Enhancing Automated Testing Capabilities on Non Platform-Dependent Mobile Applications

Hannah M. Palma
University of Utah

UUCS-21-001

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

25 March 2021

Abstract

Automated Testing software libraries have been created to take over simple and repetitive quality assurance tasks, and allow computers to assert correct user interface functionality quickly and consistently. This greatly increases the efficiency of human programmers and testers as they are able to spend their time on greater, less tedious problems, while programs automatically run large testing batches on themselves. Over the years, new testing frameworks have been developed, but there is still room for growth in testing capabilities and range of testing platforms. This thesis expands the capabilities of the Appium automated testing library on a non-conventional, non-native app by harnessing the use of Appium drivers.

ENHANCING AUTOMATED TESTING CAPABILITIES ON NON PLATFORM-DEPENDENT MOBILE APPLICATIONS

by

Hannah M. Palma

A Senior Thesis Submitted to the Faculty of
The University of Utah
In Partial Fulfillment of the Requirements for the Degree

Bachelor of Computer Science

School of Computing
The University of Utah
March 2021

Approved:

David E. Johnson Thesis Faculty Supervisor	H. James de St. Germain Director of Undergraduate Studies School of Computing
Mary Hall Director School of Computing	

Abstract

Automated Testing software libraries have been created to take over simple and repetitive quality assurance tasks, and allow computers to assert correct user interface functionality quickly and consistently. This greatly increases the efficiency of human programmers and testers as they are able to spend their time on greater, less tedious problems, while programs automatically run large testing batches on themselves. Over the years, new testing frameworks have been developed, but there is still room for growth in testing capabilities and range of testing platforms. This thesis expands the capabilities of the Appium automated testing library on a non-conventional, non-native app by harnessing the use of Appium drivers.

Table of Contents

Introduction

Before commercial software is released to the public, its functionality is usually tested to make sure that all features are working as intended and avoid any embarrassments. These tests often appear in the form of unit tests and GUI (Graphical User Interface) tests. Unit tests require extensive knowledge of the internal architecture of the application, and often involve substituting different internal structures for mock versions so that just one piece of the feature is isolated and tested at a time. They have the ability to test internal algorithms and structures that are not visible to a user. As a result, unit tests are usually written by a programmer at the time that they are developing a new feature (Runeson 2006). Interface tests focus on testing the application from the perspective of the user and do not require much (if any) knowledge of the internal workings of the software. This thesis focuses on improving the capacity of the second type of testing.

Many tech companies today have Quality Assurance teams dedicated to poring over their application's GUI and finding any weaknesses or bugs instead of systems and software to do this for them. The job itself of evaluating an app completely from start to finish can be very long, monotonous and repetitive work---and even more so when the tester has multiple browsers or mobile devices and operating systems that they must test the same software against. Test Automation is when a program is used to test another target program and report the results of everything that is functioning and malfunctioning, with the intention of saving time for human testers and allowing them to focus on greater problems. The program that performs the test against the target program does so by calling endpoints (or methods) provided by the platform or host browser that spoof human-like interactions with the user interface.

There have been a few approaches and frameworks that tackle testing on many devices or browsers consistently, and some of the most successful have been Selenium (browsers) and Appium (mobile devices). Both of these frameworks rely on some test automation features that have been built into operating systems and browsers, but they do have some limitations. As software becomes more accessible, popular and necessary to daily life, consumers are expecting high-performance apps in less time. One way that systems vendors are addressing this is through cross-platform development: using frameworks that allow them to develop one app for multiple platforms or operating systems at once (Palmieri, Singh & Cicchetti, 2012). Apps that are written to run on multiple platforms and do not use native user interface features provided by the operating system are unfortunately unsupported by testing frameworks. Also, another existing limitation for these frameworks is that there is a certain set list of commands that you can make to the device or to your app (like simulating a tap/touch, asking which elements are present, etc.) but it is very difficult to add any custom commands to send to your application to increase test capability and strength (like simulating a given testing environment, spoofing data, etc.).

Research Problem and Questions

After seeing these pros and cons and running into these specific weaknesses, the research problem for this thesis became:

Is it possible to adapt and improve existing test automation frameworks in order to service apps with non-native user interface elements?

The research questions that are asked in order to answer the research problem are:

- Is it possible to harness the existing multiple-driver system built into Appium to test a non-native UI mobile application?
- If successful, could this customization also increase the ability to test a native or non-native app by allowing for custom testing endpoints?

Background

Manual Testing vs. Automated Testing

Manual Testing has often been the first approach taken when establishing a quality assurance program (Taipale, Kasurinen, Karhu & Smolander, 2011). This usually involves a dedicated team that makes regular passes over the entire produced software, making sure that all endpoints (specific URLs that make up an API) or user interfaces (UIs) are working as expected. They often have to perform the exact same tests after every sprint, after every code change, or before every release, depending how the engineering organization is structured (Itkonen, Mantyla, & Lassenius, 2009). However, as the software complexity and speed of releases increases, the drawbacks to manual testing become more and more blatant.

Performing the same tests over and over again can be extremely monotonous work and is draining for employees to do for an extended period of time. This introduces a very real risk of testers becoming bored or complacent, and missing test steps and bugs (Dustin, Garrett, & Gauf, 2009). Even if the testers are focused on the test, it can be very difficult to perform the exact same action the same way repeatedly, especially if it requires a lot of set up, specific conditions or great attention to detail (Fewster & Graham, 1999). Humans do not excel at performing the exact same tedious task over and over, but machines do (Whittaker, 2010). Humans also are not so great at multitasking (Whittaker, 2010). This creates a bottleneck when a team is trying to scale for rapid growth. Finally, humans cannot operate at the same speeds as computers when it comes to advancing through APIs or screens in a UI and verifying behavior (Whittaker, 2010). Computers can perform stress tests on systems like no human can due to this speed.

These weaknesses can let buggy code slip into production and land in the hands of users, with product managers and customer support having to explain to clients how it was approved and released in the first place. Automated testing allows human testers to focus their time on problems that are more difficult for machines and easier for humans, like assessing quality and asking meta questions about the software ("Is it meeting the intended needs of our users? Is it easy to navigate? Is it accessible?"). It also increases the quality and efficiency of the small, repetitive test steps that need to be performed frequently (Adams, 2002). Automated tests can require skills such as programming and setting up development libraries, which makes it more difficult and a time investment but in general it also is more rewarding and mentally stimulating to quality assurance automation engineers than manual testing because of this (Honkanen, 2016).

As software continues to become more and more complex with new ways to interact with users and perform many more operations, automated testing libraries need to advance at least as quickly in order to keep up and maintain quality, but often they do not receive the same attention in research and development. It is the responsibility of each team to spend time and invest in exploring new and better ways to test their code that is compatible with their latest developments. Doing so is an investment in their own quality.

Pitfalls of Automated Testing

While automated testing comes with many advantages over manual testing, it is important to note that it still has some drawbacks that are worth acknowledging and working around if possible. Until we are able to develop more technology to bridge these gaps, knowing that they exist can help set expectations correctly.

Machines cannot see, understand and interpret in a flexible manner nearly as well as we can. They cannot interpret, add onto or improvise any test steps. They are not very good at interpreting results right off the bat. And most importantly, it is very easy to underestimate the time and the cost of creating and maintaining automated testing software. There is also a tendency to make the tests too brittle, where engineers are forced to spend almost more time maintaining the tests instead of working on their own product's development (Ramler & Wolfmaier, 2006). It takes a significant amount of time just to ramp up and prepare engineers to write successful tests, and then it also takes time to have the testing suite set up enough that it is adding enough of a real benefit to the product pipeline. It also can be a source of grumbling among a team when it comes to the added work of writing automated test scripts after a new feature has been worked on for many hours and days until completion. It also requires more time in peer review and approval which means getting more engineers up to speed and familiar with testing and more time overall in the team's code pipeline.

Automated testing also introduces more new tangible and intangible costs to a company. Tangible costs include the cost of hardware running the tests (mobile devices, servers, virtual machines, anything supporting those devices), licenses for tools and software (which may not always be free), direct training on these tools, and test environment implementation and maintenance (a testbed environment that models the production environment must be available and easily able to be reset for automated tests to run quickly and consistently) (Honkanen, 2016). Intangible costs include test case implementation (writing the scripts), test case maintenance, test execution, test results analysis (often more time-consuming than one would think), and personnel considerations (for example, perhaps quality assurance employees specializing in automation are needed on the team rather than normal QA workers, or perhaps more workers are needed) (Honkanen, 2016). All of the above outline real costs that can be quite significant, but the real return on investment lies in the test executions themselves. There are some base costs that are shared between automated testing and manual testing such as base planning, design, and defect and results reporting. These costs can be ignored while directly comparing manual and automated software testing (Hoffman, 1999).

Finally, engineers and product owners may think that automated tests might find more “new” bugs than manual testing, but that is not the case. Automated tests lack imagination because they are written once and run repeatedly, so it is better to have the expectation that they are used for maintaining quality and assuring that what was working before a code change is still working after it has been made. (Ramler & Wolfmaier, 2006).

Testing Approaches

There are a few different ways to perform software tests which target different areas of the product. White Box Testing is the detailed investigation of internal logic and structure of the code; it is necessary for a tester to have full knowledge of source code (Ehmer & Khan, 2012). An example of this may be unit testing or testing for memory leaks. Black Box Testing is defined as testing without having any knowledge of the internal workings of the application, examining the fundamental aspects of the system and has no or little relevance with the internal logical structure of the system (Ehmer & Khan, 2012). An example of this is our GUI testing.

Both white box and black box testing require some setup, although sometimes it may look different. GUI tests often depend on other services like a database or network, while unit tests do not. Despite these differences, the final goal of all of these tests is to know quickly and easily whether all features in the final project will be working as expected for the end user. For UI tests, this is achieved by setting up the software instance to a scenario that replicates what the end user will see and reproducing their interactions.

The results are then reported to the engineer running the tests on whether or not the GUI is responding correctly or not to those features (Neto, Subramanyan, Vieira, & Travassos, 2007).

Brief History

One of the most well-known testing libraries, Selenium (used for web application testing) once used an intermediary server to receive scripted commands and inject them as javascript into the browser as it loaded (Saucelabs). This architecture was very complicated and also dependent on the browser itself, and was very brittle when it came to browser updates and working with multiple popular browsers. Eventually, Selenium switched to a new, simpler architecture called Selenium WebDriver that does not require a server to serve as a proxy. Instead, it uses the browsers' native support for automation. Selenium WebDriver can receive scripted commands from many different languages using a consistent and globally recognized protocol called the JSON Wire Protocol (SeleniumHQ, 2016).

As mobile devices and applications have become exponentially popular and powerful over the last 15 years, it was clear that a similar automated testing library was needed to test mobile software. Appium was a new library that took a lot of direction from Selenium and debuted in 2013 (The JS Foundation). It functions in a similar fashion by relying on the device operating systems' native automation support in order to click buttons, swipe, take screenshots, etc. The Appium library has drivers which manage the relationship between the testing server and the device's operating system. There is an

Appium driver for Android, iOS and Windows that will handle basic testing for any application that uses native elements (stock platform UI components such as buttons and text boxes). However, there are many applications out there that do not use native elements; most often, gaming mobile applications will rely on a single image view (for example, a GL View from OpenGL) to draw an image to the screen and intercept touches. This allows developers to implement their own user interface and write it just once for multiple platforms in a common base coding language. These applications are unsupported by libraries like Appium that rely on the presence of those native/stock elements, and require completely new automated testing structures. They also do not adhere to any established automated testing protocol out of the box at this time.

Luckily, due to Selenium WebDriver's success using browser-specific drivers that can be interchanged, Appium used a similar framework structure when approaching mobile automated testing. Each driver harnesses into the operating system's specific native automated testing API.

Downsides of Native Testing

While the system of using native automation support is in theory simple and straightforward (it is in fact more complicated than it is for Selenium in a web browser but we can abstract away the differences), finesse in testing is arguably lost when only using platform automation. First off, because you rely on those native elements being present in the app's UI, you cannot test any application that does not use them as mentioned above. Secondly, after spending so much time and effort getting the automated testing set up, the tests themselves are quite simplistic, and mostly are only able to check if buttons and text boxes are present, set and check values on fields, and a few other things. They test the outermost layer of the UI, however they cannot help with much more and do not have a lot of testing framework support. For example, if you have an app that relies on the state of a particular user and need to test manipulating its data, if the script fails halfway through a test that is being run, the user will be stuck in a state that is not the same as when the script started, so it could cause failures for the next run. In other words, a vanilla Appium script cannot escape and reset a test very easily when the app is broken due to every app being different.

Methods

Custom Testing using a New Driver

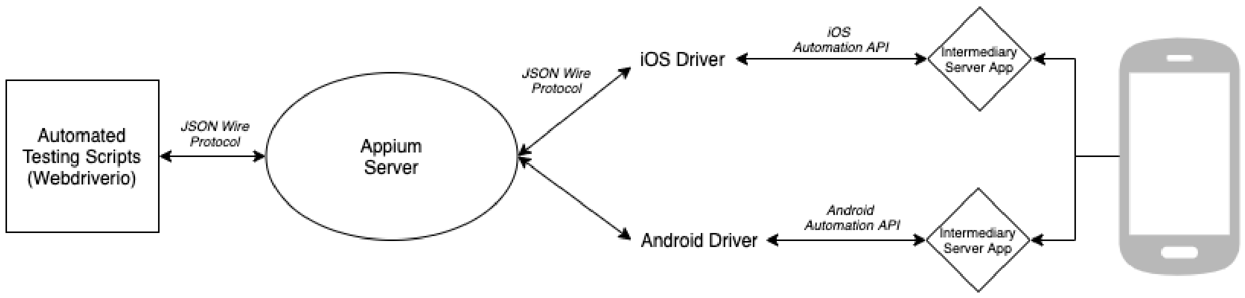


Figure 1, Appium original blueprint as is when downloaded or cloned.

Above is a figure depicting the current architecture of the Appium open source library. Test scripts send commands to the Appium Server, which selects a driver, eventually passing the call through an intermediary server app to the device's platform automation library. The library performs actions on the app's native UI elements and returns its results back through the same path it came. While the out-of-the-box testing capabilities that Appium provides are very appealing, it became clear that perhaps adding a new unique driver to Appium for a specific custom app would have added benefits on top of simply being able to run standard automated testing scripts. In addition to adding a new driver to Appium, the same API of testing actions had to be manually respected by the app (instead of the platform automation endpoints that look for native UI elements), and the app would then take action and return its result.

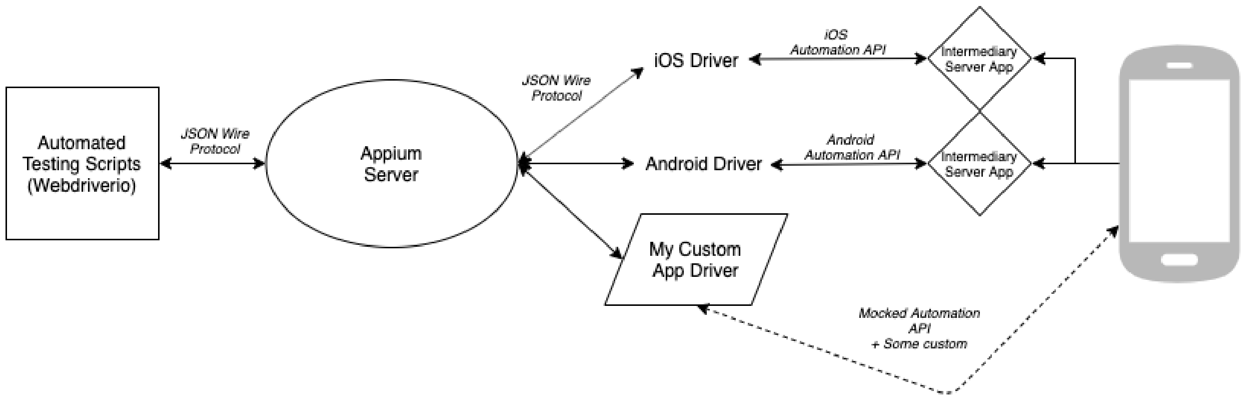
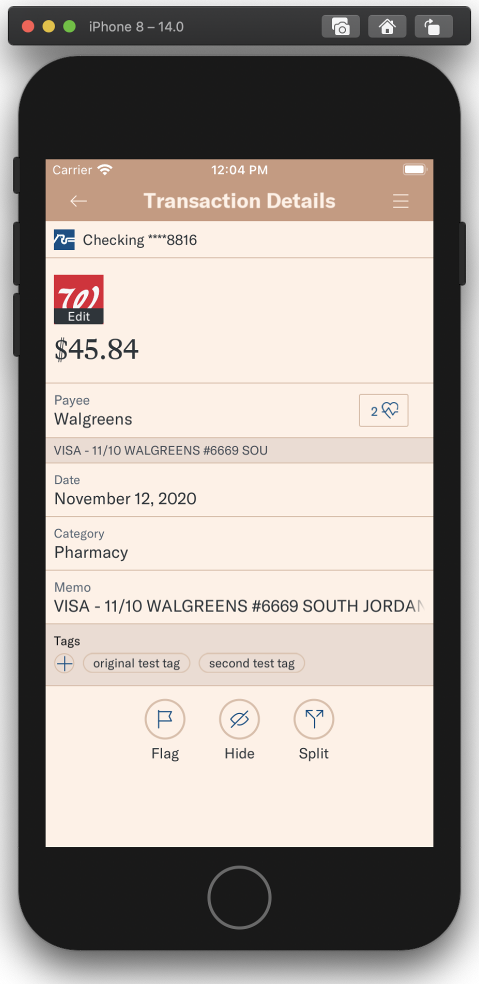


Figure 2, Planned Appium blueprint after adding Custom driver for custom app.

Figure 2 depicts the Appium open source software with the addition of the new Custom Driver, under the appearance of being a new platform. By creating the custom driver, the API then had adequate space to add as many more custom endpoints/testing actions as the developer could imagine. New custom features could include a trigger to reset the app or a particular user to a preferred testing state, insert or use mocked test data, get information about the state or configuration of the app before and during testing, measure analytics, trail logs and much more. From the perspective of the test scripts or the mobile app, the difference between the calls to the original API and the new endpoints became almost seamless. Because the Custom App Driver does not use the platform automation library, it didn't need an Intermediary Server App and instead simply established a direct connection via websocket to the app for sending commands and receiving results.

Corresponding App And Tests

These architecture enhancements required the research to include a simulation of a true app for development in order to add in this mocked API, and the ability to add in class with full permissions and access to the app database and all other running class instances as needed. This fully simulated the normal professional environment where developers and members of QA normally have access to add enhancements to the app codebase and run tests on it. The app created for research purposes consisted of one main screen appearing to belong to financial analysis software. It was constructed so that it had a large amount of data pieced together from multiple sources. The view also had a few different screens wired into it that could influence or change the data that it displayed. Each one of those screens had different UI components like buttons, edit boxes, images, search bars, and even a calendar. A layout like this with a good amount of moving parts would take a reasonable amount of time to test and would require specific attention from a manual tester. A human going through the screen to check its functionality would have to make repeated changes and verify every time that all data displayed was correct and only changing as expected. A larger screenshot of this screen can be found in Appendix 1 at the end of this thesis, and a screenshot of the Date Changer screen with the calendar UI component can be found in Appendix 2 for reference.



Checking ****8816

$45.84

Payee
Walgreens

VISA - 11/10 WALGREENS #6669 SOU

Date
November 12, 2020

Category
Pharmacy

Memo
VISA - 11/10 WALGREENS #6669 SOUTH JORDAN

Tags
original test tag
second test tag

Figure 3, View with many data entry types requiring multiple testing techniques.

This app was built on a third party video cross-platform game engine named Cocos2d-x, which meant that it wouldn't use native UI elements and could simulate an environment where normal out-of-the-box testing software would not be able to be tested.

Appium is an already established, open-source automated testing library that uses a documented protocol, which made it straightforward to research and respect the existing pieces of the API within the app. This API first and foremost needed to catch what the platform drivers would have originally done, like clicking on buttons, setting text to fields, etc. and reporting the results back to the scripts. After that, the opportunity would be present to add any custom calls to augment the capability of the testing framework.

Because the test scripts are intended to be written in full coding languages (this research used Javascript and the Webdriverio node module to connect to the Appium server), it was possible to add new methods to invoke the new custom test commands, which were sent through the Appium server to the device, which then returned back the results through the same channel to the test scripts.

Measuring Test Automation

In order to consider how to measure the growth of the Appium library's testing capabilities, it is helpful to consider the research problem of this thesis again: *Is it possible to adapt and improve existing test automation frameworks in order to service apps with non-native user interface elements?* The coordinating research questions are,

- Is it possible to harness the existing multiple-driver system built into Appium to test a non-native UI mobile application?
- If successful, could this customization also increase the ability to test a native or non-native app by allowing for custom testing endpoints?

First, the research had to construct the ability to respect this automation library on a cross-platform app in order to answer the first research question. After that, it was important to test and confirm whether or not the testing features of the framework were increased by taking the time to establish a custom testing connection for a non-conventional app. This thesis makes this comparison by documenting the total number of features that were able to be tested using the new custom testing commands versus the commands that would have been able to have been used on a vanilla Appium library. (Of course, on a non-native app that number would be zero due to Appium's inability to support non-native elements, but we can compare what it *could* have tested with this fact aside to consider if this approach has value even when applied to traditional native apps.) Another datapoint of interest that is recorded in the Conclusion of this paper is the recorded extra time taken to create the custom implementation of the testing API, versus the time humans must spend manually testing features that weren't covered without the custom testing abilities. These gathered data will support the research problem and questions above.

Development Process

I began work on this project by forking the open source Appium repository from Github. This step was necessary because there were a few places in the code where testing scripts needed to specify which platform and Appium driver they wanted to use. Unfortunately there was a select list of drivers that were allowed so I had to add my own to the whitelist in order for Appium to allow it to be used. I also forked Appium Base

Driver and made a few changes there to enable my own driver and set up my websocket server that rerouted received re-routed commands from the main appium server.

After this, I used the most simple existing platform driver I could find (Apple MacOS) and gutted what I did not need. From there, I found in the MacOS driver code where the server was taking the JSON Wire Protocol HTTP commands and converting them to accessibility & automation calls to the operating system. I wanted to keep my driver as simple as possible, so I just set up the custom driver to take whatever HTTP call it got and forward it directly onto the connected device's app for it to be handled in its entirety there (with the correct JSON Wire Protocol Response).

I added a new class to my app (based on the Cocos2dx framework) to make and handle websocket connections. Once the connection was established, my class would accept messages containing the original HTTP from the Appium server and then would call methods internally just like the MacOS driver's server would. These methods would do things like check for the presence of an element in the scene tree (the internal data structure representing the arrangement of UI elements), simulate taps and swipes, and read and edit fields.

Appium has several ways to identify a UI element or node as it is drawn in the app. These include finding it by xpath (not recommended, but is a string that represents how to navigate to a specific node in an xml tree; it is brittle and entirely dependent on the shape of the tree), accessibility id, class, and sometimes matchers. Accessibility ID is the preferred method of identifying a node (OpenJS Foundation, 2019). My UI components did not previously have any form of ID, and unfortunately Cocos2dx does not support accessibility at this time, so I added a “name” class member string onto my base component class. From there, for nodes that I needed to look up and use in a test, I manually added a unique node name that would be easy to use and look up. This made it easier to create readable test scripts when they were tapping on named elements instead of convoluted xpaths.

I had to frame out the test scripts while I was working on the mock platform API within my app so that I could test it and make sure that all calls were being handled as expected. Once I felt confident that I had addressed enough endpoints to start getting my tests rolling, I refocused my efforts there. I started out writing the scripts in plain Javascript ECMAScript 5 with a few libraries like Webdriverio (OpenJS Foundation, 2019) and Chai (provides the ability to assert behavior or fail the test, and provide error messaging) (Chai Community, 2018).

Results

The research problem at the crux of this thesis was:

> Is it possible to adapt and improve existing test automation frameworks in order to service apps with non-native user interface elements?

Below I will review each one of the corresponding research questions.

Is it possible to harness the existing multiple-driver system built into Appium

to test a non-native UI mobile application? The literature research conducted in this thesis highlighted the presence of an interchangeable driver system existing in the open source Appium library that was used in scripts to set up the tests and target the correct device, platform and app. I proposed that I could theoretically create a new custom driver specifically for my app and respect the automation API on my own. In my scripts, I could then select that custom driver and use it to send clicks, swipes, and other requests to the device running my app. This did not go completely as planned because as mentioned above, I did end up having to fork the Appium repository itself and make some changes to allow my new "Automation" to be permitted. Unfortunately, there were some systems in place to only allow specific drivers. After I got past this step, I was able to do this and demonstrate the feasibility of the research approach. I successfully created a test script that used my new driver and then wrote tests for each feature in one of the screens.

If successful, could this customization also apply for apps with native elements that are looking to add more custom features to their tests and improve their strength? Once the bulk of the work of creating a custom driver and adding the API to the app was complete, adding any extra custom endpoints did not involve a heavy lift and ended up having a bigger impact than I predicted on the total capability the framework had to test each feature.

	Existing Capability (on a native app)	**Capability after Custom Framework**
Navigating to test screen	★★	★★
Changing merchant	★	★★

Linked data appears when relevant		★★
Change payee	★	★★
Verify static amount		★★
Verify static account	★	★★
Verify static feed description		★★
Verify related transactions appears when relevant	★	★★
Change date	★	★★
Change category	★	★★
Change memo	★	★★
Change tags	★	★★
Flag state	★	★★
Hidden state	★	★★
Split	★	★★
Resetting User after test		★★

Table 1, New testing capabilities possible after establishing a custom testing framework. "★" represents little to some capability, "★★" represents moderate to complete testing capabilities, and a blank represents no testing capabilities. Each row represents an overview testing one feature, which in reality consists of many test steps.

As I performed these tests, there were a few places that stood out as examples where the custom framework would always outshine the existing testing endpoints. The first example achieved this by adding more “value” to existing endpoints that retrieve information about UI elements. For example, there is an image on screen that represents what Merchant is assigned to a transaction, but no readable text. By setting a “value” to the image representing the path to the actual asset itself, the scripts could register when a change was made to the image and when it was not. The second example of a great increase in testing capability was through the SQL querier, which passes a query to the database on the device itself and returns the results on what the appside “backend” contains so it can be compared with what the UI displays. The best applications of this feature were for verifying that static data was correct as displayed when the user first enters the view (things like transaction amount, feed description, account) as well as verifying that changes made to data in the UI were successfully applied to the database. Without this custom tool, a tester would have to either decrypt the local database on a desktop build of the app and inspect it, or unpin the network certs on all calls and use a network traffic tracker to verify that the original data synced from the backend was correct. Both of these manual approaches, if possible, would require a high degree of time, skill, and permissions---it’s not entirely reasonable to think that a standard Quality Assurance employee would have the ability to do these things. However, they could get a ballpark idea of the functionality by doing things like exiting and returning to the app and seeing if the same non-static data populated after it was changed. The ability to ask the database directly for the information it stores also reduces the number of external factors that testers need to rely on, like other services being up and running.

The SQL querier also proved extremely helpful in other cases outside of just static data. Another example of its strength is in its use to verify that the number of similar transactions displayed in the Similar Transactions button (see Appendix 1) was correct. If that feature was unavailable, I would have had to tell my script to walk manually through all transactions and count how many had the same payee. This would take an extremely long amount of time given that a normal user has thousands of transactions. It would be safe to say that making sure that number was displaying correctly would not be testable without the custom endpoint. If I wanted to test that number personally, I would have to perhaps search in the backend services for all transactions with a similar name and count there, again taking more time and requiring more knowledge on my part in addition to relying on more services. However, most engineers and testers do not have security access to run queries on the live prod databases, so at best, relying on the ability to test this feature manually is doubtful.

There was one more noteworthy endpoint added to help manipulate the state of the app, which is something that is not usually possible for human testers to perform. It would reset the state of the app and user to how it was expected to be at the beginning of the test suite. This step would occur in the “after” function of a set of tests so that when tests inevitably catch UI failures part way through an execution and halt, consecutive test runs that rely on state are not doomed to fail. This feature wasn’t as visible in the specific table of features that became more testable, but it did impact each feature as a whole in the scope of long term, non-interrupted testing. Other endpoints similar to this in larger apps could simulate states such as triggering onboarding UI without having to perform a drastic measure like uninstalling and reinstalling the mobile app.

Time spent writing a test script for a custom testing API in non-native mobile app	8 hours
Time spent to run complete automated test script	0.1 hours
# Automated script executions needed to make up for script development	80 executions
Total days before time debt is met at 10 test executions per day	8 days

Table 2, Table outlining the time spent writing a testing script and calculating how long it would take before the script paid for itself.

After analyzing the impact on each feature in the app simulation, I measured the total time spent on completing this task and calculated how long it would take to make a return on the time investment of writing test scripts. Once the framework is set up, writing a scripted test for one view still can take a substantial amount of time, practice and skill. However, the time is gained back quickly once the author is able to run the scripts repeatedly as needed through the Appium server, which is faster and much more consistent than a human tester. After 80 automated test iterations, the time spent writing the script is returned, but also the full time of testing the feature with human testers has been saved.

One last unexpected observation that came of this research was that a good amount of time was saved due to the inspiration of the Appium project. Because Appium uses an established protocol, adding custom endpoints and features following the same pattern was simple and reduced development time.

Conclusion

Using the automation driver system in Appium for custom testing was straightforward to implement. While it did require an investment in time and patience, it allows the app developer the chance of creating automated testing scripts for an app that normally would not have any solution available out of the box. Once the initial connection was complete, it also allowed for more custom tools to be built on to increase the ability to test an app deliberately and specifically while maintaining the benefits of reproducibility, consistency and speed found in automated testing. Through my research, I was able to test difficult features and components that were unavailable to the original Appium library, but testable for humans such as images. I was also able to simulate environments and test difficult features that a human may not have been able to test such as resetting a user’s state and comparing UI data with the device database.

I would recommend following the approach proved above to create a custom driver and custom testing endpoints for any mobile app, native or non-native. The returned investment on time and effort will not only enable engineers to test apps with custom interfaces, but will greatly increase their ability to test and maintain their software.

References

Adams, E. (2002). The business argument for investing in test automation. Retrieved November 4, 2019, from http://www.ibm.com/developerworks/rational/library/content/RationalEdge/dec02/TestAutomation_TheRationalEdge_Dec2002.pdf.

Chai Community. (2018, September 25). chai. Retrieved December 20, 2019, from https://www.npmjs.com/package/chai.

Dustin, E., Garrett, T., & Gauf, B. (2009). Implementing automated software testing how to save time and lower costs while raising quality. Upper Saddle River (N.J.): Addison-Wesley. ISBN 978-0-321-58051-1.

Ehmer, M., & Khan, F. (2012). A Comparative Study of White Box, Black Box and Grey Box Testing Techniques. International Journal of Advanced Computer Science and Applications, 3(6), 12-15. doi: 10.14569/ijacsa.2012.030603.

Fewster, M., & Graham, D. (1999). Software test automation: effective use of test execution tools. ACP press/Addison-Wesley Publishing Co. ISBN 0-210-33140-3.

Honkanen, H. (2016). Investigating Effects of Test Automation In a Large Software Project: A Case Study [Master's Thesis, Computer Science, Aalto University]. *Aalto University*. https://aaltodoc.aalto.fi/bitstream/handle/123456789/20153/master_Honkanen_Heikki_2016.pdf.

Itkonen, J., Mantyla, M. V., & Lassenius, C. (2009). How do testers do it? An exploratory study on manual testing practices, 494-497. *2009 3rd International Symposium on Empirical Software Engineering and Measurement*. doi:

10.1109/esem.2009.5314240.

Neto, A. C. D., Subramanyan, R., Vieira, M., & Travassos, G. H. (2007). A survey on model-based testing approaches, 31-36. *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies Held in Conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007 - WEASELTech 07*. doi: 10.1145/1353673.1353681.

OpenJS Foundation. (2019). Selectors · WebdriverIO. Retrieved December 19, 2019, from https://webdriver.io/docs/selectors.html.

Palmieri, M., Singh, I., & Cicchetti, A. (2012). Comparison of cross-platform mobile development tools. 2012 16th International Conference on Intelligence in Next Generation Networks, 179-186. doi: 10.1109/icin.2012.6376023.

Ramler, R., & Wolfmaier, K. (2006). Economic perspectives in test automation: balancing automated and manual testing with opportunity cost, 85-91. *Proceedings of the 2006 international workshop on Automation of software test* (*AST '06*). Association for Computing Machinery. doi: https://doi.org/10.1145/1138929.1138946.

Runeson, P. (2006, July 17). A survey of unit testing practices. IEEE Software, 23(4), 22–29. doi: 10.1109/MS.2006.91.

Saucelabs. (n.d.). A Brief History of the Selenium Testing Framework. Retrieved December 2, 2019 from https://saucelabs.com/blog/a-brief-history-of-the-selenium-testing-framework.

SeleniumHQ. (2016, February 25). JsonWireProtocol. Retrieved December 2, 2019, from https://github.com/SeleniumHQ/selenium/wiki/JsonWireProtocol.

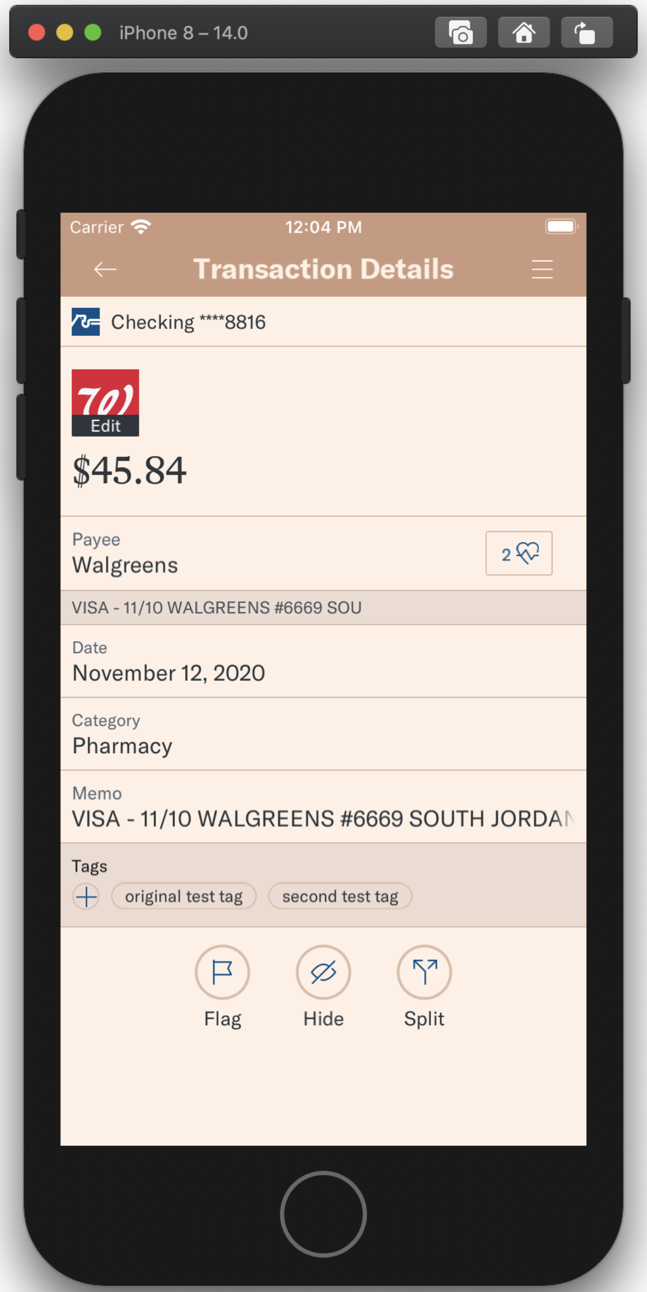
Taipale, O., Kasurinen, J., Karhu, K., & Smolander, K. (2011, April). Trade-off between automated and manual software testing. International Journal of System Assurance Engineering and Management, 2(2), 114–125. doi: 10.1007/s13198-011-0065-6.

The JS Foundation. (n.d.). The History of Appium. Retrieved December 2, 2019 from http://appium.io/history.html.

Whittaker, J. A. (2010). Exploratory software testing. Upper Saddle River, NJ: Addison-Wesley.

Appendices

Appendix 1 - Transaction Details Screenshot



Checking ****8816

$45.84

Payee
Walgreens

VISA - 11/10 WALGREENS #6669 SOU

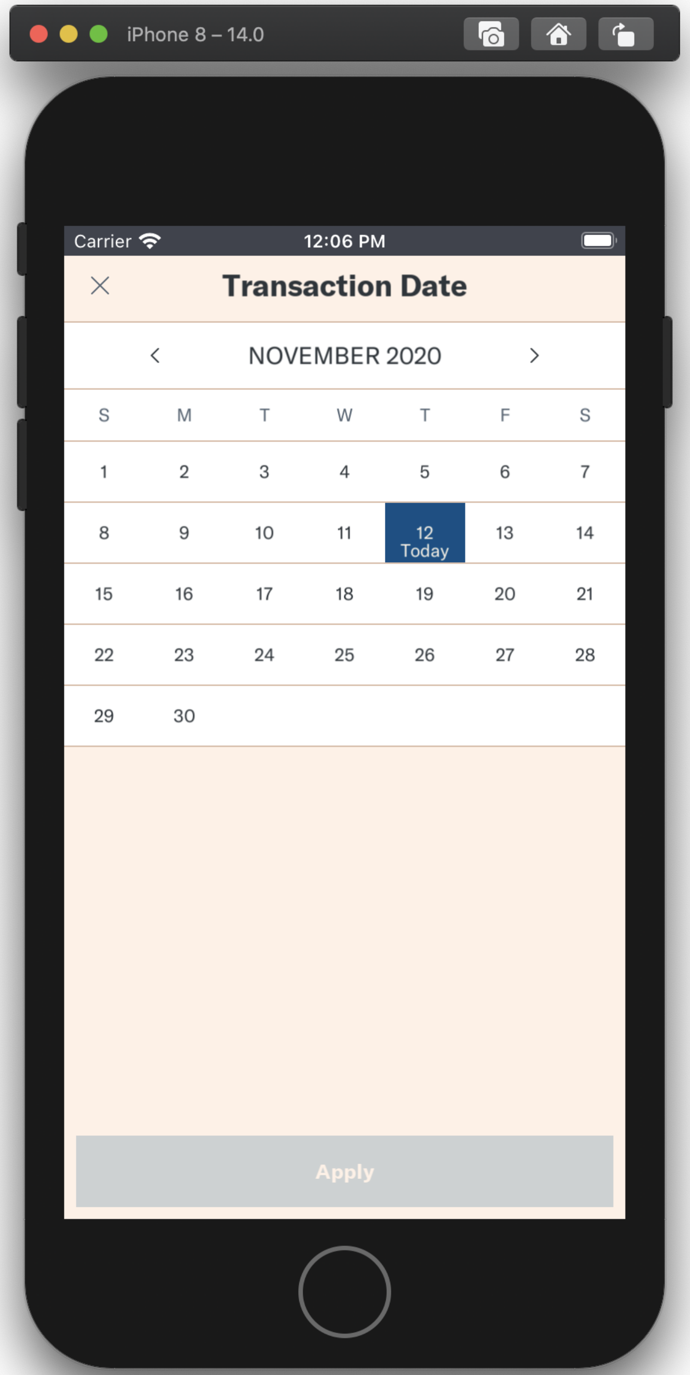
Date
November 12, 2020

Category
Pharmacy

Memo
VISA - 11/10 WALGREENS #6669 SOUTH JORDA

Tags
original test tag
second test tag

Appendix 2 - Date Changer Screenshot



S	M	T	W	T	F	S
1	2	3	4	5	6	7
8	9	10	11	12 Today	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30					

Appendix 3 - Sample Script Code for Starting a Test session

Selecting a Custom Automation Driver:

```
function getCustomOptions () {
  const customCaps = {
    platformName: global.deviceInfo.deviceOsName,
    automationName: 'MyCustomDriver',
    deviceName: global.deviceInfo.deviceName,
    uuid: global.deviceInfo.uuid,
    newCommandTimeout: 1500,
    App: ‘MyCustomApp’
};

  const customOptions = Object.assign({ capabilities: customCaps }, getServerConfig());
  return customOptions;
}
```

Using the Custom Capabilities above to start a test session:

```
const { remote } = require('webdriverio');

async function startDriver () {
  // Driver is used internally to make all Webdriverio calls
  _driver = await remote(getCustomOptions());
}
```

Example of a wrapper function that calls the Webdriverio actions:

```
// Finds center point of node and taps it.
// Error returned & step failed if node is not clickable.
async function click (name) {
  await _waitForClickable(name, 'Click');
  const element = await await _driver.$(`~${name}`);
  await _driver.elementClick(element.elementId);
}
```

Appendix 4 - Sample Script Code for Transaction Details

```
const assert = require('chai').assert;
const { initValues, customDeviceCommand } = require('../../../helpers/utils');
const { click, getText, setText, isDisplayed, isNotDisplayed, waitForNotDisplayed,
waitForDisplayed, scrollTo, scrollToTop, scrollToBottom, pause } =
require('../../../helpers/actions');
require('../../../helpers/global-hooks');

describe('Transaction Details', function () {
  initValues();

  it('Taps Menu Button if it\'s visible', async function () {
    if (await isDisplayed('Menu Button')) {
      await click('Menu Button');
      await waitForDisplayed('Main Menu: Sync');
    }
  });

  it('Taps Go To Transactions', async function () {
    await click('Main Menu List: Go To Transactions');
    await waitForDisplayed('Transaction View');
  });

  it('Finds a transaction', async function () {
    await findNextTransaction();
  });

  testEditMerchant();
  testPayee();
  testStaticDataForCorrectness(); // Test after payee since we use that field
  testMerchantHistory();
  testDate();
  testCategory();
  testMemo();
  testTags();
  testFlag();
  testHide();
  testSplit();
});

...

function testEditMerchant () {
  describe('Edit Merchant', function () {
    it('Checks current selected merchant', async function () {
      previousValues.merchantLogo = await getText('Merchant Logo');
    });

    it('Taps edit merchant', async function () {
      await click('Edit Merchant Logo Overlay');
      await waitForDisplayed('Merchant Search PopTart');
    });

    it('Selects a different popular merchant', async function () {
      // Make sure we have some results showing
      await waitForDisplayed('Merchant List: Item 1');

      // Pick a popular merchant that is different than the current transaction's logo
      let index = 1;
      let found = false;
      let logo = '';
      while (!found && index < 3) {
        logo = await getText(`Merchant List: Item ${index} - Merchant Logo`);
        if (logo !== previousValues.merchantLogo) {
          found = true;
        } else {
          index++;
        }
      }
      assert.isTrue(found, 'Unable to find a Merchant that is different than the
current transaction merchant');
      newValues.merchantLogo = logo;
      await click(`Merchant List: Item ${index}`);
      await waitForNotDisplayed ('Merchant Search PopTart');
    });

    it('Verified merchant logo changed', async function () {
      let newDisplayedLogo = await getText('Merchant Logo');
      assert.isTrue(previousValues.merchantLogo !== newDisplayedLogo, 'Merchant Logo is
still displaying its original logo after changing');
      assert.isTrue(newValues.merchantLogo === newDisplayedLogo, 'Merchant Logo is not
displaying the new selected logo');
      previousValues.merchantLogo = newDisplayedLogo;
    });

    it('Taps edit merchant', async function () {
      await click('Edit Merchant Logo Overlay');
      await waitForDisplayed('Merchant Search PopTart');
    });

    it('Searches for a merchant', async function () {
      await setText('Merchant Search PopTart: Search Bar', 'target');
      await pause(2);
      await waitForDisplayed('Merchant List: Item 0');
      newValues.merchantLogo = await getText('Merchant List: Item 0 - Merchant Logo');
      await click('Merchant List: Item 0');
      await waitForNotDisplayed ('Merchant Search PopTart');
    });

    it('Verifies merchant logo changed', async function () {
      const newDisplayedLogo = await getText('Merchant Logo');
      assert.isTrue(previousValues.merchantLogo !== newDisplayedLogo, 'Merchant Logo is
still displaying its original logo after changing');
      assert.isTrue(newValues.merchantLogo === newDisplayedLogo, 'Merchant Logo is not
displaying the new selected logo');
      previousValues.merchantLogo = newDisplayedLogo;
    });

    it('Opens Merchants back up again', async function () {
      await click('Edit Merchant Logo Overlay');
      await waitForDisplayed('Merchant Search PopTart');
    });

    it('Cancels', async function () {
      await click('Merchant Search PopTart: Cancel');
      await waitForNotDisplayed ('Merchant Search PopTart');
    });

    it('Verifies merchant logo is the same', async function () {
      let newDisplayedLogo = await getText('Merchant Logo');
      assert.isTrue(previousValues.merchantLogo === newDisplayedLogo, 'Merchant Logo is
displaying something different than it was after cancelling poptart');
      previousValues.merchantLogo = newDisplayedLogo;
    });

    it('Opens Merchants back up again', async function () {
      await click('Edit Merchant Logo Overlay');
      await waitForDisplayed('Merchant Search PopTart');
    });
    it('Clears merchant', async function () {
      await click('Merchant Search PopTart: Clear Merchant');
      await waitForNotDisplayed ('Merchant Search PopTart');
    });

    it('Verifies merchant logo changed', async function () {
      let newDisplayedLogo = await getText('Merchant Logo');
      assert.isTrue(previousValues.merchantLogo !== newDisplayedLogo, 'Merchant Logo is
still displaying its original logo after clearing merchant');
      previousValues.merchantLogo = newDisplayedLogo;
    });
  });
}

...

function testStaticDataForCorrectness () {
  describe('Account, Amount, Feed Description', function () {
    it('Changes the payee to something unique', async function () {
      previousValues.payee = await getText('Transaction Detail Payee Edit Box');
      newValues.payee = randomStr();
      await setText('Transaction Detail Payee Edit Box', newValues.payee);
      // Bulk Renaming modal
      if (await isDisplayed('DialogContainer')) {
        await click('Dialog Cancel Button'); // "This time only" button
      }
    });

    it('Reads account field', async function () {
      previousValues.accountName = await getText('Account Display Name');
    });

    it('Reads amount field', async function () {
      previousValues.amount = (await getText('Amount')).replace('$', '');
    });

    it('Reads feed description field', async function () {
      previousValues.feedDescription = await getText('Feed Description');
    });

    it('Looks for this transaction in the database', async function () {
      const query = `select * from transactions where \
        description = "${newValues.payee}" and \
        amount = ${previousValues.amount} and \
        feed_description = "${previousValues.feedDescription}"`;

      const columnsAndRows = (await customDeviceCommand('sql', JSON.stringify({ query
}))).value;
      assert.equal(columnsAndRows.error, '', `Error was returned after query:
${columnsAndRows.error}`);
      assert.notEqual(columnsAndRows.rows.length, 0, 'No matching transaction was found
in the database, some of the displayed data appears incorrect'); // Broken out for
separate error message
      assert.equal(columnsAndRows.rows.length, 1, 'Multiple transactions matched this
one in the database, should have found just one, some of the displayed data appears
incorrect');
    });

    it('Resets original payee name', async function () {
      await setText('Transaction Detail Payee Edit Box', previousValues.payee);
    });
  });
}

... Tests continue.
```