

iOS Development Guidelines

This document is mandatory reading for all developers working on this project. Code that does not adhere to these guidelines will be rejected during code review.

Table of Contents

1. [Philosophy & Expectations](#)
 2. [Architecture Overview](#)
 3. [SOLID Principles](#)
 4. [Module-Based Architecture](#)
 5. [VIPER Pattern \(UIKit\)](#)
 6. [MVVM Pattern \(SwiftUI\)](#)
 7. [Creating Swift Packages](#)
 8. [Code Quality Standards](#)
 9. [SwiftLint & SwiftFormat Rules](#)
 10. [Testing Requirements](#)
 11. [Code Review Checklist](#)
-

Philosophy & Expectations

This project demands **professional-grade code quality**. Every line of code you write represents the team. We value:

- **Clarity over cleverness** - Code is read far more often than it is written
- **Simplicity over complexity** - The best code is the code you don't have to write
- **Consistency over personal preference** - Follow the established patterns
- **Testability over convenience** - If it can't be tested, it shouldn't be merged

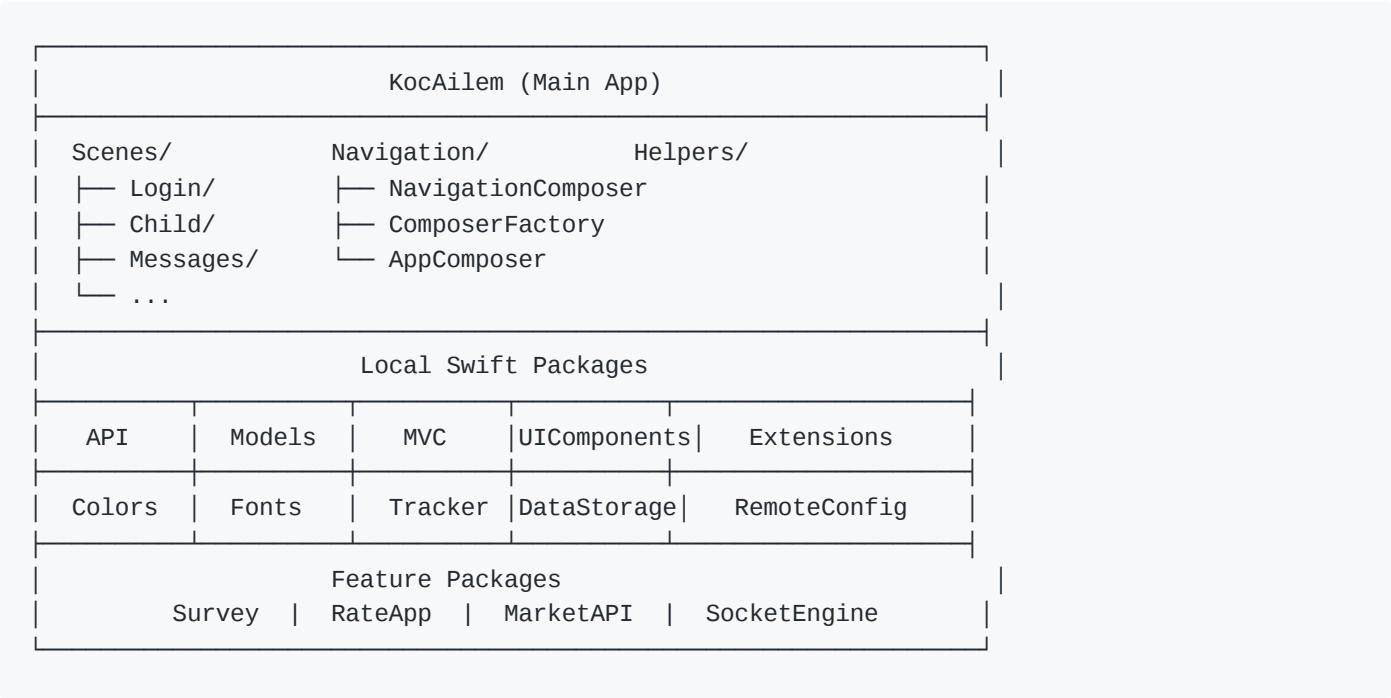
What We Expect From You

1. **Read and understand existing code** before making changes
 2. **Follow the established architecture patterns** - do not invent new ones without team consensus
 3. **Write self-documenting code** - if you need a comment to explain what code does, rewrite the code
 4. **Take ownership** - if you touch it, you own it
 5. **Review your own code** before submitting a PR as if you were reviewing someone else's work
-

Architecture Overview

This project uses a **hybrid architecture** combining:

- **VIPER** for UIKit-based screens (View, Interactor, Presenter, Router, Entity)
- **MVVM** for SwiftUI-based screens (Model, View, ViewModel)
- **Swift Package Manager (SPM)** for modular code organization



SOLID Principles

SOLID is not optional. It is the foundation of how we write code.

S - Single Responsibility Principle

A class/function should have one, and only one, reason to change.

BAD - Multiple responsibilities:

```
// This class does too many things
class UserManager {
    func fetchUser() async throws -> User { ... }
    func saveUserToDatabase(_ user: User) { ... }
    func formatUserForDisplay(_ user: User) -> String { ... }
    func validateUserEmail(_ email: String) -> Bool { ... }
    func sendWelcomeEmail(to user: User) { ... }
}
```

GOOD - Single responsibility:

```
// Each class has one job
protocol UserFetching {
    func fetchUser() async throws -> User
}
```

```

}

protocol UserPersisting {
    func save(_ user: User) throws
}

protocol UserValidating {
    func validateEmail(_ email: String) -> Bool
}

protocol UserNotifying {
    func sendWelcomeEmail(to user: User) async throws
}

// Implementations
struct UserAPIService: UserFetching {
    func fetchUser() async throws -> User {
        // Only fetches user from API
    }
}

struct UserDatabaseService: UserPersisting {
    func save(_ user: User) throws {
        // Only saves user to database
    }
}

```

For Functions - One function, one job:

```

// BAD - Function does multiple things
func processOrder(_ order: Order) {
    validateOrder(order)
    calculateTotal(order)
    applyDiscounts(order)
    saveToDatabase(order)
    sendConfirmationEmail(order)
    updateInventory(order)
}

// GOOD - Each function does one thing
func processOrder(_ order: Order) async throws {
    let validatedOrder = try validateOrder(order)
    let pricedOrder = calculateTotal(validatedOrder)
    let finalOrder = applyDiscounts(pricedOrder)

    try await orderRepository.save(finalOrder)
    try await notificationService.sendConfirmation(for: finalOrder)
    try await inventoryService.update(for: finalOrder)
}

```

O - Open/Closed Principle

Software entities should be open for extension, but closed for modification.

BAD - Requires modification for new types:

```
class DiscountCalculator {  
  func calculate(for type: String, amount: Double) -> Double {  
    switch type {  
      case "student":  
        return amount * 0.20  
      case "senior":  
        return amount * 0.15  
      case "veteran":  
        return amount * 0.25  
      // Adding new discount type requires modifying this class  
      default:  
        return 0  
    }  
  }  
}
```

GOOD - Open for extension:

```
protocol DiscountPolicy {  
  var discountPercentage: Double { get }  
  func isApplicable(to user: User) -> Bool  
}  
  
struct StudentDiscount: DiscountPolicy {  
  let discountPercentage = 0.20  
  func isApplicable(to user: User) -> Bool {  
    user.isStudent  
  }  
}  
  
struct SeniorDiscount: DiscountPolicy {  
  let discountPercentage = 0.15  
  func isApplicable(to user: User) -> Bool {  
    user.age >= 65  
  }  
}  
  
// New discounts can be added without modifying existing code  
struct VeteranDiscount: DiscountPolicy {  
  let discountPercentage = 0.25  
  func isApplicable(to user: User) -> Bool {  
    user.isVeteran  
  }  
}  
  
class DiscountCalculator {  
  private let policies: [DiscountPolicy]  
  
  init(policies: [DiscountPolicy]) {  
    self.policies = policies  
  }  
  
  func calculateDiscount(for user: User, amount: Double) -> Double {
```

```

        let applicablePolicy = policies.first { $0.isApplicable(to: user) }
        return amount * (applicablePolicy?.discountPercentage ?? 0)
    }
}

```

L - Liskov Substitution Principle

Objects of a superclass should be replaceable with objects of its subclasses without breaking the application.

BAD - Subclass breaks expected behavior:

```

class Bird {
    func fly() {
        print("Flying...")
    }
}

class Penguin: Bird {
    override func fly() {
        fatalError("Penguins can't fly!") // Violates LSP
    }
}

```

GOOD - Proper abstraction:

```

protocol Bird {
    func move()
}

protocol FlyingBird: Bird {
    func fly()
}

struct Sparrow: FlyingBird {
    func move() { fly() }
    func fly() { print("Flying...") }
}

struct Penguin: Bird {
    func move() { swim() }
    func swim() { print("Swimming...") }
}

```

I - Interface Segregation Principle

Clients should not be forced to depend on interfaces they do not use.

BAD - Fat interface:

```

protocol Worker {
    func work()
}

```

```

func eat()
func sleep()
func attendMeeting()
func writeReport()
}

// Robot is forced to implement methods it doesn't need
struct Robot: Worker {
    func work() { ... }
    func eat() { fatalError("Robots don't eat") }
    func sleep() { fatalError("Robots don't sleep") }
    func attendMeeting() { ... }
    func writeReport() { ... }
}

```

GOOD - Segregated interfaces:

```

protocol Workable {
    func work()
}

protocol Feedable {
    func eat()
}

protocol Restable {
    func sleep()
}

protocol MeetingAttendee {
    func attendMeeting()
}

struct Human: Workable, Feedable, Restable, MeetingAttendee {
    func work() { ... }
    func eat() { ... }
    func sleep() { ... }
    func attendMeeting() { ... }
}

struct Robot: Workable, MeetingAttendee {
    func work() { ... }
    func attendMeeting() { ... }
}

```

D - Dependency Inversion Principle

High-level modules should not depend on low-level modules. Both should depend on abstractions.

BAD - Direct dependency on concrete implementation:

```

class UserViewModel {
    private let apiService = APIService() // Direct dependency
}

```

```
func fetchUser() async throws -> User {
    try await apiService.getUser()
}
}
```

GOOD - Depend on abstractions:

```
protocol UserServiceProtocol {
    func getUser() async throws -> User
}

struct UserAPIService: UserServiceProtocol {
    func getUser() async throws -> User {
        // Real implementation
    }
}

struct MockUserService: UserServiceProtocol {
    func getUser() async throws -> User {
        // Mock for testing
    }
}

class UserViewModel {
    private let userService: UserServiceProtocol

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

    func fetchUser() async throws -> User {
        try await userService.getUser()
    }
}
```

Module-Based Architecture

Package Structure

Every Swift Package in this project MUST follow this structure:

```
PackageName/
├─ Package.swift
├─ Sources/
│   └─ PackageName/
│       ├── Models/
│       ├── Services/
│       ├── Views/
│       ├── Extensions/
│       └─ Utilities/
└─ Tests/
```

```
|   └─ PackageNameTests/
|       └─ ...
└─ README.md (optional)
```

Package.swift Template

```
// swift-tools-version:6.0

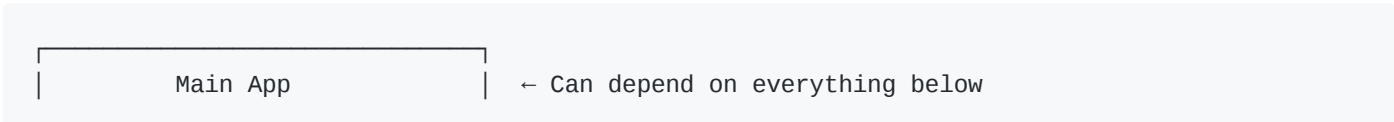
import PackageDescription

let package = Package(
    name: "PackageName",
    platforms: [.iOS(.v15)],
    products: [
        .library(
            name: "PackageName",
            targets: ["PackageName"]
        )
    ],
    dependencies: [
        // Only add dependencies that are absolutely necessary
        .package(name: "Models", path: "../Models"),
    ],
    targets: [
        .target(
            name: "PackageName",
            dependencies: ["Models"],
            swiftSettings: swiftSettings
        ),
        .testTarget(
            name: "PackageNameTests",
            dependencies: ["PackageName"],
            swiftSettings: swiftSettings
        )
    ]
)

var swiftSettings: [SwiftSetting] { [
    .enableUpcomingFeature("DisableOutwardActorInference"),
    .enableExperimentalFeature("StrictConcurrency"),
    .swiftLanguageMode(.v5)
] }
```

Dependency Rules

1. **Minimize dependencies** - Every dependency is a liability
2. **No circular dependencies** - Package A cannot depend on Package B if B depends on A
3. **Depend on abstractions** - Import protocols, not implementations when possible
4. **Layer dependencies flow downward:**



Feature Packages	← Can depend on Core only
Core Packages (Models, Extensions, API)	← Can depend on Foundation only
Foundation Packages (Colors, Fonts, Tracker)	← No internal dependencies

VIPER Pattern (UIKit)

Every UIKit scene MUST follow the VIPER pattern with this structure:

```
SceneName/
├─ SceneNameViewController.swift    # View
├─ SceneNameInteractor.swift       # Business Logic
├─ SceneNamePresenter.swift        # Presentation Logic
├─ SceneNameRouter.swift           # Navigation
├─ SceneNameWorker.swift           # Data Operations
└─ SceneNameModels.swift           # Data Transfer Objects
```

Models (SceneNameModels.swift)

```
import Foundation

enum SceneName {

    // MARK: - Use Cases

    enum FetchData {
        struct Request {
            let id: String
        }

        struct Response {
            let data: SomeDTO
            let timestamp: Date
        }

        struct ViewModel {
            let title: String
            let subtitle: String
            let formattedDate: String
        }
    }

    enum SubmitAction {
        struct Request {
            let input: String
        }
    }
}
```

```

    struct Response {
        let success: Bool
        let message: String
    }

    struct ViewModel {
        let alertTitle: String
        let alertMessage: String
    }
}

```

View (SceneNameViewController.swift)

```

import UIKit

// MARK: - Display Logic Protocol

protocol SceneNameDisplayLogic: AnyObject {
    func displayData(_ viewModel: SceneName.FetchData.ViewModel)
    func displaySubmitResult(_ viewModel: SceneName.SubmitAction.ViewModel)
    func displayError(_ message: String)
    func displayLoading(_ isLoading: Bool)
}

// MARK: - View Controller

final class SceneNameViewController: UIViewController {

    // MARK: - Properties

    var interactor: SceneNameBusinessLogic?
    var router: (SceneNameRoutingLogic & SceneNameDataPassing)?

    // MARK: - UI Components

    private lazy var titleLabel: UILabel = {
        let label = UILabel()
        label.translatesAutoresizingMaskIntoConstraints = false
        return label
    }()

    // MARK: - Lifecycle

    override func viewDidLoad() {
        super.viewDidLoad()
        setupUI()
        fetchData()
    }

    // MARK: - Setup

    private func setupUI() {
        view.backgroundColor = .systemBackground
    }
}

```

```

        // Layout code
    }

    // MARK: - Actions

    private func fetchData() {
        let request = SceneName.FetchData.Request(id: "123")
        interactor?.fetchData(request)
    }

    @objc private func submitButtonTapped() {
        let request = SceneName.SubmitAction.Request(input: "data")
        interactor?.submitAction(request)
    }
}

// MARK: - Display Logic

extension SceneNameViewController: SceneNameDisplayLogic {

    func displayData(_ viewModel: SceneName.FetchData.ViewModel) {
        titleLabel.text = viewModel.title
    }

    func displaySubmitResult(_ viewModel: SceneName.SubmitAction.ViewModel) {
        // Show alert
    }

    func displayError(_ message: String) {
        // Show error
    }

    func displayLoading(_ isLoading: Bool) {
        // Show/hide loading indicator
    }
}

```

Interactor (SceneNameInteractor.swift)

```

import Foundation

// MARK: - Business Logic Protocol

protocol SceneNameBusinessLogic {
    func fetchData(_ request: SceneName.FetchData.Request)
    func submitAction(_ request: SceneName.SubmitAction.Request)
}

// MARK: - Data Store Protocol

protocol SceneNameDataStore {
    var selectedItem: ItemModel? { get set }
}

// MARK: - Interactor

```

```

final class SceneNameInteractor: SceneNameBusinessLogic, SceneNameDataStore {

    // MARK: - Properties

    var presenter: SceneNamePresentationLogic?
    var worker: SceneNameWorkingLogic
    var selectedItem: ItemModel?

    // MARK: - Initialization

    init(worker: SceneNameWorkingLogic = SceneNameWorker()) {
        self.worker = worker
    }

    // MARK: - Business Logic

    func fetchData(_ request: SceneName.FetchData.Request) {
        Task { @MainActor in
            do {
                let data = try await worker.fetchData(id: request.id)
                let response = SceneName.FetchData.Response(
                    data: data,
                    timestamp: Date()
                )
                presenter?.presentData(response)
            } catch {
                presenter?.presentError(error)
            }
        }
    }

    func submitAction(_ request: SceneName.SubmitAction.Request) {
        Task { @MainActor in
            do {
                let result = try await worker.submitData(request.input)
                let response = SceneName.SubmitAction.Response(
                    success: result.success,
                    message: result.message
                )
                presenter?.presentSubmitResult(response)
            } catch {
                presenter?.presentError(error)
            }
        }
    }
}

```

Presenter (SceneNamePresenter.swift)

```

import Foundation

// MARK: - Presentation Logic Protocol

protocol SceneNamePresentationLogic {

```

```

func presentData(_ response: SceneName.FetchData.Response)
func presentSubmitResult(_ response: SceneName.SubmitAction.Response)
func presentError(_ error: Error)
}

// MARK: - Presenter

final class SceneNamePresenter: SceneNamePresentationLogic {

    // MARK: - Properties

    weak var viewController: SceneNameDisplayLogic?

    private let dateFormatter: DateFormatter = {
        let formatter = DateFormatter()
        formatter.dateStyle = .medium
        formatter.timeStyle = .short
        return formatter
    }()

    // MARK: - Presentation Logic

    func presentData(_ response: SceneName.FetchData.Response) {
        let viewModel = SceneName.FetchData.ViewModel(
            title: response.data.title,
            subtitle: response.data.description,
            formattedDate: dateFormatter.string(from: response.timestamp)
        )
        viewController?.displayData(viewModel)
    }

    func presentSubmitResult(_ response: SceneName.SubmitAction.Response) {
        let viewModel = SceneName.SubmitAction.ViewModel(
            alertTitle: response.success ? "Success" : "Error",
            alertMessage: response.message
        )
        viewController?.displaySubmitResult(viewModel)
    }

    func presentError(_ error: Error) {
        viewController?.displayError(error.localizedDescription)
    }
}

```

Router (SceneNameRouter.swift)

```

import UIKit

// MARK: - Routing Logic Protocol

protocol SceneNameRoutingLogic {
    func routeToDetail()
    func routeToSettings()
}

```

```

// MARK: - Data Passing Protocol

protocol SceneNameDataPassing {
    var datastore: SceneNameDataStore? { get }
}

// MARK: - Router

final class SceneNameRouter: SceneNameRoutingLogic, SceneNameDataPassing {

    // MARK: - Properties

    weak var viewController: SceneNameViewController?
    var datastore: SceneNameDataStore?

    // MARK: - Routing

    func routeToDetail() {
        let destinationVC = DetailViewController()
        var destinationDS = destinationVC.router?.dataStore
        passDataToDetail(source: datastore, destination: &destinationDS)
        navigateToDetail(source: viewController, destination: destinationVC)
    }

    func routeToSettings() {
        let destinationVC = SettingsViewController()
        navigateToSettings(source: viewController, destination: destinationVC)
    }

    // MARK: - Navigation

    private func navigateToDetail(source: SceneNameViewController?, destination: DetailViewCont
        source?.navigationController?.pushViewController(destination, animated: true)
    }

    private func navigateToSettings(source: SceneNameViewController?, destination: SettingsView
        source?.present(destination, animated: true)
    }

    // MARK: - Data Passing

    private func passDataToDetail(source: SceneNameDataStore?, destination: inout DetailDataSto
        destination?.item = source?.selectedItem
    }
}

```

Worker (SceneNameWorker.swift)

```

import Foundation

// MARK: - Working Logic Protocol

protocol SceneNameWorkingLogic {
    func fetchData(id: String) async throws -> SomeDTO
    func submitData(_ input: String) async throws -> SubmitResult
}

```

```

}

// MARK: - Worker

struct SceneNameWorker: SceneNameWorkingLogic {

    // MARK: - Dependencies

    private let apiService: APIServiceProtocol

    // MARK: - Initialization

    init(apiService: APIServiceProtocol = APIService.shared) {
        self.apiService = apiService
    }

    // MARK: - Working Logic

    func fetchData(id: String) async throws -> SomeDTO {
        try await apiService.fetch(endpoint: .getData(id: id))
    }

    func submitData(_ input: String) async throws -> SubmitResult {
        try await apiService.post(endpoint: .submit(data: input))
    }
}

// MARK: - Mock Worker (for Testing)

final class MockSceneNameWorker: SceneNameWorkingLogic {

    var fetchDataResult: Result<SomeDTO, Error> = .failure(NSError())
    var submitDataResult: Result<SubmitResult, Error> = .failure(NSError())

    func fetchData(id: String) async throws -> SomeDTO {
        try fetchDataResult.get()
    }

    func submitData(_ input: String) async throws -> SubmitResult {
        try submitDataResult.get()
    }
}

```

MVVM Pattern (SwiftUI)

For SwiftUI views, use MVVM with clear separation:

FeatureName/	
├ FeatureNameView.swift	# SwiftUI View
├ FeatureNameViewModel.swift	# Observable ViewModel
├ FeatureNameModel.swift	# Models/DTOs
└ FeatureNameService.swift	# Service Protocol & Implementation

ViewModel (FeatureNameViewModel.swift)

```
import Foundation
import SwiftUI

// MARK: - ViewModel

@MainActor
final class FeatureNameViewModel: ObservableObject {

    // MARK: - Published Properties

    @Published private(set) var items: [ItemModel] = []
    @Published private(set) var isLoading = false
    @Published private(set) var errorMessage: String?
    @Published var searchText = ""

    // MARK: - Dependencies

    private let service: FeatureNameServiceProtocol

    // MARK: - Computed Properties

    var filteredItems: [ItemModel] {
        guard !searchText.isEmpty else { return items }
        return items.filter { $0.title.localizedCaseInsensitiveContains(searchText) }
    }

    var hasError: Bool {
        errorMessage != nil
    }

    // MARK: - Initialization

    init(service: FeatureNameServiceProtocol = FeatureNameService()) {
        self.service = service
    }

    // MARK: - Public Methods

    func loadData() async {
        isLoading = true
        errorMessage = nil

        do {
            items = try await service.fetchItems()
        } catch {
            errorMessage = error.localizedDescription
        }

        isLoading = false
    }

    func deleteItem(_ item: ItemModel) async {
        do {
            try await service.deleteItem(item.id)
        }
    }
}
```



```

        items.removeAll { $0.id == item.id }
    } catch {
        errorMessage = error.localizedDescription
    }
}

func clearError() {
    errorMessage = nil
}
}

```

View (FeatureNameView.swift)

```

import SwiftUI

struct FeatureNameView: View {

    // MARK: - Properties

    @StateObject private var viewModel: FeatureNameViewModel
    @Environment(\.dismiss) private var dismiss

    // MARK: - Initialization

    init(viewModel: FeatureNameViewModel = FeatureNameViewModel()) {
        _viewModel = StateObject(wrappedValue: viewModel)
    }

    // MARK: - Body

    var body: some View {
        NavigationStack {
            content
                .navigationTitle("Feature Name")
                .searchable(text: $viewModel.searchText)
                .task { await viewModel.loadData() }
                .alert("Error", isPresented: .constant(viewModel.hasError)) {
                    Button("OK") { viewModel.clearError() }
                } message: {
                    Text(viewModel.errorMessage ?? "")
                }
        }
    }

    // MARK: - Content

    @ViewBuilder
    private var content: some View {
        if viewModel.isLoading {
            loadingView
        } else if viewModel.filteredItems.isEmpty {
            emptyView
        } else {
            listView
        }
    }
}

```

```

}

private var loadingView: some View {
    ProgressView()
        .frame(maxWidth: .infinity, maxHeight: .infinity)
}

private var emptyView: some View {
    ContentUnavailableView(
        "No Items",
        systemImage: "tray",
        description: Text("Items will appear here")
    )
}

private var listView: some View {
    List {
        ForEach(viewModel.filteredItems) { item in
            ItemRow(item: item)
        }
        .onDelete(perform: deleteItems)
    }
}

// MARK: - Actions

private func deleteItems(at offsets: IndexSet) {
    Task {
        for index in offsets {
            await viewModel.deleteItem(viewModel.filteredItems[index])
        }
    }
}

// MARK: - Subviews

private struct ItemRow: View {
    let item: ItemModel

    var body: some View {
        VStack(alignment: .leading, spacing: 4) {
            Text(item.title)
                .font(.headline)
            Text(item.subtitle)
                .font(.subheadline)
                .foregroundColor(.secondary)
        }
    }
}

// MARK: - Preview

#Preview {
    FeatureNameView()
}

```

Service (FeatureNameService.swift)

```
import Foundation

// MARK: - Service Protocol

protocol FeatureNameServiceProtocol: Sendable {
    func fetchItems() async throws -> [ItemModel]
    func deleteItem(_ id: String) async throws
}

// MARK: - Service Implementation

struct FeatureNameService: FeatureNameServiceProtocol {

    private let networking: NetworkingProtocol

    init(networking: NetworkingProtocol = NetworkingService.shared) {
        self.networking = networking
    }

    func fetchItems() async throws -> [ItemModel] {
        try await networking.fetch(endpoint: .items)
    }

    func deleteItem(_ id: String) async throws {
        try await networking.delete(endpoint: .item(id: id))
    }
}

// MARK: - Mock Service (for Testing & Previews)

final class MockFeatureNameService: FeatureNameServiceProtocol, @unchecked Sendable {

    var itemsToReturn: [ItemModel] = []
    var shouldThrowError = false
    var deleteWasCalled = false

    func fetchItems() async throws -> [ItemModel] {
        if shouldThrowError {
            throw NSError(domain: "Mock", code: 1)
        }
        return itemsToReturn
    }

    func deleteItem(_ id: String) async throws {
        if shouldThrowError {
            throw NSError(domain: "Mock", code: 1)
        }
        deleteWasCalled = true
    }
}
```

Creating Swift Packages

When to Create a New Package

Create a new Swift Package when:

- 1. **Functionality is reusable** across multiple features
- 2. **Code has clear boundaries** and minimal external dependencies
- 3. **Testing in isolation** would be beneficial
- 4. **Build times can be improved** by separating compilation units

Package Categories

Category	Examples	Purpose
Foundation	Colors, Fonts, Extensions	Basic utilities, no internal deps
Core	Models, API, DataStorage	Core functionality
Feature	Survey, RateApp, SocketEngine	Isolated features
UI	UIComponents, MVC	Reusable UI elements

Step-by-Step: Creating a New Package

- 1. **Create the directory structure:**

```
NewPackage/  
├─ Package.swift  
├─ Sources/  
│   └─ NewPackage/  
│       └─ NewPackage.swift  
└─ Tests/  
    └─ NewPackageTests/  
        └─ NewPackageTests.swift
```

- 2. **Create Package.swift:**

```
// swift-tools-version:6.0  
  
import PackageDescription  
  
let package = Package(  
    name: "NewPackage",  
    platforms: [.iOS(.v15)],  
    products: [  
        .library(  
            name: "NewPackage",  
            targets: ["NewPackage"]  
        )  
    ],
```

```

dependencies: [
    // Keep dependencies minimal
],
targets: [
    .target(
        name: "NewPackage",
        dependencies: [],
        swiftSettings: swiftSettings
    ),
    .testTarget(
        name: "NewPackageTests",
        dependencies: ["NewPackage"],
        swiftSettings: swiftSettings
    )
]
)

var swiftSettings: [SwiftSetting] { [
    .enableUpcomingFeature("DisableOutwardActorInference"),
    .enableExperimentalFeature("StrictConcurrency"),
    .swiftLanguageMode(.v5)
] }

```

3. Add to the main project:

- In Xcode, right-click on the project
- Select "Add Package Dependencies..."
- Choose "Add Local..."
- Select your new package directory

Example: Localization Package

```

Localization/
├─ Package.swift
├─ Sources/
│   └─ Localization/
│       ├── L10n.swift
│       ├── Resources/
│       │   ├── en.lproj/
│       │   │   └─ Localizable.strings
│       │   └─ tr.lproj/
│       │       └─ Localizable.strings
│       └─ String+Localization.swift
└─ Tests/
    └─ LocalizationTests/
        └─ LocalizationTests.swift

```

L10n.swift:

```

import Foundation

public enum L10n {
    public enum Common {

```

```

        public static let ok = "common.ok".localized
        public static let cancel = "common.cancel".localized
        public static let error = "common.error".localized
    }

    public enum Login {
        public static let title = "login.title".localized
        public static let emailPlaceholder = "login.email.placeholder".localized
        public static func welcomeMessage(_ name: String) -> String {
            String(format: "login.welcome.message".localized, name)
        }
    }
}

```

String+Localization.swift:

```

import Foundation

extension String {
    var localized: String {
        NSLocalizedString(self, bundle: .module, comment: "")
    }
}

```

Example: Analytics Package

```

// AnalyticsEvent.swift
public protocol AnalyticsEvent {
    var name: String { get }
    var parameters: [String: Any] { get }
}

// AnalyticsProvider.swift
public protocol AnalyticsProvider {
    func track(_ event: AnalyticsEvent)
    func setUserProperty(_ value: String?, forName name: String)
}

// AnalyticsService.swift
public final class AnalyticsService {
    public static let shared = AnalyticsService()

    private var providers: [AnalyticsProvider] = []

    private init() {}

    public func register(_ provider: AnalyticsProvider) {
        providers.append(provider)
    }

    public func track(_ event: AnalyticsEvent) {
        providers.forEach { $0.track(event) }
    }
}

```

Code Quality Standards

File Organization

Every Swift file MUST follow this structure:

```
import Foundation
import UIKit

// MARK: - Protocols (if defining protocols)

protocol SomeProtocol {
    // ...
}

// MARK: - Type Definition

final class ClassName {

    // MARK: - Constants

    private enum Constants {
        static let animationDuration: TimeInterval = 0.3
        static let cornerRadius: CGFloat = 8
    }

    // MARK: - Properties

    private let dependency: DependencyProtocol
    private var state: State = .idle

    // MARK: - UI Components (if applicable)

    private lazy var button: UIButton = {
        let button = UIButton(type: .system)
        return button
    }()

    // MARK: - Initialization

    init(dependency: DependencyProtocol) {
        self.dependency = dependency
    }

    // MARK: - Public Methods

    func doSomething() {
        // ...
    }

    // MARK: - Private Methods
```

```

    private func helperMethod() {
        // ...
    }
}

// MARK: - Protocol Conformance

extension ClassName: SomeProtocol {
    // ...
}

// MARK: - Nested Types

extension ClassName {
    enum State {
        case idle
        case loading
        case loaded
    }
}

```

Naming Conventions

Type	Convention	Example
Classes/Structs	PascalCase	UserProfileViewController
Protocols	PascalCase + suffix	UserServiceProtocol, Configurable
Functions	camelCase, verb-first	fetchUser(), validateEmail()
Variables	camelCase	userName, isLoading
Constants	camelCase	maximumRetryCount
Enum cases	camelCase	.loading, .error(message:)
Type aliases	PascalCase	typealias CompletionHandler = () -> Void

Access Control

Apply the **most restrictive access level** possible:

```

// Public - Only for APIs meant for external use
public struct PublicModel { }

// Internal (default) - Accessible within the module
struct InternalModel { }

// Private - Accessible only within the enclosing declaration
private func helperMethod() { }

// File-private - Accessible within the same file
fileprivate extension SomeType { }

```

SwiftLint & SwiftFormat Rules

SwiftLint Configuration (.swiftlint.yml)

```
# Disabled Rules
disabled_rules:
  - trailing_whitespace
  - identifier_name
  - inclusive_language

# Excluded Paths
excluded:
  - Pods
  - KocAilemTests
  - reports
  - vendor
  - .build
  - fastlane
  - Build
  - Carthage
  - Frameworks/Sources/Frameworks/LockSmith
  - Frameworks/Sources/Frameworks/PieCharts
  - Frameworks/Sources/Frameworks/Karte
  - Frameworks/Sources/Frameworks/ImageSlideShow
  - SPM

# Opt-in Rules (IMPORTANT - Enable these!)
opt_in_rules:
  - array_init
  - closure_body_length
  - closure_end_indentation
  - closure_spacing
  - collection_alignment
  - contains_over_filter_count
  - contains_over_filter_is_empty
  - contains_over_first_not_nil
  - contains_over_range_nil_comparison
  - discouraged_assert
  - discouraged_object_literal
  - empty_collection_literal
  - empty_count
  - empty_string
  - enum_case_associated_values_count
  - explicit_init
  - fallthrough
  - fatal_error_message
  - file_name
  - first_where
  - flatmap_over_map_reduce
  - force_unwrapping
  - function_default_parameter_at_end
  - identical_operands
  - implicitly_unwrapped_optional
  - joined_default_parameter
```

- last_where
- legacy_multiple
- legacy_random
- literal_expression_end_indentation
- lower_acl_than_parent
- modifier_order
- multiline_arguments
- multiline_function_chains
- multiline_literal_brackets
- multiline_parameters
- nimble_operator
- nslocalizedstring_key
- number_separator
- operator_usage_whitespace
- optional_enum_case_matching
- overridden_super_call
- override_in_extension
- pattern_matching_keywords
- prefer_self_type_over_type_of_self
- prefer_zero_over_explicit_init
- private_action
- private_outlet
- prohibited_super_call
- reduce_into
- redundant_nil_coalescing
- redundant_type_annotation
- single_test_class
- sorted_first_last
- sorted_imports
- static_operator
- strong_iboutlet
- toggle_bool
- unavailable_function
- unneeded_parentheses_in_closure_argument
- unowned_variable_capture
- untyped_error_in_catch
- vertical_parameter_alignment_on_call
- vertical_whitespace_closing_braces
- vertical_whitespace_opening_braces
- xct_specific_matcher
- yoda_condition

Identifier Name Rules

identifier_name:

min_length:

error: 2

max_length:

warning: 50

error: 60

excluded:

- id
- ok
- to
- x
- y
- i
- j

```
# Type Name Rules
type_name:
  min_length: 3
  max_length:
    warning: 50
    error: 60

# LINE LENGTH - 120 characters max
line_length:
  warning: 120
  error: 150
  ignores_comments: true
  ignores_urls: true
  ignores_interpolated_strings: true

# FILE LENGTH - 400 lines max
file_length:
  warning: 400
  error: 500
  ignore_comment_only_lines: true

# TYPE BODY LENGTH - 300 lines warning, 400 lines error
type_body_length:
  warning: 300
  error: 400

# FUNCTION BODY LENGTH - Keep functions small!
function_body_length:
  warning: 30
  error: 50

# CYCLOMATIC COMPLEXITY - Functions should be simple
cyclomatic_complexity:
  warning: 5
  error: 10
  ignores_case_statements: false

# FUNCTION PARAMETER COUNT - Too many = code smell
function_parameter_count:
  warning: 4
  error: 6
  ignores_default_parameters: true

# NESTING LEVEL - Avoid deep nesting
nesting:
  type_level:
    warning: 2
    error: 3
  function_level:
    warning: 3
    error: 4

# CLOSURE BODY LENGTH
closure_body_length:
  warning: 25
  error: 40
```

```
# ENUM CASE ASSOCIATED VALUES COUNT
enum_case_associated_values_count:
  warning: 4
  error: 6

# LARGE TUPLE
large_tuple:
  warning: 3
  error: 4

# FORCE CAST - Absolutely forbidden in production code
force_cast: error

# FORCE TRY - Absolutely forbidden in production code
force_try: error

# FORCE UNWRAPPING - Warning (sometimes necessary but discouraged)
force_unwrapping:
  severity: warning

# IMPLICITLY UNWRAPPED OPTIONAL - Warning
implicitly_unwrapped_optional:
  severity: warning
  mode: all_except_iboutlets

# TRAILING CLOSURE - Prefer trailing closure syntax
trailing_closure:
  only_single_muted_parameter: true

# VERTICAL WHITESPACE
vertical_whitespace:
  max_empty_lines: 1

# COMMA - Proper spacing
comma: error

# COLON - Proper spacing
colon:
  severity: error
  flexible_right_spacing: false
  apply_to_dictionaries: true

# OPENING BRACE
opening_brace:
  severity: error

# STATEMENT POSITION
statement_position:
  severity: error
  statement_mode: default

# RETURN ARROW WHITESPACE
return_arrow_whitespace: error

# Custom Rules
custom_rules:
```

```
no_print_statements:  
  name: "No Print Statements"  
  regex: "\\bprint\\s*\\s*("  
  message: "Use proper logging instead of print statements"  
  severity: warning  
  
no_hardcoded_strings_in_ui:  
  name: "No Hardcoded Strings"  
  regex: '(?!\\|\\.)(Text|Label|title|message|placeholder)\\s*[:=]\\s*"^[^"]*[a-zA-Z][^"]*"'  
  message: "Use localized strings for UI text"  
  severity: warning  
  
mark_comment_format:  
  name: "MARK Comment Format"  
  regex: "// MARK: -?[^\n]"  
  message: "MARK comments should have a space after the colon"  
  severity: warning  
  
todo_without_name:  
  name: "TODO Without Name"  
  regex: "// TODO:[^\n]"  
  message: "TODOs should include author name: // TODO: [AuthorName] Description"  
  severity: warning
```

SwiftFormat Configuration (.swiftformat)

```
# SwiftFormat Configuration
# Run: swiftformat . --config .swiftformat

# Format Options
--swiftversion 5.9

# Indentation
--indent 4
--tabwidth 4
--smarttabs enabled
--indentcase false
--ifdef no-indent

# Line Breaks
--linebreaks lf
--maxwidth 120
--wraparguments before-first
--wrapparameters before-first
--wrapcollections before-first
--wrapreturntype preserve
--wrapconditions after-first
--wraptypealiases before-first

# Spacing
--operatorfunc spaced
--nospaceoperators
--ranges spaced
--trimwhitespace always
```

```
# Braces
--allman false
--elseposition same-line
--guardelse next-line

# Semicolons
--semicolons never

# Imports
--importgrouping testable-bottom
--sortedimports true

# Attributes
--funcattributes prev-line
--typeattributes prev-line
--varattributes same-line

# Other
--self remove
--selfrequired
--stripunusedargs closure-only
--header ignore
--marktypes never
--markextensions always
--extensionacl on-declarations
--redundanttype inferred

# File Exclusions
--exclude Build,Pods, .build, Carthage, SPM, vendor

# Disabled Rules
--disable wrapMultilineStatementBraces
--disable trailingCommas
--disable andOperator
--disable preferForLoop
--disable blankLinesBetweenImports
--disable blankLinesAtEndOfScope
--disable blankLinesAtStartOfScope

# Enabled Rules (explicitly)
--enable isEmpty
--enable sortedSwitchCases
--enable wrapEnumCases
--enable wrapSwitchCases
--enable markTypes
--enable organizeDeclarations
--enable blankLineAfterImports
```

Testing Requirements (Optional, but preferred)

Unit Testing Guidelines

1. Every public method must have at least one test

2. Use dependency injection to enable mocking

3. Follow the Arrange-Act-Assert pattern

4. Name tests descriptively: `test_methodName_condition_expectedResult`

```
import XCTest
@testable import FeatureName

final class FeatureViewModelTests: XCTestCase {

    // MARK: - Properties

    private var sut: FeatureViewModel!
    private var mockService: MockFeatureService!

    // MARK: - Setup & Teardown

    override func setUp() {
        super.setUp()
        mockService = MockFeatureService()
        sut = FeatureViewModel(service: mockService)
    }

    override func tearDown() {
        sut = nil
        mockService = nil
        super.tearDown()
    }

    // MARK: - Tests

    func test_loadData_whenServiceSucceeds_updatesItems() async {
        // Arrange
        let expectedItems = [ItemModel(id: "1", title: "Test")]
        mockService.itemsToReturn = expectedItems

        // Act
        await sut.loadData()

        // Assert
        XCTAssertEqual(sut.items, expectedItems)
        XCTAssertFalse(sut.isLoading)
        XCTAssertNil(sut.errorMessage)
    }

    func test_loadData_whenServiceFails_setsErrorMessage() async {
        // Arrange
        mockService.shouldThrowError = true

        // Act
        await sut.loadData()

        // Assert
        XCTAssertTrue(sut.items.isEmpty)
        XCTAssertFalse(sut.isLoading)
        XCTAssertNotNil(sut.errorMessage)
    }
}
```

```

func test_deleteItem_removesItemFromList() async {
    // Arrange
    let item = ItemModel(id: "1", title: "Test")
    mockService.itemsToReturn = [item]
    await sut.loadData()

    // Act
    await sut.deleteItem(item)

    // Assert
    XCTAssertTrue(sut.items.isEmpty)
    XCTAssertTrue(mockService.deleteWasCalled)
}
}

```

Code Review Checklist

Before submitting a PR, verify:

Architecture & SOLID

- ☐ Single Responsibility: Each class/function has one job
- ☐ Open/Closed: New features extend, not modify existing code
- ☐ Liskov Substitution: Subclasses are substitutable for base classes
- ☐ Interface Segregation: No fat interfaces; protocols are focused
- ☐ Dependency Inversion: Depends on protocols, not concrete types

Code Quality

- ☐ No force unwrapping (`!`) without justification
- ☐ No force try (`try!`) in production code
- ☐ No force cast (`as!`) in production code
- ☐ No `print()` statements (use proper logging)
- ☐ No hardcoded strings in UI (use localization)
- ☐ No magic numbers (use named constants)
- ☐ No commented-out code
- ☐ All `TODO` s include author name and ticket number

File Organization

- ☐ File length under 400 lines
- ☐ Class/struct body under 300 lines
- ☐ Functions under 30 lines
- ☐ Cyclomatic complexity under 5
- ☐ Proper MARK comments for sections

- [] Imports sorted and organized

Testing (Optional, but preferred)

- [] Unit tests for new functionality
- [] Mock objects use protocols
- [] Tests follow Arrange-Act-Assert pattern
- [] No flaky tests
- [] All tests pass locally

Documentation

- [] Complex logic has explanatory comments
- [] Public APIs have documentation comments
- [] README updated if needed

Quick Reference Card

CODE LIMITS AT A GLANCE

Line Length:	120 chars (warning), 150 (error)
File Length:	400 lines (warning), 500 (error)
Type Body Length:	300 lines (warning), 400 (error)
Function Body Length:	30 lines (warning), 50 (error)
Cyclomatic Complexity:	5 (warning), 10 (error)
Function Parameters:	4 (warning), 6 (error)
Nesting Level:	3 (warning), 4 (error)
Closure Body Length:	25 lines (warning), 40 (error)

FORBIDDEN

- ✗ `force_cast (as!)`
- ✗ `force_try (try!)`
- ✗ `print()` statements
- ✗ Hardcoded UI strings
- ✗ Commented-out code
- ✗ Magic numbers
- ✗ God classes (classes that do everything)
- ✗ Circular dependencies between packages

REQUIRED

- ✓ Protocol-based dependencies
- ✓ MARK comments for file organization
- ✓ Unit tests for business logic (Optional, but preferred)
- ✓ Localized strings for UI
- ✓ `async/await` for asynchronous code
- ✓ `@MainActor` for UI updates
- ✓ Dependency injection

✓ Single responsibility per class/function

Final Words

Good code is not about being clever. It's about being clear.

Every line you write will be read by another developer (including future you). Write code that you would be proud to show. Write code that explains itself. Write code that follows these guidelines.

When in doubt:

1. Follow the existing patterns in the codebase
2. Ask the team
3. Keep it simple

Remember: **We are professionals. Our code reflects our standards.**

Last Updated: December 2024 Version: 1.0