iOS/SwiftUI Development Guide

This guide explains how to use the AI Orchestrator for iOS/SwiftUI development with Cursor IDE.

Overview

The AI Orchestrator now supports iOS/SwiftUI development, allowing you to:
- Build iOS projects for the iOS Simulator
- Run apps in the Simulator
- Execute XCTest unit and UI tests
- Auto-detect and fix Swift compilation errors
- Manage iOS Simulators

Requirements

- macOS with Xcode installed (version 15+ recommended)
- Xcode Command Line Tools (`xcode-select --install`)
- iOS Simulator runtimes installed

Quick Start

1. List Available Simulators

Use the MCP tool to list available iOS Simulators:

```
list_ios_simulators
```

This will show all available devices with their UDIDs, states, and iOS versions.

2. Boot a Simulator

Before running an app, boot a simulator:

```
boot_ios_simulator(device_name="iPhone 15")
```

3. Build an iOS Project

Build your SwiftUI project:

```
build_ios_project(
    project_path="/path/to/MyApp",
    scheme="MyApp",  # Optional, auto-detected
    configuration="Debug"
)
```

4. Run in Simulator

Build and run in the Simulator:

```
run_ios_app(
    project_path="/path/to/MyApp",
    simulator_device="iPhone 15"
)
```

5. Run Tests

Execute XCTest tests:

```
test_ios_project(
    project_path="/path/to/MyApp",
    scheme="MyApp"
)
```

Project Detection

The orchestrator automatically detects iOS projects by looking for:
- `.xcodeproj` files
- `.xcworkspace` files
- `Package.swift` (Swift Package Manager)
- Swift files containing `import SwiftUI` or `import UIKit`

MCP Tools Reference

`list_ios_simulators`

List all available iOS Simulators.

Parameters:
- `available_only` (bool, default: true): Only show available simulators

`boot_ios_simulator`

Boot an iOS Simulator.

Parameters:
- `device_name` (string, default: “iPhone 15”): Device name to boot

`build_ios_project`

Build an iOS project for the Simulator.

Parameters:
- `project_path` (string, required): Path to the project
- `scheme` (string, optional): Xcode scheme to build
- `configuration` (string, default: “Debug”): Build configuration
- `simulator_device` (string, default: “iPhone 15”): Target simulator

`run_ios_app`

Build and run an app in the Simulator.

Parameters:
- `project_path` (string, required): Path to the project
- `scheme` (string, optional): Xcode scheme

- `simulator_device` (string, default: “iPhone 15”): Target simulator
- `boot_simulator` (bool, default: true): Auto-boot if needed

`test_ios_project`

Run XCTest tests.

Parameters:
- `project_path` (string, required): Path to the project
- `scheme` (string, optional): Test scheme
- `simulator_device` (string, default: “iPhone 15”): Target simulator

`take_simulator_screenshot`

Capture a screenshot from the Simulator.

Parameters:
- `output_path` (string, default: “./screenshot.png”): Output path

`reset_ios_simulator`

Reset a Simulator to factory state.

Parameters:
- `device_name` (string, optional): Device to reset (or booted one)

Configuration

Set these environment variables to customize iOS behavior:

```
# Simulator settings
IOS_SIMULATOR_DEVICE=iPhone 15
IOS_SIMULATOR_OS=iOS 17.0
IOS_AUTO_BOOT_SIMULATOR=true

# Build settings
XCODE_BUILD_TIMEOUT=600
XCODE_TEST_TIMEOUT=600
XCODE_CONFIGURATION=Debug

# Code signing
IOS_SKIP_CODE_SIGNING=true

# Derived data
IOS_USE_PROJECT_DERIVED_DATA=true
IOS_DERIVED_DATA_PATH=/custom/path

# Build behavior
IOS_CLEAN_BEFORE_BUILD=false
IOS_PARALLEL_BUILDS=true
```

Error Detection

The orchestrator detects these iOS-specific error categories:

Swift Compilation Errors

- Syntax errors

- Type mismatches
- Missing imports
- Unresolved identifiers

Code Signing Errors

- Missing certificates
- Invalid provisioning profiles
- Team ID issues

Simulator Errors

- Boot failures
- Device not available
- Runtime issues

Build Errors

- Missing schemes
- Linker errors
- Compiler crashes

SwiftUI Preview Errors

- Preview crashes
- Timeout errors
- Missing preview providers

Auto-Fix Strategies

SwiftSyntaxFixer

Generates AI prompts for Swift syntax errors.

SimulatorFixer

- Boot simulators
- Reset simulators
- Find available devices

SwiftPackageFixer

- Resolve SPM dependencies
- Update packages
- Clean build artifacts

SigningFixer

- Generate signing-disabled build commands
- Provide signing fix suggestions

Workflow Example

Here’s a typical iOS development workflow with Cursor:

```
1. User: "Create a new SwiftUI counter app"

2. Cursor generates the code using Claude

3. User: "Build and run it"

4. Orchestrator:
   - Detects iOS project
   - Boots iPhone 15 Simulator
   - Builds with xcodebuild
   - Installs app on Simulator
   - Launches app

5. User: "Add unit tests and run them"

6. Cursor generates tests
   Orchestrator runs XCTest

7. If errors occur:
   - ErrorDetector categorizes errors
   - SwiftSyntaxFixer generates fix prompts
   - AutoFixer applies corrections
   - VerificationLoop retries build
```

Troubleshooting

“xcodebuild not found”

Install Xcode Command Line Tools:

```
xcode-select --install
```

“Simulator not available”

1. Open Xcode > Preferences > Components
2. Download the required iOS Simulator runtime

“Code signing error”

For simulator builds, signing is automatically disabled. For device builds:
1. Open the project in Xcode
2. Go to Signing & Capabilities
3. Enable “Automatically manage signing”
4. Select your development team

“Build timeout”

Increase the timeout:

```
export XCODE_BUILD_TIMEOUT=900
```

“No scheme found”

Specify the scheme explicitly:

```
build_ios_project(
    project_path="/path/to/project",
    scheme="MyApp"
)
```

Programmatic Usage

```python
from ai_orchestrator.execution import (
    iOSSimulatorManager,
    iOSProjectBuilder,
    list_simulators,
    boot_simulator,
    build_ios_project,
    run_ios_tests,
)

# List simulators
sims = list_simulators()
for sim in sims:
    print(f"{sim.name} ({sim.os_version}) - {sim.state.value}")

# Boot a simulator
result = boot_simulator("iPhone 15 Pro")
print(f"Boot: {result.success} - {result.message}")

# Build a project
build_result = build_ios_project("/path/to/MyApp", scheme="MyApp")
print(f"Build: {build_result.success}")
print(f"App path: {build_result.app_path}")

# Run tests
test_result = run_ios_tests("/path/to/MyApp")
print(f"Tests: {'Passed' if test_result.success else 'Failed'}")
```

Integration with Verification Loop

The verification loop fully supports iOS projects:

```
verify_project(
    project_path="/path/to/MyApp",
    max_cycles=10,
    run_tests=true,
    auto_fix=true
)
```

This will:
1. Build the iOS project
2. Run XCTest tests
3. Detect compilation/test errors
4. Generate fixes using AI
5. Apply fixes and rebuild
6. Repeat until success or max cycles