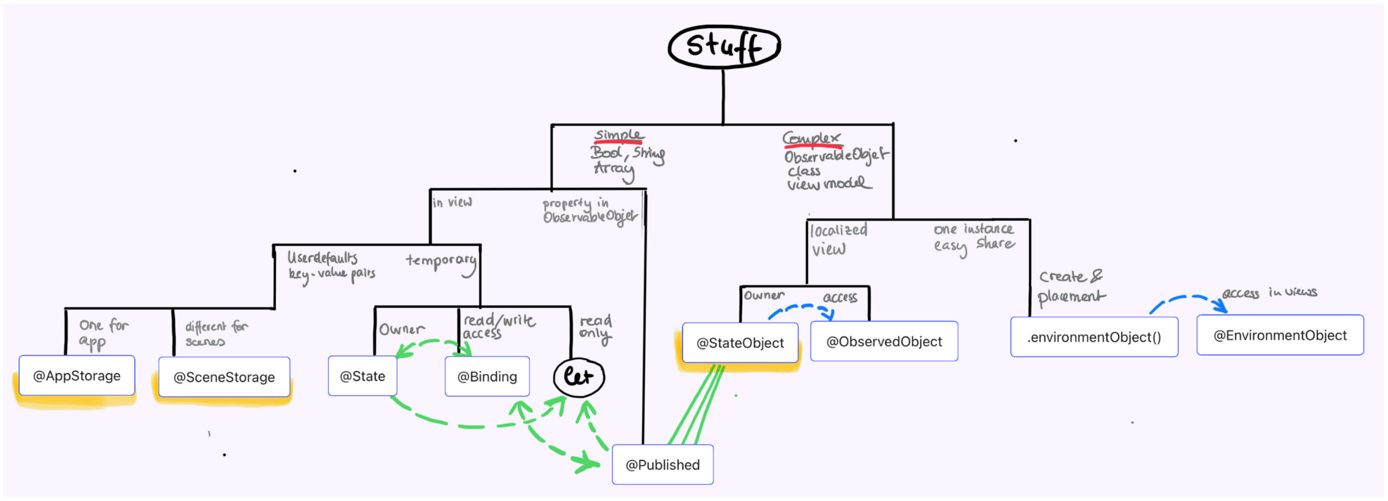
@Published: Property wrapper that automatically publishes changes to ObservableObject subscribers.

@State: SwiftUI's way to manage simple, local view state.

@Binding: Two-way connection between a property and a view that displays and changes it.

@StateObject: Use when your view creates and owns an ObservableObject.

@EnvironmentObject: Way to provide data to multiple views without explicit passing.



@Observable: New iOS 17+ way to make objects observable. Simpler than ObservableObject.