User Interface Testing

UI testing gives you the ability to find and interact with the UI of your app in order to validate the properties and state of the UI elements.

UI testing includes UI recording, which gives you the ability to generate code that exercises your app's UI the same way you do, and which you can expand upon to implement UI tests. This is a great way to quickly get started writing UI tests.

Test reports are enhanced to provide detailed information about UI tests, including snapshots of the state of the UI at test failures.

UI tests rests upon two core technologies: the XCTest framework and Accessibility.

- XCTest provides the framework for UI testing capabilities, integrated with Xcode. Creating and using UI testing expands upon what you know about using XCTest and creating unit tests. You create a UI test target, and you create UI test classes and UI test methods as a part of your project. You use XCTest assertions to validate that expected outcomes are true. You also get continuous integration via Xcode Server and xcodebuild. XCTest is fully compatible with both Objective-C and Swift.
- Accessibility is the core technology that allows disabled users the same rich experience for iOS and macOS that other users receive. It includes a rich set of semantic data about the UI that users can use can use to guide them through using your app. Accessibility is integrated with both UIKit and AppKit and has APIs that allow you to fine-tune behaviors and what is exposed for external use. UI testing uses that data to perform its functions.

Creating UI tests in source code is similar to creating unit tests. You create a UI test target for your app; then Xcode creates a default UI test group and implementation file for you with an example test method template in the implementation file. When you create the UI test target, you specify the app that your tests will address.

UI testing works by finding an app's UI objects with queries, synthesizing events and sending them to those objects, and providing a rich api enabling you to examine the UI objects properties and state to compare them against the expected state.

Requirements

UI testing depends upon services and APIs not only in the development tools but in the OS platforms. You'll need Xcode 7, macOS 10.11, and iOS 9 (or later versions). UI testing protects privacy:

- iOS devices need to be enabled for development and connected to a trusted host.
- macOS needs permissions granted to a special Xcode Helper app. You are prompted for this automatically on your first use of UI tests.

iOS devices need to be enabled for development and connected to a trusted host. macOS needs permission granted to a special Xcode Helper app (prompt automatically on first use).

Concepts And APIs

UI testing differs from unit testing in fundamental ways. Unit testing enables you to work within your app's scope and allows you to exercise functions and methods with full access to your app's variables and state. UI testing exercises your app's UI in the same way that users do without access to your app's internal methods, functions, and variables. This enables your tests to see the app the same way a user does, exposing UI problems that users encounter.

Your test code runs as a separate process, synthesizing events that UI in your app responds to.

APIs

UI tests are based on the implementation of three new classes:

- XCUIApplication
- XCUIElement
- XCUIElementQuery

Get Started with UI Recording

Start with UI recording. It generates source code into a test implementation file that can be edited to construct tests or playback a particular use scenario. UI recording is also useful for exploring new UI or for learning how to write UI test sequences. The basic sequence of operations is:

1. Using the test navigator, create an UI testing target.
2. In the template file that is created, place the cursor into the test function.
3. Start UI recording.

   The app launches and runs. Exercise the app doing a sequence of UI actions. Xcode captures the actions into source in the body of the function.

4. When done with the actions you want to test, stop UI recording.
5. Add XCTest assertions to the source.

Writing UI Tests

API tests can have both functional and performance aspects, and so can UI tests. UI tests operate at the surface space of the app and tend to integrate a lot of low level functionality into what the user sees as presentation and response.

UI tests fundamentally operate on the level of events and responses.

- Query to find an element.
- Know the expected behavior of the element as reference.
- Tap or click the element to elicit a response.
- Measure that the response matches or does not match the expected for a pass/fail result.

Creating UI tests with XCTest is an extension of the same programming model as creating unit tests. Similar operations and programming methodology is used overall, with differences given the basic notions of the UI testing APIs and how they operate that were described in User Interface Testing.

In the test class structure, the provided `setUp` method includes two differences from `setUp` in a unit test class.

```
- (void)setUp {
    [super setUp];


    // Put setup code here. This method is called before the invocation of each test
method in the class.
```

```
    self.continueAfterFailure = NO;
    [[[XCUIApplication alloc] init] launch];
}
```

The value of `self.continueAfterFailure` is set to `NO` as a default. This is usually the correct configuration because each step in a UI test method tends to be dependent upon the success of the preceding one; if one step fails, all the following tests also fail as well.

The other addition to the `setUp` method to include creating an instance of `XCUIApplication` and launching it. UI tests must launch the applications that they test, and since `setUp` runs before each test method, this ensures that the app is launched for each test method.

When writing UI test methods, you should use the UI recording feature to create a basic set of steps for your test. You then edit this basic sequence for your purposes, using the XCTest assertions to provide pass or fail results as with unit tests. UI tests have both functional and performance aspects, just like unit tests.

The general pattern of a UI test for correctness is as follows:

- Use an XCUIElementQuery to find an XCUIElement.
- Synthesize an event and send it to the XCUIElement.
- Use an assertion to compare the state of the XCUIElement against an expected reference state.

To construct a UI test for performance, wrap a repeatable UI sequence of steps into the `measureBlock` structure seen in Writing Performance Tests.