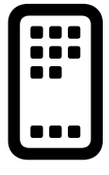
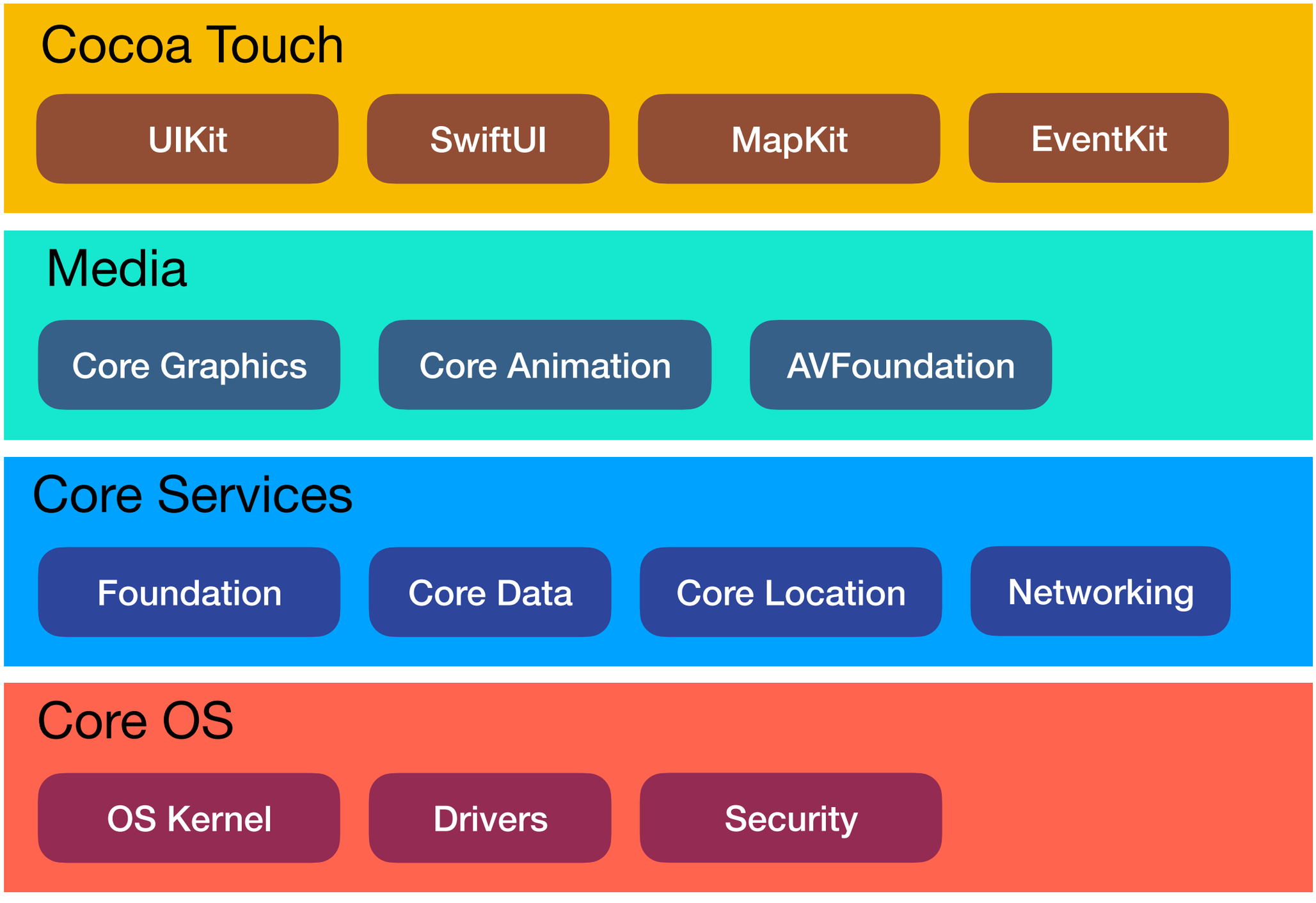


iOS Development Fundamentals

Understanding UIKit, Architecture & Navigation

iOS Architecture Layers



Cocoa Touch

UIKit	The fundamental framework for building iOS apps, providing essential UI components, event handling, and app architecture support.
SwiftUI	A modern, declarative framework for building user interfaces across Apple platforms using Swift's powerful features.
MapKit	Framework for embedding maps and location services directly into your app's interface.
EventKit	Handles calendar and reminder events, allowing apps to access and modify user's calendar data.

Media Layer

CoreGraphics	Low-level drawing engine for handling 2D rendering, images, and PDF content.
Core Animation	Powers smooth animations and visual effects in your app's interface.
AVFoundation	Handles audio and video playback, recording, and media file management.

Core Services

Foundation	Provides fundamental data types, collections, and operating-system services for all apps.
Core Data	Framework for managing the model layer objects in your application, handling data persistence.
Core Location	Provides location and heading information to apps, including GPS and compass data.
Networking	Handles network operations, including HTTP requests, downloading, and data transfer.

Core OS

OS Kernel	The core of the operating system, managing hardware resources and providing low-level services.
Drivers	Software components that enable communication between hardware and the operating system.
Security	Ensures app and data security through encryption, certificates, and access controls.

iOS Developers vs Other Programmers

Other Programmers

iOS Developers

When you realise Swift lets you do in one line what takes other languages 10 lines... 😎

Project Structure

AppDelegate.swift	Entry point of your app that manages app lifecycle events and configuration. Handles app launch, background states, and system notifications.
SceneDelegate.swift	Manages the app's window and root view controller, particularly important for iPad multitasking support. Handles scene lifecycle events.
ViewController.swift	Contains the logic for managing views and user interactions in your app's interface. Each screen typically has its own view controller.
Main.storyboard	Contains all your app's images, icons, and other media resources. Organized catalog for managing different image sizes and device variations.
Assets.xcassets	Contains all your app's images, icons, and other media resources. Organized catalog for managing different image sizes and device variations.
Info.plist	Configuration file containing essential app settings, permissions, and capabilities. Defines how your app interacts with iOS and what features it can access.

Project Structure

```
import UIKit

@main
class AppDelegate: UIResponder, UIApplicationDelegate {
    func application(_ application: UIApplication, didFinishLaunchingWithOptions
                     launchOptions: [UIApplication.LaunchOptionsKey: Any]?) -> Bool {
        // Override point for customization after application launch.
        return true
    }

    // MARK: UISceneSession Lifecycle

    func application(_ application: UIApplication, configurationForConnecting
                     connectingSceneSession: UISceneSession,
                     options: UIScene.ConnectionOptions) -> UISceneConfiguration {
        // Called when a new scene session is being created.
        // Use this method to select a configuration to create the new scene with.
        return UISceneConfiguration(name: "Default Configuration",
                                    sessionRole: connectingSceneSession.role)
    }
}
```

What is UIKit?

View Management	The system for creating, organizing, and managing the visual components of your app's interface through a hierarchy of views and view controllers.
Event Handling	The process of detecting, processing, and responding to user interactions like touches, gestures, and control events through the responder chain.
User Interface Controls	Pre-built interactive components like buttons, text fields, and switches that provide standardized ways for users to input data and trigger actions.
Navigation	The system for managing transitions between different screens and content using patterns like push/pop, modal presentations, and tab-based navigation.
Gesture Recognizers	Objects that detect specific touch patterns (taps, swipes, pinches) and convert them into meaningful actions in your app.

Foundational Concepts

View Controllers - Lifecycle Methods

1. loadView()
2. viewDidLoad()
3. viewWillAppear(_:)
4. viewDidAppear(_:)
5. viewWillDisappear(_:)
6. viewDidDisappear(_:)

The Conductors of Your App Orchestra

1. loadView() First Called

Creates the view that the controller manages.

When to use:

- Creating views programmatically instead of using storyboards
- Custom view initialization
- Rarely overridden if using Interface Builder

The Conductors of Your App Orchestra

2. viewDidLoad() Setup Phase

Called after the view is loaded into memory. Perfect for one-time setup.

Common uses:

- Initial UI setup and configuration
- Data initialization
- Setting up delegates and data sources
- Network requests setup

The Conductors of Your App Orchestra

3. viewWillAppear(_:) Pre-Display

Called just before the view becomes visible to the user.

Perfect for:

- Updating data that might have changed
- Starting animations
- Refreshing UI elements
- Showing/hiding elements based on state

The Conductors of Your App Orchestra

4. viewDidAppear(_:) Post-Display

Called right after the view is displayed on screen.

Ideal for:

- Starting time-consuming tasks
- Playing animations or videos
- Analytics tracking
- Network requests that update UI

The Conductors of Your App Orchestra

5. viewWillDisappear(_:) Pre-Remove

Called just before the view is removed from the screen.

Use for:

- Saving user changes
- Stopping animations
- Resigning first responder
- Cleanup of temporary UI states

The Conductors of Your App Orchestra

6. viewDidDisappear(_:) Cleanup

Called after the view is removed from the screen.

Best for:

- Cleanup of resources
- Stopping background tasks
- Removing notifications
- Releasing heavy resources

The Conductors of Your App Orchestra

Key Points to remember

1. Methods are called in sequence as view loads/ unloads.
2. Always call super when overriding these methods
3. Keep these methods lightweight to ensure smooth transitions
4. Use appropriate method for timing-specific code

Outlets in Xcode

1. Reference UI elements
2. @IBOutlet weak var label: UILabel!
3. Access properties and methods
4. Update UI programmatically

Connecting UI to Code

Actions in Xcode

1. Handle user interactions
2. @IBAction func buttonTapped(_ sender: UIButton)
3. Connect multiple controls
4. Respond to events

Respond to User Interactions

Navigation Types

1. Push/Pop
2. Modal Presentation
3. Tab Bar Navigation
4. Custom Transitions

Navigation Segues

1. Show (Push)
2. Present Modally
3. Show Detail
4. Popover

Segue Implementation

1. Storyboard segues
2. Programmatic navigation
3. Custom transitions
4. Unwind segues

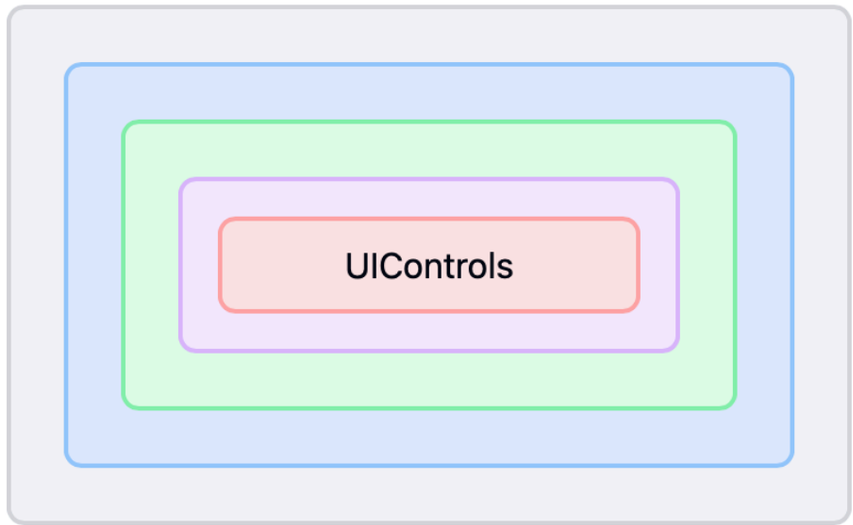
Passing Data Between ViewControllers

1. Direct Property
2. Prepare for Segue
3. Delegation
4. Closures

Communication Patterns

View Hierarchy - View Stack

1. UIWindow
2. Root View Controller
3. Container Views
4. Custom Views
5. UIControls



↑
View Stack Order

Building Your Interface Layer by Layer

View Hierarchy - Key Concepts

1. Superview-Subview
2. View Bounds
3. Frame vs Bounds
4. Z-order

Building Your Interface Layer by Layer

View Hierarchy - Common Tasks

1. Adding/Removing Views
2. View Layout
3. View Transform
4. Hit Testing

Building Your Interface Layer by Layer

UIView Fundamentals

1. Properties
2. frame: CGRect
3. bounds: CGRect
4. center: CGPoint
5. transform: CGAffineTransform

UIView - Visual Properties

1. backgroundColor: UIColor
2. alpha: CGFloat
3. isHidden: Bool
4. contentMode: UIView.ContentMode

Frame vs Bounds - Frames

1. Rectangle in superview's coordinate system
2. Position relative to parent view
3. Used for view positioning
4. Includes any transforms

```
view.frame = CGRect(x: 50, y: 50,
                    width: 200, height: 100)
```

Understanding View Coordinate Systems

Frame vs Bounds - Bounds

1. Rectangle in own coordinate system
2. Internal drawing space
3. Used for content layout
4. Independent of transforms

```
view.bounds = CGRect(x: 0, y: 0,
                     width: 200, height: 100)
```

Understanding View Coordinate Systems

Common UI Controls - Basic

1. UIButtons
2. UILabel
3. UITextField
4. UISwitch
5. UIImageView

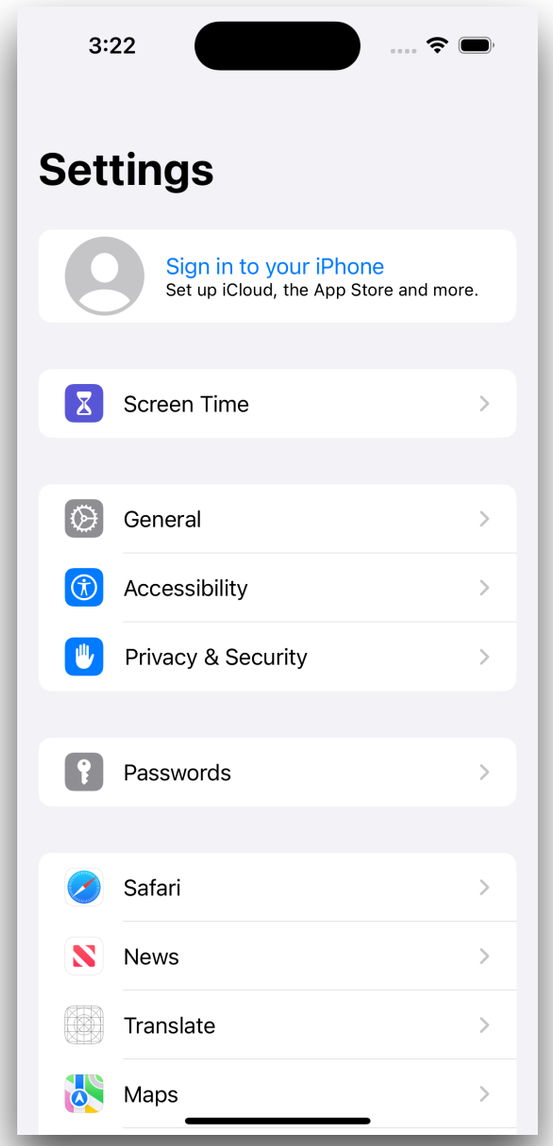
Image Credit - Apple

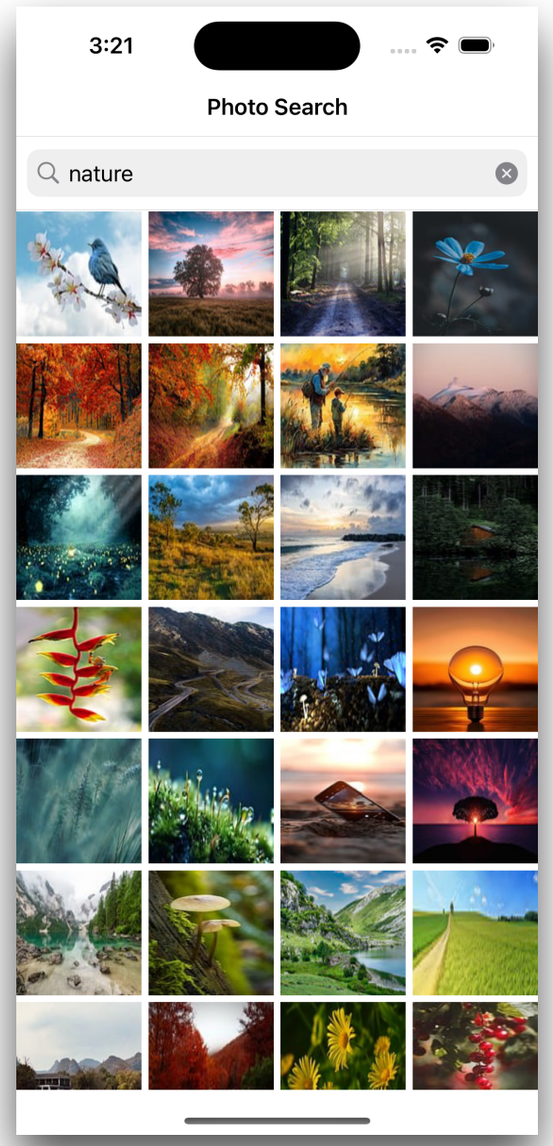
Foundational Concepts

Common UI Controls - Container Views

1. UIScrollView
2. UITableView
3. UICollectionView
4. UIStackView
5. UIPageViewController

UITableView & UICollectionView





Common UI Controls - Special Controls

1. UIProgressView
2. UIActivityIndicatorView
3. UIPickerView
4. UISegmentedControl

Gesture Recognizers

UITapGestureRecognizer	Detects a tap or multiple taps on a view.
UIPanGestureRecognizer	Detects panning (dragging a finger across the screen).
UISwipeGestureRecognizer	Detects a swipe in a particular direction.
UIPinchGestureRecognizer	Detects pinch gestures, usually used for zooming in and out.
UIRotationGestureRecognizer	Detects rotation gestures, commonly used for rotating views.
UILongPressGestureRecognizer	Detects a long press gesture, where the user holds down their
UIScreenEdgePanGestureRecognizer	Detects a swipe that starts from the edge of the screen.

Common UI Controls - Control Patterns

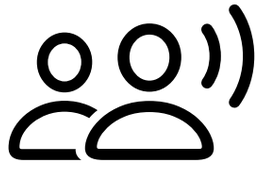
1. Target Action

```
button.addTarget(self, action: #selector(handleTap),
                 for: .touchUpInside)
```

2. Delegate Pattern

```
textField.delegate = self
```

Essential Interface Elements



Delegate Pattern

Fundamental Design Pattern

Delegate Pattern

1. Fundamental Design Pattern
2. Allows object communication
3. Promotes loose coupling
4. Used in UIKit and Foundation

Key Concepts

1. Delegation - Object hands off responsibility
2. Protocols - Defines delegate methods
3. Weak References - Prevents retain cycles
4. Optional/Required Methods - Flexibility in implementation

Common Use Cases

1. UITableViewDelegate - Handling table view interactions
2. UITextFieldDelegate - Managing text input and editing
3. URLSessionDelegate - Handling network request events
4. CLLocationManagerDelegate - Location Updates
5. WKNavigationDelegate - Managing Navigation Requests

Delegates vs Closures vs Notifications vs KVO

Delegates	Closures	Notifications	KVO
One-to-one Communication	Simple inline & single use callbacks	One-to-many communication	One-to-many property observation
Strong type checking	Capture context easily	Broadcast events widely	Automatic updates on property changes
Multiple Callback methods	More flexible and concise	Useful for system-wide events	Decouples objects from property changes
Used for complex interfaces like UITableViewDelegate	Example: Completion Handlers	Example: Keyboard show/hide	Example: Observing model changes

UITableView

Core Concepts and Setup

UITableView Fundamentals

1. A scrollable list of organised data
2. Displays content in single-column rows
3. Supports both plain and grouped styles
4. Perfect for settings, lists and forms

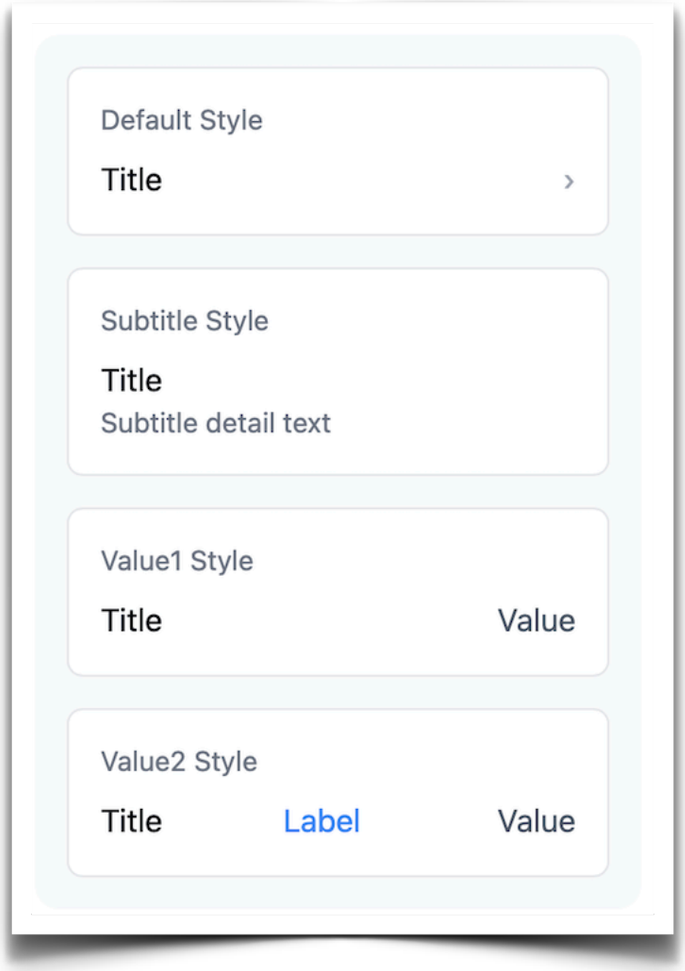
Key Components

1. UITableViewDataSource - Provides data
2. UITableViewDelegate - Handles Interaction
3. UITableViewCell - Base cell for content
4. UITableViewHeaderFooterView - Section headers/ footers

```swift
// Initialize TableView
let tableView = UITableView(frame: .zero, style: .insetGrouped)

// Register cell
tableView.register(UITableViewCell.self,
                   forCellReuseIdentifier: "Cell")

// Set delegates
tableView.dataSource = self
tableView.delegate = self

// Add to view
view.addSubview(tableView)
tableView.translatesAutoresizingMaskIntoConstraints = false
```

Cell Styles & Customisation

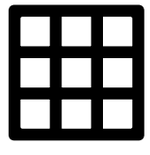
1. Default - Basic title and optional subtitle.
2. Subtitle - Title with detail text below
3. Value1 - Title and value aligned right
4. Value2 - Title, coloured label, cell layout
5. Custom - Fully customised cell layout



Implementation Example

1. Register Cell Identifiers
2. Implement Required Protocols
3. Configure cell appearance
4. Handle rows selection
5. Manage data updates

```swift
func tableView(_ tableView: UITableView,
               didSelectRowAt indexPath: IndexPath) {
    tableView.deselectRow(at: indexPath, animated: true)
    let item = rows[indexPath.row]
    // Handle selection
}
```



UICollectionView

Building Grid and Custom Layouts

UICollectionView Basics

1. A flexible way to present items in a grid or custom layout.
2. Similar to UITableView but with more layout options
3. Perfect for Photo galleries, menus, or card - based interfaces.

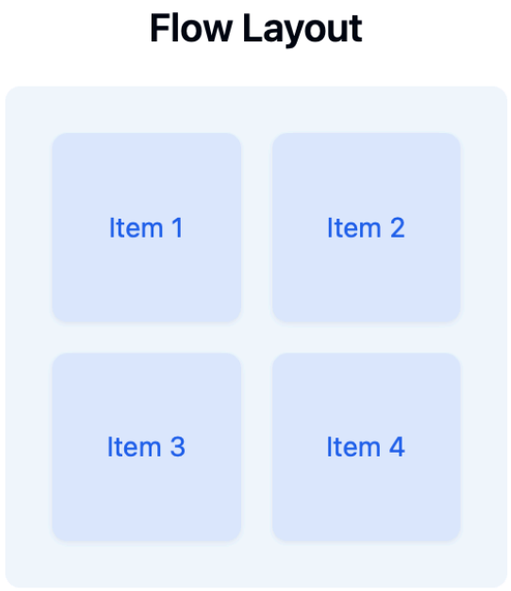
Key Components

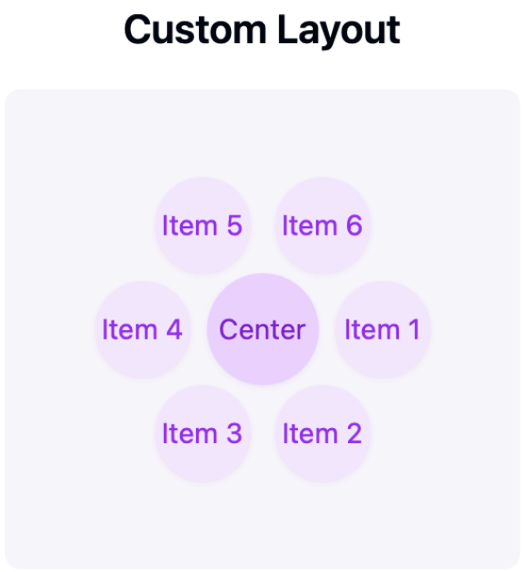
1. UICollectionViewCell - Custom Cell for content.
2. UICollectionViewLayout - Manages item arrangement.
3. UICollectionViewDataSource - Provides data
4. UICollectionViewDelegate - Handles Interaction

```swift
// Initialize with Flow Layout
let layout = UICollectionViewFlowLayout()
layout.itemSize = CGSize(width: 100, height: 100)
layout.minimumInteritemSpacing = 10

// Create Collection View
let collectionView = UICollectionView(
    frame: .zero,
    collectionViewLayout: layout
)
```

Layout Options

1. Flow Layout - Grid based arrangement
2. Custom Layout - Complete layout control
3. Compositional Layout - Complex, nested layouts





Data Source & Delegate Implementation

1. Register cell classes
2. Configure layout
3. Implement data source methods
4. Handle selection and updates

```swift
// Required Methods
func collectionView(_ collectionView: UICollectionView,
                    numberOfItemsInSection section: Int) -> Int {
    return rows.count
}

func collectionView(_ collectionView: UICollectionView,
    cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {
    let cell = collectionView.dequeueReusableCell(
        withReuseIdentifier: "Cell",
        for: indexPath
    )
    return cell
}
```

```swift
// Selection Handling
func collectionView(
    _ collectionView: UICollectionView, didSelectItemAt indexPath: IndexPath
) {
    let item = rows[indexPath.item]
    handleSelection(item)
}
```

Auto Layout

Constraint based layout system

Auto Layout

1. A constraint-based layout system for iOS
2. Allows dynamic adaptation to different screen sizes and orientations
3. Based on relationships between UI elements
4. Key to creating responsive and adaptive interfaces

Key Concepts

1. Constraints: Rules defining layout relationships
2. Intrinsic Content Size: Natural size of UI elements
3. Content Hugging Priority: Resistance to growing larger
4. Compression Resistance Priority: Resistance to shrinking
5. Layout Guides: Abstract layout areas (e.g., safe area)

Layout Guides

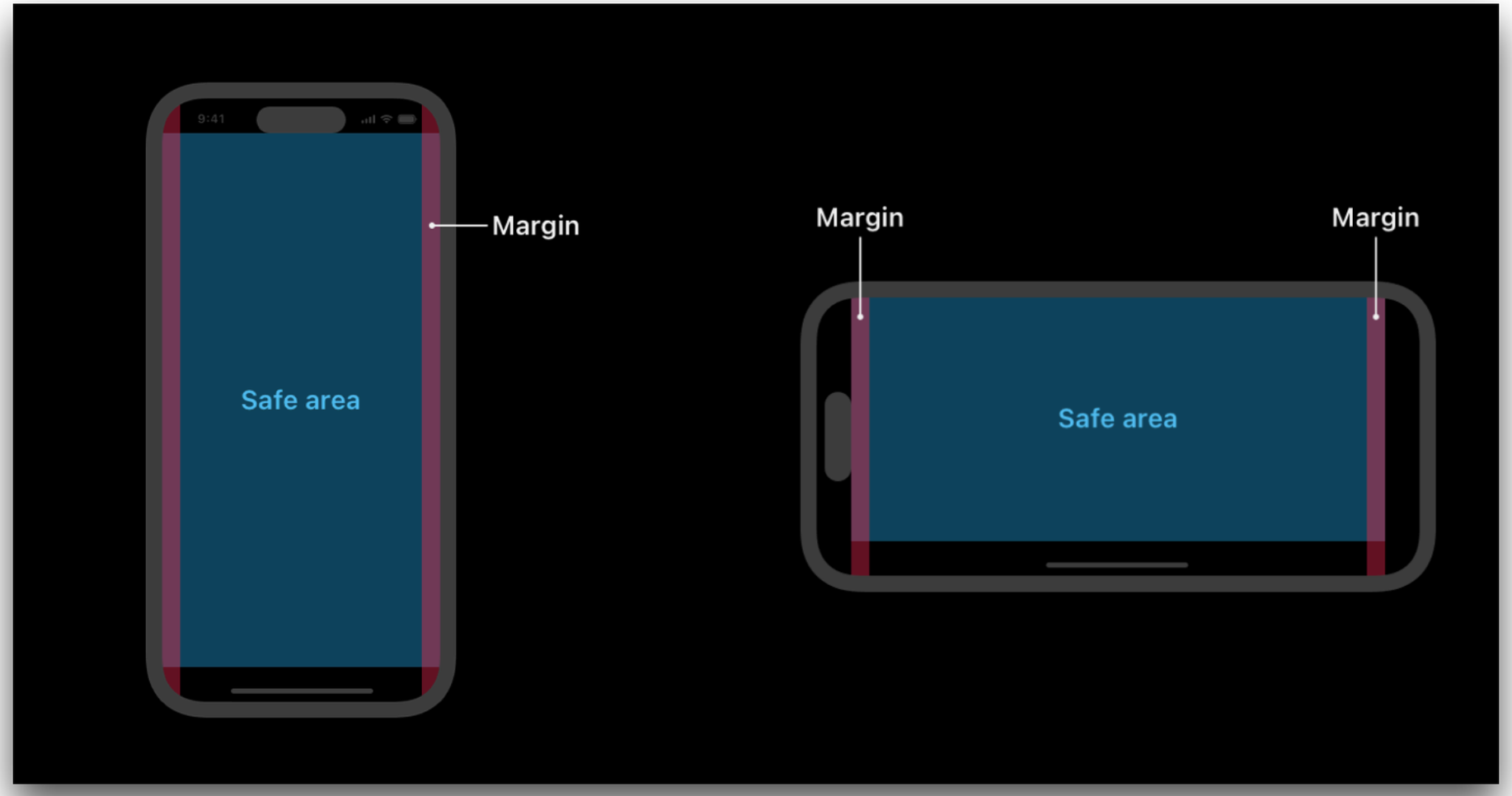


Image Credit: Apple

Creating Constraints

1. Interface Builder: Visual editor in Xcode
2. NSLayoutConstraint API: Programmatic constraint creation
3. Visual Format Language (VFL): String-based constraint definition
4. Layout Anchors: Convenient API for common constraints

Common Patterns

1. Centering: centerX and centerY constraints
2. Aspect ratio: Maintain width-to-height ratio
3. Dynamic text: Allow labels to expand/shrink
4. Scrolling content: Use scroll view with content size constraints
5. Equal spacing: Use distribution in stack views

Auto Layout with Stack Views

1. Simplified Auto Layout for common patterns
2. Manages layout of a linear series of views
3. Properties: axis, distribution, alignment, spacing
4. Reduces number of explicit constraints needed
5. Ideal for forms, simple lists, and button arrays

Debugging Auto Layout

1. Use Xcode's Debug View Hierarchy
2. Check console for Auto Layout error messages
3. Use visual debugging techniques (e.g., colored backgrounds)
4. Verify intrinsic content sizes
5. Use 'Debug Auto Layout' button in Interface Builder

Best Practices

1. Use stack views for simple layouts
2. Avoid conflicting constraints
3. Set priorities to resolve conflicts when necessary
4. Use intrinsic content size when possible
5. Test layouts on various device sizes and orientations