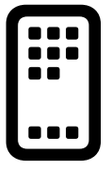


SwiftUI

Declarative. Powerful. Revolutionary

What is SwiftUI?

1. Modern, declarative framework for building user interfaces across Apple platforms
2. Released in 2019 as a successor to UIKit
3. Single codebase for iOS, iPadOS, macOS, and watchOS
4. Built with Swift programming language

Why SwiftUI?

1. Faster Development
2. Less code
3. Real-time preview
4. Declarative Syntax
5. Automatic State Management
6. Composition Based
7. Built-in Animation

Project Structure

1. ProjectNameApp.swift
2. ContentView.swift
3. Assets.xcassets
4. Preview Content/
5. Info.plist

```swift
@main
struct HelloWorldApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}
```

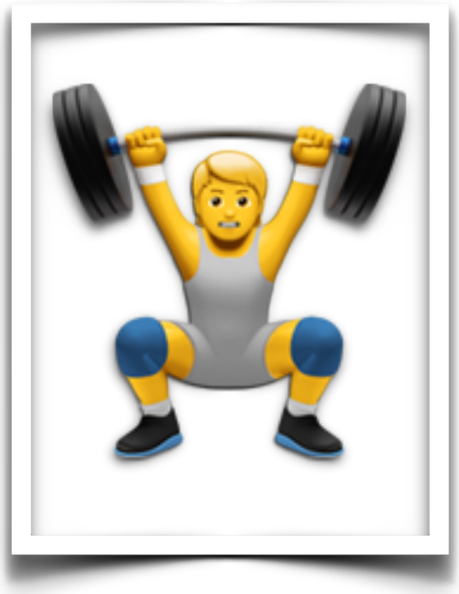
Declarative vs Imperative

```
VStack {
    Text("Hello")
    Button("Tap me") {
        // action
    }
}
```

```
let label = UILabel()
label.text = "Hello"
label.frame = CGRect(...)
view.addSubview(label)
```

*Describe what you want **vs** Describe how to do it*

UIKit vs SwiftUI



UIKit
Storyboards and constraints

SwiftUI
Casting UI spells with SwiftUI

SwiftUI: Where UI challenges disappear faster than you can say "Abracadabra!" 🎩✨

Basic Views

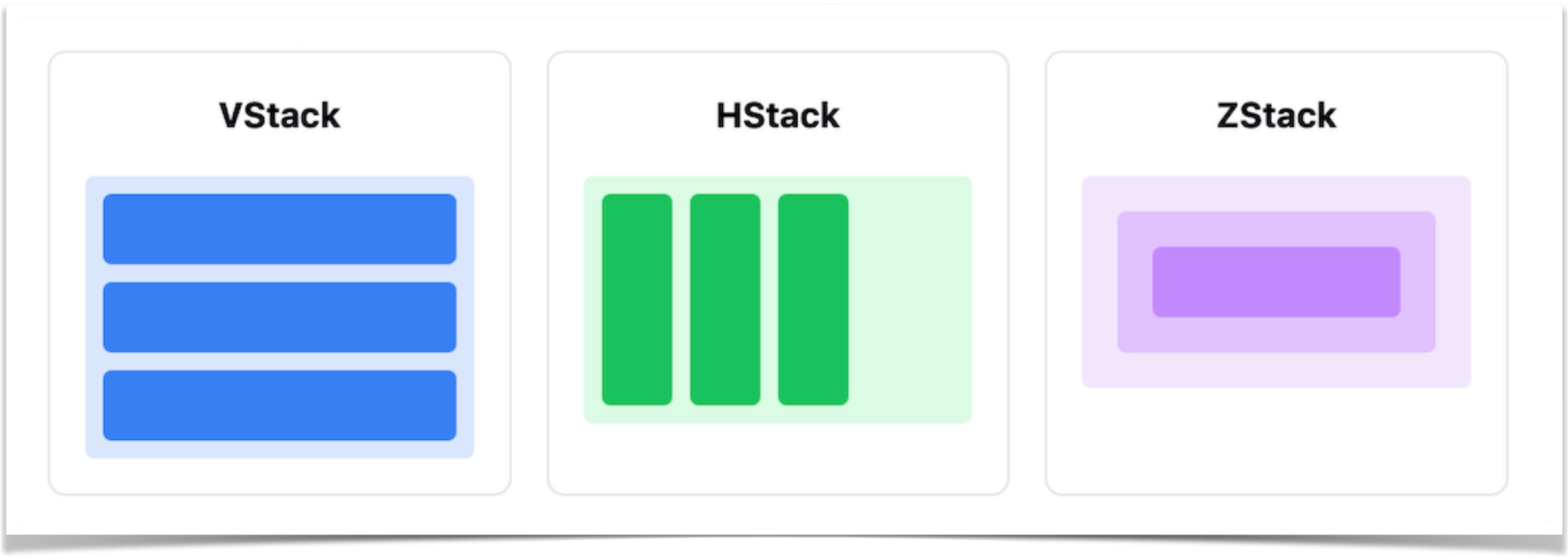
1. Text
2. Image
3. Button
4. TextField
5. Toggle

Basic Modifiers

1. .font()
2. .foregroundColor()
3. .padding()
4. .background()
5. .frame()

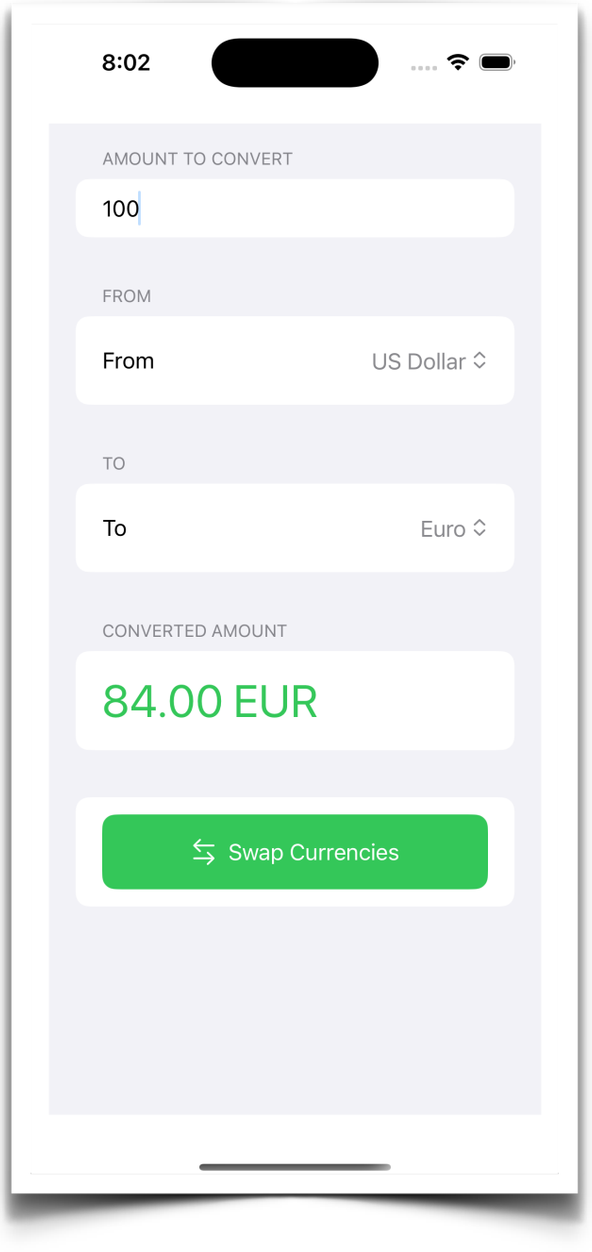
```
Text("Hello, World!")
    .font(.title)
    .foregroundColor(.blue)
    .padding()
```

Layout System



Form Controls in SwiftUI

1. Form - Container for form elements
2. Section - Grouped content
3. TextField - Text input
4. Picker - Selection control

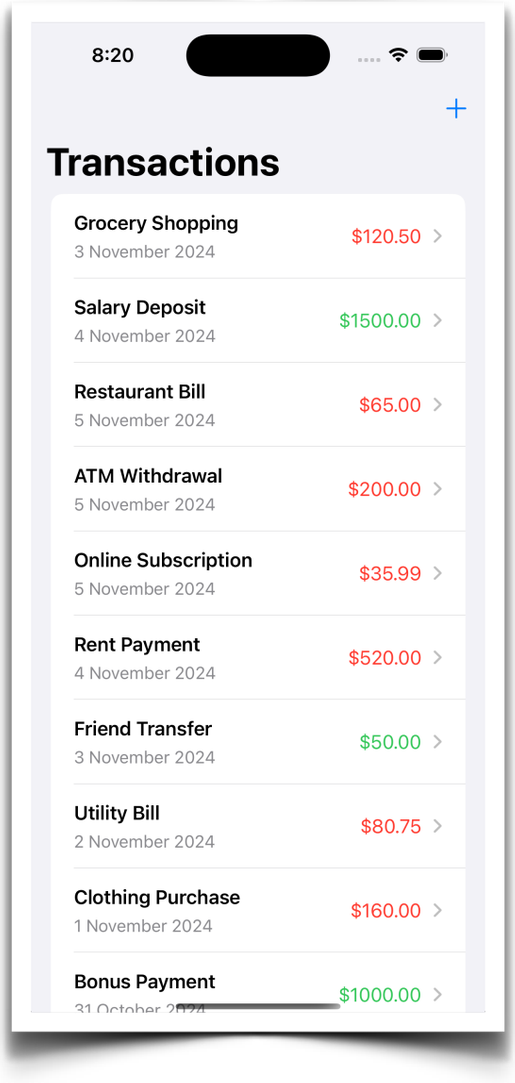


Form Controls in SwiftUI

```
Form {
    Section(header: Text("Amount to convert")) {
        TextField("Amount", text: $amount)
            .keyboardType(.decimalPad)
    }

    Section(header: Text("From")) {
        Picker("From", selection: $viewModel.selectedFromCurrency) {
            ForEach(viewModel.currencies) { currency in
                Text(currency.name).tag(currency)
            }
        }
    }

    Section(header: Text("To")) {
        Picker("To", selection: $viewModel.selectedToCurrency) {
            ForEach(viewModel.currencies) { currency in
                Text(currency.name).tag(currency)
            }
        }
    }

    Section(header: Text("Converted Amount")) {
        Text("\(viewModel.convertedAmount(amount), specifier: "%.2f")
            \(viewModel.selectedToCurrency.symbol)")
            .font(.largeTitle)
            .foregroundColor(.green)
    }
}
```

List and Navigation

1. NavigationView - Container for hierarchical content
2. Stack based navigation
3. Platform-adaptive behaviour
4. Automatic back button
5. NavigationLink - Navigation trigger
6. List - Scrolling container
7. ForEach - Dynamic content

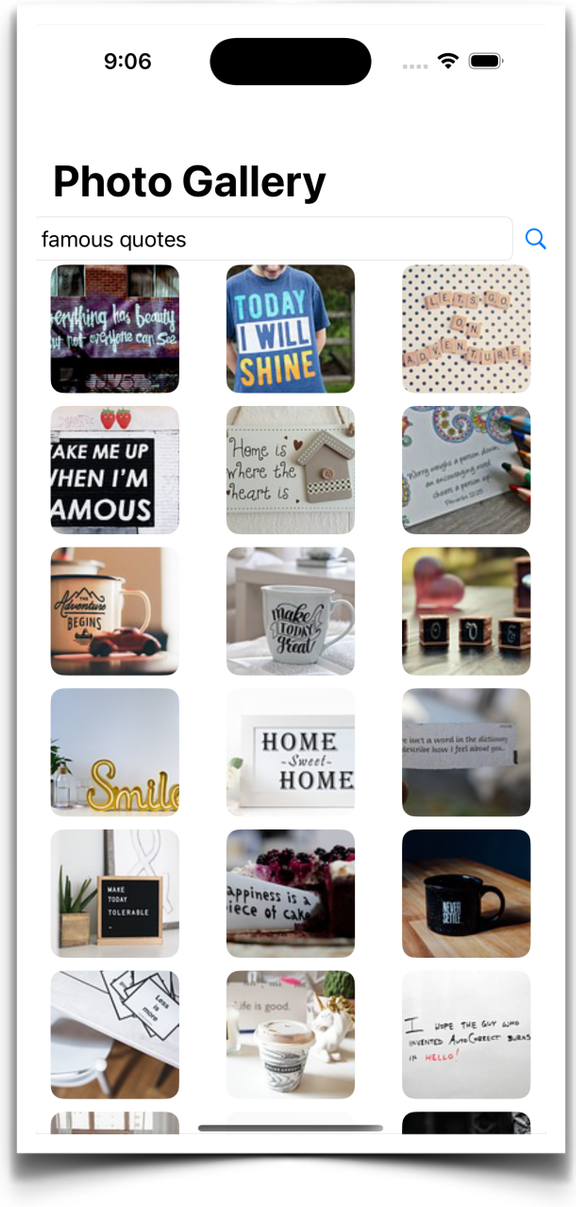


List and Navigation

```
NavigationView {
    List(transactions) { transaction in
        NavigationLink(destination:TransactionDetailView(transaction: transaction)) {
            TransactionRowView(transaction: transaction)
        }
    }
    .navigationTitle("Transactions")
    .navigationBarItems(trailing: Button(action: {
        isShowingAddTransaction = true
    }) {
        Image(systemName: "plus")
    })
}
```

Grid and Image Layouts

1. LazyVGrid - Efficient grid layout
2. GridItem - Grid configuration
3. ScrollView - Scrollable container
4. Image modifiers



Grid and Image Layouts

```
ScrollView {
    LazyVGrid(columns: columns, spacing: 10) {
        ForEach(photos) { photo in
            NavigationLink(destination: PhotoDetailView(photo: photo)) {
                WebImage(url:URL(string: photo.previewURL))
                                                    .resizable()
                                                    .indicator(.activity)
                                                    .scaledToFill()
                                                    .frame(width: 100, height: 100)
                                                    .clipShape(RoundedRectangle(cornerRadius:
                                                        10))
            }
        }
    }
}
.navigationTitle("Photo Gallery")
```

Gestures and Interactions

Available Gestures

1. MagnificationGesture - Pinch to zoom
2. DragGesture - Dragging
3. TapGesture - Tapping
4. RotationGesture - Rotating

Gesture Handlers

1. onChanged - During gesture
2. onEnded - Gesture completion
3. updating - State updates
4. simultaneously - Combined gestures

```
Image(imageName)
    .resizable()
    .scaleEffect(scale)
    .gesture(
        MagnificationGesture()
            .onChanged { value in
                scale = value
            }
    )
    .onTapGesture(count: 2) {
        scale = scale > 1.0 ? 1.0 : 2.0
    }
```

State Management

Understanding the tools for managing app state

What is State?

1. 'State is any data that can change over time'
2. Examples:
    - User input
    - API Responses
    - Toggle Switches
    - Animation Progress

Why it matters?

1. Consistency between UI and data
2. Performance optimization
3. Code organization and maintainability
4. Predictable app behaviour

SwiftUI vs UIKit

1. @State
2. @Binding
3. @ObservableObject
4. @StateObject
5. @EnvironmentObject
6. @Environment

1. Properties
2. Delegation
3. Key-Value Observing (KVO)
4. NotificationCenter

@State

1. Single source of truth
2. Value type storage
3. Triggers view updates
4. Local to a view

```
struct CounterView: View {
    @State private var count = 0

    var body: some View {
        Button("Count: \(count)") {
            count += 1
        }
    }
}
```

@Binding

1. Two-way connection
2. Reference to state
3. Child can modify parent
4. Shared state

```
struct ToggleButton: View {
    @Binding var isOn: Bool

    var body: some View {
        Button(isOn ? "On" : "Off") {
            isOn.toggle()
        }
    }
}
```

@ObservableObject & @Published

1. Reference type storage
2. Complex state management
3. Shared across views
4. External data sources

Class based State Management

@ObservableObject & @Published

```swift
class AppSettings: ObservableObject {
    @Published var isDarkMode = false
    @Published var fontSize = 14
}

// Root View - CREATES the instance
struct RootView: View {
    @StateObject private var settings = AppSettings()

    var body: some View {
        ContentView()
            .environmentObject(settings)
    }
}

// Child Views - RECEIVE via environment
struct SettingsView: View {
    @EnvironmentObject var settings: AppSettings

    var body: some View {
        Toggle("Dark Mode", isOn: $settings.isDarkMode)
    }
}
```

Creation vs Injection

```swift
// CORRECT: Creating new instance
struct RootView: View {
    @StateObject private var settings = AppSettings()

    var body: some View {
        ContentView()
            .environmentObject(settings)
    }
}

// CORRECT: Receiving as parameter
struct SubView: View {
    @ObservedObject var settings: AppSettings

    var body: some View {
        Toggle("Dark Mode", isOn: $settings.isDarkMode)
    }
}

// CORRECT: Receiving from environment
struct ChildView: View {
    @EnvironmentObject var settings: AppSettings

    var body: some View {
        Toggle("Dark Mode", isOn: $settings.isDarkMode)
    }
}
```

When to Use What

1. @State - Simple, local view state
2. @Binding - Child parent communication
3. @StateObject - When you create the instance, at the highest level needed, for lifecycle management
4. @ObservedObject - When receiving as direct parameter, for explicit dependencies
5. @EnvironmentObject - Global App State, for deep dependency injection, when accessing shared state
6. @Environment - System Values

MVVM in SwiftUI

Understanding Through a Currency Converter App

What is MVVM?

1. Model - Data and business logic
2. View - UI Elements and layout
3. ViewModel - Bridge between Model and View

Why MVVM?

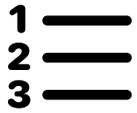
1. Separation of Concerns
2. Testable Code
3. Reusable Components
4. SwiftUI’s natural pattern

The Model Layer

1. Pure data structure - holds only data and its properties
2. No UI logic - doesn't know anything about how it will be displayed
3. Implements Identifiable for SwiftUI lists - enables ForEach usage
4. Implements Hashable for selection in Picker - allows value selection
5. Contains simple computed properties if needed
6. Focuses on data integrity and structure

```swift
struct Currency: Identifiable, Hashable {
    var id: String { symbol }  // Identifiable
    let name: String           // Data property
    let symbol: String         // Data property
    let conversionRate: Double // Data property

    // No UI logic here!
}
```

The ViewModel Layer



1. Acts as a bridge between Model and View
2. Contains all business logic
3. Manages state with @Published properties
4. Handles data transformations
5. Provides data formatting for the View
6. Contains no UI elements or UI logic

The ViewModel Layer

```
class CurrencyViewModel: ObservableObject {
    @Published var currencies: [Currency]
    @Published var selectedFromCurrency: Currency
    @Published var selectedToCurrency: Currency

    init() {
        currencies = [
            Currency(name: "US Dollar", symbol: "USD", conversionRate: 1.0),
            Currency(name: "Euro", symbol: "EUR", conversionRate: 0.84),
            Currency(name: "British Pound", symbol: "GBP", conversionRate: 0.72),
            Currency(name: "Japanese Yen", symbol: "JPY", conversionRate: 109.51)
        ]
        selectedFromCurrency =  Currency(name: "US Dollar", symbol: "USD", conversionRate: 1.0)
        selectedToCurrency = Currency(name: "Euro", symbol: "EUR", conversionRate: 0.84)
    }

    func convertedAmount(_ amount: String) -> Double {
        guard let amountDouble = Double(amount) else { return 0 }
        let inUSD = amountDouble / selectedFromCurrency.conversionRate
        return inUSD * selectedToCurrency.conversionRate
    }

    func swapCurrencies() {
        let temp = selectedFromCurrency
        selectedFromCurrency = selectedToCurrency
        selectedToCurrency = temp
    }
}
```

View Layer

1. Contains only UI elements and layout
2. Uses @ObservedObject for ViewModel updates
3. Delegates all business logic to ViewModel
4. Handles user interaction
5. Manages UI state with @State
6. Focuses on how things look and feel

```swift
struct ContentView: View {
    @ObservedObject var viewModel: CurrencyViewModel
    @State private var amount: String = ""

    var body: some View {
        Form {
            TextField("Amount", text: $amount)
            // UI elements here
        }
    }
}
```

Data Flow in MVVM

View -> ViewModel

1. User enters amount
2. User selects currencies
3. User taps swap button

ViewModel -> Model

1. Creates Currency objects
2. Manages currency data

ViewModel -> View

1. Published properties trigger UI updates
2. Conversion calculations update display

Best Practices

1. Keep Views Dumb - Views should only handle UI and delegate logic to ViewModel
2. Single Responsibility - Each component has a specific role
3. View Model Independence - ViewModels shouldn’t know about Views
4. State Management - Use proper property wrappers