Available online www.jsaer.com

***Journal of Scientific and Engineering Research*, 2019, 6(11):305-310**

Research Article

ISSN: 2394-2630
CODEN(USA): JSERBR

On Device - Utilizing UI Elements to Capture Logs in Test Automation for Apple Applications

Amit Gupta

Staff Engineer
San Jose, CA, USA
Email id: gupta25@gmail.com

Abstract XCUITest is a robust tool for automating UI tests for Apple platform's applications. It offers a comprehensive framework for simulating user interactions and verifying UI behaviours. However, a significant challenge arises from Apple's sandboxing mechanism, which restricts access to the application container and, consequently, the logs generated by the application. This limitation can hinder effective debugging and validation of application behaviour. This paper explores a method to overcome this restriction by using a UI element within the application itself to capture and retrieve logs during testing. By integrating a lightweight logging mechanism directly into the application's UI, we can ensure that log messages are easily accessible to the test code. The proposed approach is simple to implement, highly reliable, and does not depend on external servers or network connections, making it a practical and efficient solution for developers and testers. Additionally, this method maintains the security and integrity of the application's sandbox environment while providing valuable insights into its runtime behaviour.

Keywords XCUI Test, logging framework, Apple Platform, UI Test Automation, UI Automation, Test Automation, iOS Test Automation, XC Test

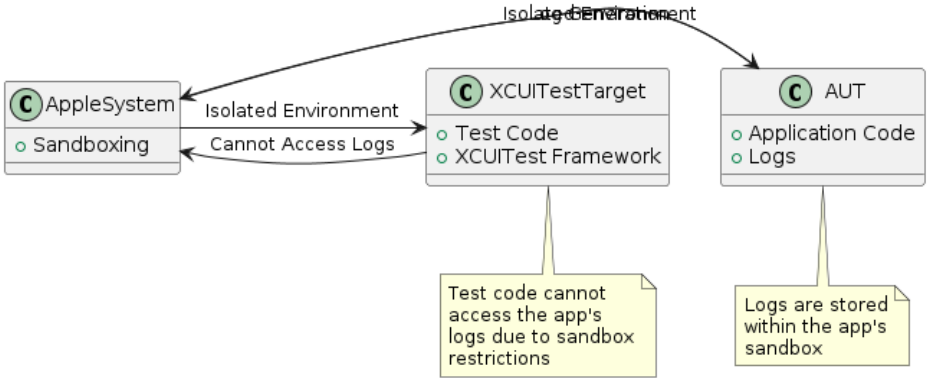

Figure 1: Shows the issue with XCUI Tests can't access the application logs

Introduction

XCUI Test is an excellent tool for automating UI testing of applications on Apple platforms. It provides a robust framework for simulating user interactions and verifying UI behaviours, ensuring that applications function correctly across various scenarios. XCUI Test operates by running UI tests as a separate target, which involves an additional proxy application being installed on the device. This architecture effectively isolates the test code from the application under test (AUT), enhancing the security and stability of the testing process. However, this architecture also presents a significant challenge: the sandboxing mechanism enforced by Apple. This security feature restricts test code from accessing the application container, making it difficult to retrieve logs generated

by the application. These logs are crucial for diagnosing and triaging failures encountered during test execution. Without access to these logs, developers and testers may struggle to understand the root causes of issues, hindering the debugging process. This limitation necessitates innovative approaches to log retrieval that comply with Apple's security constraints while providing the necessary diagnostic information to ensure comprehensive and effective testing.

To address this challenge, two primary approaches can be considered:

A. Using a Log Server: This approach involves writing a lightweight log server or utilizing an existing commercial log server on the cloud to capture and post messages from the application. By implementing a custom log server, developers can tailor the logging solution to meet specific needs, ensuring that only relevant log data is captured and stored. Alternatively, leveraging a commercial log server can provide a robust, scalable solution with minimal setup, often including advanced features like log aggregation, searching, and filtering. Once the application is configured to send log messages to the server, the test code can then retrieve these log messages using REST APIs. This enables real-time access to the logs, facilitating immediate analysis and debugging. The REST API integration allows for seamless communication between the test code and the log server, ensuring that log retrieval is efficient and reliable. This method not only aids in capturing comprehensive log data but also provides a centralized location for log management, making it easier to monitor, analyse, and respond to issues that arise during testing.

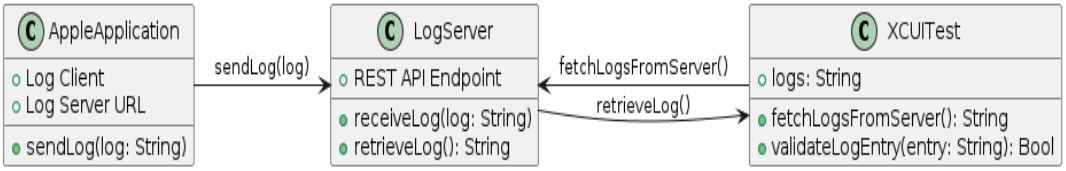

Figure 2: Shows an alternative solution based on Logging Server

B. Using a UI Element: This method entails creating an application-wide hidden UI element within the application that conditionally compiles to display log messages. This UI element, often a small and invisible component like a `UI Text View`, is embedded into the application's main window, ensuring it is available across all screens and contexts of the application. The log messages generated by the application during runtime are appended to this UI element, providing a live feed of diagnostic information. These log messages can then be retrieved and validated using XCUI Tests by accessing the UI element directly. By leveraging the capabilities of XCUI Test to interact with UI components, testers can extract the log data embedded within this hidden element, allowing for detailed analysis and verification of application behaviour. This approach effectively bypasses the sandbox restrictions imposed by iOS, as the test code can interact with the UI elements even if it cannot directly access the application's file system or log files. Furthermore, the conditional compilation of this logging UI element ensures that it can be included or excluded from production builds, maintaining the application's performance and security while still providing a valuable tool for debugging and testing during development and quality assurance processes.

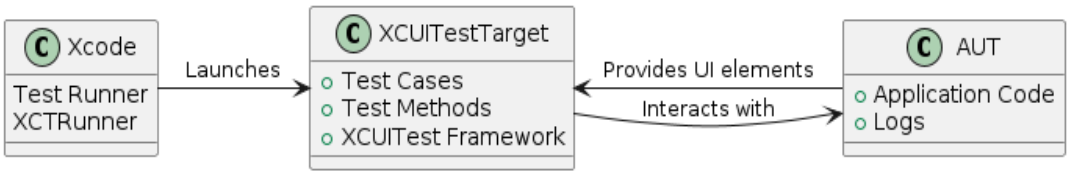

Figure 3: Shows how UI Element based log UI Element works

Methodology

A. Implementation in the Application

[1]. Adding a UI Text View Element

Incorporate a UI Text View element into the application's main window to ensure its availability across all screens. This UI Text View should be designed to be small and unobtrusive, potentially even hidden, so it does not interfere with the user interface or user experience. Placing the UI Text View in the main window guarantees that it remains accessible no matter which view or screen the user navigates to within the application. This element will serve as a container for log messages, capturing and displaying logs generated during the application's runtime. By embedding this UI Text View, developers create a persistent log viewer that can be accessed and queried by the test code, facilitating real-time log monitoring and analysis across the entire application lifecycle.

[2]. Creating a Logging Class

Develop a class, named Automation Logging View, responsible for appending log statements to the UI Text View. This class should encapsulate all the functionality related to logging, providing a clean and organized way to handle log messages. The Automation Logging View class should offer methods to manage and append log entries, ensuring that logs are systematically recorded and accessible. This includes methods to initialize the UI Text View, append new log messages, and potentially clear old logs to prevent the log view from becoming overloaded. Additionally, the class could provide utility functions to format log messages, include timestamps, and categorize logs by severity or type. By centralizing logging functionality in this class, developers can ensure consistent and reliable log management throughout the application, making it easier to track and debug issues during testing and development.

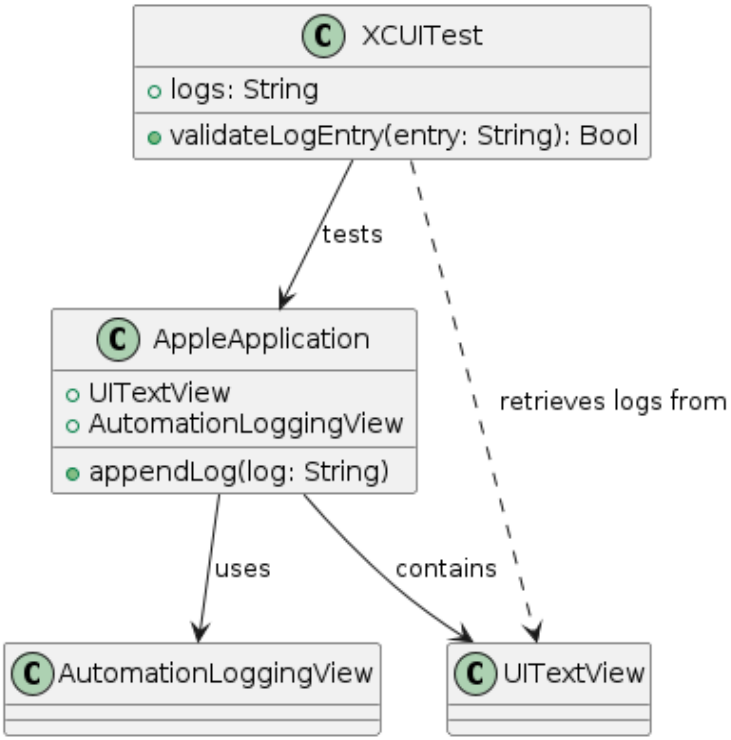

Figure 4: Logging classes for implementation

B. Code example: adding uitextview

```
class AutomationLoggingView: UITextView {
    static let shared = AutomationLoggingView()

    private override init(frame: CGRect, textContainer: NSTextContainer?) {
        super.init(frame: frame, textContainer: textContainer)
        self.isEditable = false
        self.isHidden = true
    }

    required init?(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

    func appendLog(_ log: String) {
        self.text.append("\(log)\n")
    }
}
```

C. Implementation in XCUITests

[1]. Accessing the UI Element

Utilize `XCUITest` to retrieve the `UITextView` element. This involves writing test code that specifically locates the `UITextView` within the application's UI hierarchy. Once the `UITextView` is identified, the test can access its content, which contains the log messages generated by the application. Parsing the logs contained within the element is the next crucial step. The test code must extract the log data from the `UITextView`, which may involve reading the text property of the element and processing it to separate individual log entries. This parsing process should be designed to handle various log formats and ensure that all relevant information is accurately extracted. By systematically accessing and parsing the logs, the test code can gather detailed insights into the application's behavior, facilitating thorough validation and debugging.

[2]. Creating Validation Methods

Develop methods within the test code to verify the content of the logs. These validation methods are essential for ensuring that the logs generated by the application meet the expected criteria and accurately reflect the application's behavior. The methods should check for specific log entries that indicate successful operations, error messages, or other significant events. For example, the test code might look for confirmation messages that certain functions were executed or check for the presence of error logs when a failure occurs. These methods should be designed to handle a variety of log formats and structures, allowing them to parse and interpret the log data effectively. Additionally, the validation methods should include assertions to compare the actual log entries with the expected results, ensuring that any discrepancies are promptly identified and addressed. By implementing comprehensive validation methods, testers can verify that the application behaves as intended and quickly detect and diagnose issues that arise during testing.

D. Code Example: Retrieving Logs in XCUITests

```
import XCTest

extension XCUIApplication {
    var logs: String {
        let textView = self.textViews["AutomationLoggingView"]
        return textView.exists ? textView.value as? String ?? "" : ""
    }

    func validateLogEntry(_ entry: String) -> Bool {
        return logs.contains(entry)
    }
}
```

Analysis

[1]. Advantages

A. Simplicity and Speed:

[1]. This approach is straightforward to implement, requiring minimal additional code.

[2]. It operates swiftly, ensuring rapid feedback during testing.

[3]. This approach can be adopted to any other platform where we may have restriction of accessing applications under test log or container.

B. Reliability:

[1]. Works seamlessly with both simulators and real devices without necessitating changes in code or configuration.

C. No External Dependencies:

[1]. The method does not rely on external servers or network connections, enhancing its robustness and reliability.

[2]. Disadvantages

A. Additional Code in Application:

[1]. Incorporating logging code into the application can introduce overhead and complexity, which must be managed appropriately.

[2]. This approach will not work if Application code is not owned by the same team which is writing the UI Test automation using XCUITest on Apple's platform.

B. Log Persistence:

[1]. Logs will be lost if the application is terminated, requiring strategies to handle such scenarios.

Future Work

A. Capturing Streaming Logs from stdout:

[1]. Explore capturing streaming logs directly from stdout using Apple APIs. This approach may require applying filters to reduce noise and extract relevant log entries.

[2]. Investigate methods to effectively filter and manage stdout logs to ensure that only pertinent log information is captured and stored.

B. Grabbing All Logs in Case of Failures:

[1]. Develop a mechanism within XCUI Test to capture comprehensive logs from the device in the event of test failures. This would involve obtaining all Sy diagnose logs and filtering them to isolate logs specific to the identified bundle ID.

[2]. Create efficient methods to handle large log files, ensuring that only essential log data is retained and analysed.

C. Find ways to get logs if application code is not owned by the same team or does not have access to application code to add additional UI Element to access the logs.

Conclusion

The approach of using UI elements to capture logs in XCUI Test presents a viable and efficient solution for obtaining application logs during Apple application's UI testing. By embedding a small, invisible UI element that can store log messages, testers can access valuable diagnostic information without needing to bypass Apple's sandbox restrictions. This method significantly simplifies the logging process, making it accessible directly from the testing environment. Despite some limitations, such as the additional code required within the application and the potential loss of logs if the app is terminated unexpectedly, its ease of implementation and reliability make it a compelling choice for many testing scenarios. The fact that it works seamlessly with both simulators and real devices without needing changes in code or configuration further underscores its practicality. Future work will explore additional methods for capturing and managing logs, including leveraging stdout and comprehensive device logs, to further enhance the testing process. Specifically, capturing streaming logs from stdout could provide real-time insights into application behavior, though this may require sophisticated filtering techniques to manage noise. Additionally, implementing a way to collect all relevant logs from the device in the event of test failures, including Sy diagnose logs, could offer a more holistic view of the application's state, enabling more thorough debugging and analysis. These advancements would build on the current approach, aiming to provide even more robust and detailed logging capabilities for XCUI Test

References

[1]. Bierig, Ralf, Jacek Gwiazda, and Michael J. Cole. "A user-centered experiment and logging framework for interactive information retrieval." Proceedings of the SIGIR 2009 Workshop on Understanding the User: Logging and interpreting user interactions in information search and retrieval. 2009.

[2]. Cinque, Marcello, Domenico Cotroneo, and Alessandro Testa. "A logging framework for the on-line failure analysis of android smart phones." Proceedings of the 1st European Workshop on App Roaches to Mobi Qui Tous Resilience. 2012.

[3]. Zhongruan, D. E. N. G., and Andreas Bauer. "Design and implementation of an advanced events logging framework for Android." (2011).

[4]. Apple UI Tests: https://developer.apple.com/documentation/xctest/user_interface_tests

[5]. Newlyn, Kenneth E. "A security analysis of system event logging with syslog." SANS Institute, no. As part of the Information Security Reading Room (2003).

[6]. Kuļešovs, Ivans, et al. "An Approach for iOS Applications' Testing." Baltic Journal of Modern Computing 6.1 (2018): 56-91.

[7]. Kulesovs, Ivans. "Automated Testing of iOS Apps: tTap Extension for Apple UIAutomation." (2015).

[8]. Tirodkar, Aditya Atul, and Sundeep Singh Khandpur. "EarlGrey: iOS UI automation testing framework." 2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft). IEEE, 2019.

[9]. Mishra, Abhishek. IOS code testing: test-driven development and behavior-driven development with Swift. Apress, 2017.

[10]. Relan, Kunal, and Kunal Relan. "Blackbox Testing iOS Apps." iOS Penetration Testing: A Definitive Guide to iOS Security (2016): 47-72.