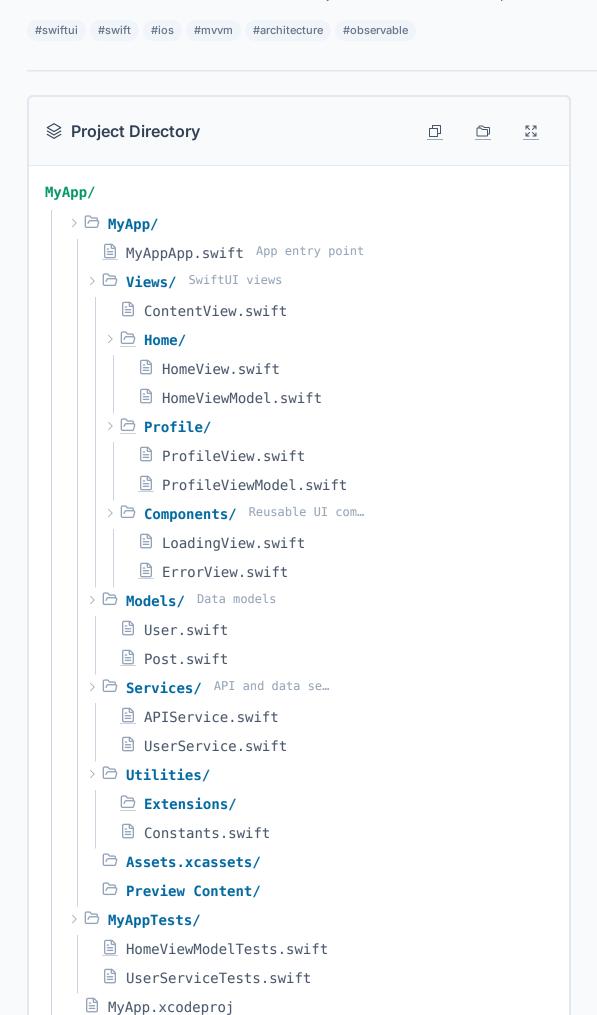


MVVM architecture with ObservableObject view models. Clean separation for testable, maintainable apps.



Why This Structure?

MVVM separates views from logic. Views observe ViewModels via @StateObject or @ObservedObject . ViewModels contain business logic and are testable without UI. Services handle external data. This pattern scales to medium-sized apps.

☑ PNG

PDF PDF

☐ Copy

</> Prompt

□ Key Directories

Views/ - SwiftUI views grouped by feature with their ViewModels

Models/ - Codable data structures for API and persistence

Services/ - Network calls, database access, external APIs

Components/ - Reusable UI pieces shared across features

```
// Views/Home/HomeViewModel.swift
@MainActor
class HomeViewModel: ObservableObject {
    @Published var posts: [Post] = []
    @Published var isLoading = false

    private let service: PostService

    func loadPosts() async {
        isLoading = true
        posts = await service.fetchPosts()
        isLoading = false
    }
}
```

</> View with ViewModel

☑ When To Use This

- Apps with 5+ screens
- Need unit-testable business logic
- Teams familiar with MVVM from other platforms
- Apps with significant async/network operations
- Medium complexity apps with clear feature boundaries

বা Trade-offs

Boilerplate - Each feature needs View + ViewModel files

Navigation complexity - Routing logic can get scattered

Dependency injection - Need manual DI or a container

Aa Naming Conventions

Views - {Feature}View.swift (HomeView.swift)
ViewModels - {Feature}ViewModel.swift (HomeViewModel.swift)
Services - {Resource}Service.swift (UserService.swift)

Models - Singular nouns (User.swift, Post.swift)

☑ Best Practices

- Use @MainActor on ViewModels for thread safety
- Inject services via initializer for testability
- Keep Views declarative—no business logic
- Use @Published for observable state only
- Group View + ViewModel in feature folders