WHITE PAPER

AUTOMATED MOBILE APP TESTING USING EMULATORS, SIMULATORS AND REAL DEVICES

Simulators, emulators and real devices each play a key role in continuous test automation for mobile. The proper strategy must analyze and determine the right amount of each for the best testing approach.

SAUCELABS

TABLE OF CONTENTS

EXECUTIVE SUMMARY

The mobile deploy pipeline is much more complex than a web application, which are often built and tested locally. Instead of local testing, the code needs to be installed on one or more mobile devices. Instruction and tooling will need to be installed in order to run automated tests; this may need to be coordinated across multiple devices. If the "device" is actually in memory on the laptop, plumbing needs to be installed to get the automation to find the virtual device. Of the handful of choices, those local, virtual devices operate the least like the real thing. This article will cover some fundamental tools -- simulators, emulators, and real devices -- for testing mobile applications, and discuss where they fit into the test process. It will also cover when to test manually, and also how to reuse test tools on different platforms.

EMULATORS SIMULATORS AND REAL DEVICES

Emulators, simulators, and real devices all function slightly differently, and are used by different people on the team.

Simulators: Simulators are essentially a virtual machine running on a desktop or laptop. Simulators offer the fastest testing platform, but also the furthest away from what a customer will actually encounter when they use the application. Simulators are typically used in the context of a development cycle. An iOS developer might write a few lines of code in Swift to add a new text field to an existing page, and have data from that field submitted to an existing API when the form is submitted. They write the code, write a couple of unit tests to verify that data can be saved and that nothing bad happens when the field is null. Before adding the code to version control the developer needs to really see the software running locally.

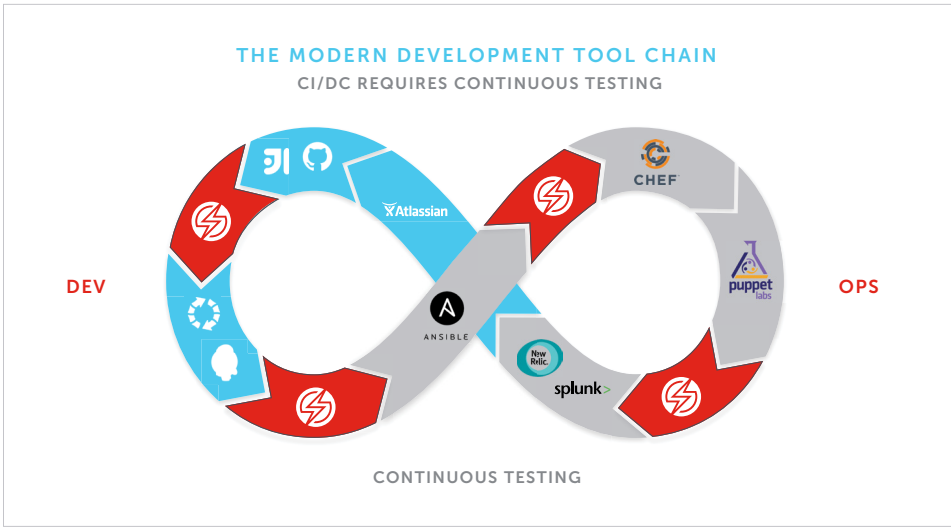
Emulators: Emulators are either accessed through a web browser or launched from a developer environment on the desktop; their main benefits are scale and availability. A tester can access their emulation environment, select a device type and an operating system, and almost instantly have a running version of a mobile device. This is fairly similar to Virtual Machines. Emulators are generally the next step in the development and testing cycle after a developer has built some new code, built unit tests that can run in Continuous Integration, and then done their own testing on a simulator. Unlike real device testing, which will be described next, emulators are infinitely scalable. Access to emulators is purchased through a service provider. Companies can get access to as many different devices, for as long as they want them, as long as they are willing to pay.

Devices: Using real mobile devices provides benefits not available with emulators and simulators. These devices can be in the cloud, connected through a tool, or an actual physical device in the hand. Developers and testers using real devices will get a more authentic experience, something much closer to what customers will see when they access the software. There are two main considerations for teams that use real devices as part of their mobile testing strategy -- device selection and test classification.

YOUR PIPELINE

Designing the build deploy pipeline is one of the first things teams will need to do when considering a mobile automation project.

A developer, or in some cases teams of developers in different parts of the world will be making code changes. Developers might test a change locally on their machine using an emulator, and then commit to a feature branch or a release branch. Committing the code creates a pull request. That pull request gets reviewed by another developer for easy to catch bugs, and stylistic consistency, and then merged into the appropriate branch. Merging the code change triggers unit tests to kick off, and if those are green, integration tests that cover the service layer after that.



When unit and integration tests come back green from Continuous Integration, deployment to a test and staging environment begin. When the server code is deployed to a test environment and complete, and for native applications, the software is installed on the target devices, then the mobile UI automation test suite can run.

Over time, this test suite can grow to take hours to provide feedback, lowering the value of that feedback. Designing the tests to run in parallel

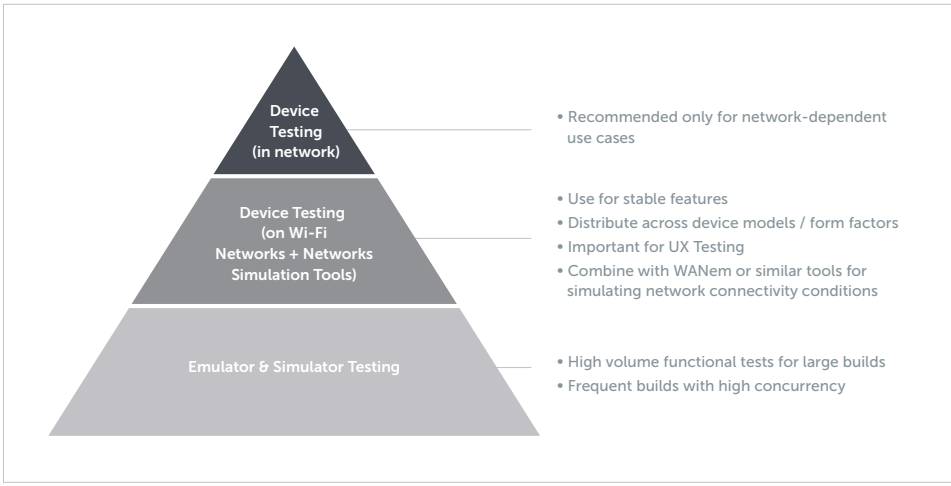
from the start can prevent these problems. The entire point of a build and deploy pipeline is facilitating fast feedback for the development team, and the people making release decisions. A mobile UI automation suite run on one environment might take an hour or more to complete. Using simulators, a team could spin up as many instances as they need to get feedback from their test suite in a reasonable amount of time. Rather than tests taking hours in a single environment, they might complete in 15 minutes using simulators running at the same time.

The general idea here is to get feedback about a code change at the earliest point possible. This reduces many of the low hanging fruit type bugs that testers spend their time finding, so that they can focus their efforts on deeper and more complex testing tasks with real devices.

THE PYRAMID

Most people that have worked in software testing and development are familiar with the test automation pyramid. The general idea here is to focus automation efforts on the parts of the product where it is fastest to create and maintain, and also cheapest to develop. Mobile test automation has its own pyramid based on where in the tool stack people should focus their efforts -- simulators, emulators, and real devices.

The base of this pyramid is made up of simulators and emulators. Developers and testers use these tools for expediency and repeatability. Both emulators and simulators are very easy to spin up as a new test environment. They also have none of the lag time a real test environment has. A developer might write a few lines of code, kick off their local simulator to investigate those changes quickly, and then run a full automation suite using a cloud based emulator to get a better feel for where they are.



HOW TO DO IT IN CODE

The ideal test tool strategy would be to have one framework to target web or native applications on emulators, simulators, or real devices. Using that framework, a tester will create a test a single time, then reuse that test on different devices, platforms, and locations based on a simple switch.

The test needs to handle responsive design - to adapt the test activities based on the user interface. One way to do this is by separation of concerns.

The page object controls behavior. Sample C# code for a login form might be something like this, where m_driver is a browser driver object that is a member of the class:

```
public void login(string username, string password) {
  driver.FindElement(USERNAME_FIELD).Clear();
  driver.FindElement(USERNAME_FIELD).SendKeys(username);

  driver.FindElement(PASSWORD_FIELD).Clear();
  driver.FindElement(PASSWORD_FIELD).SendKeys(username);

  driver.FindElement(SUBMIT).Click();

}
```

The function belongs to a page object class - but not the login page object. This is the desktop login page object. It is served up by the desktop page object factory. There is also a tablet page object factory, a phone page object factory, and a tablet page object factory.

At run time, the test run adds a series of form factors to a list, then iterates through that list. Each item in the list has an associated factory. The tests don't change. Instead, tests call methods on their page object -- but the page objects are switched in at run time for the appropriate device. This allows the tester to create one test per scenario, but reuse the test across various devices.

Switching between platforms (simulators, emulators, desktop) is a similar process - create a factory that takes the type of connection in as a string, then write the code to connect to that device (virtual or real), passing the object that can control the browser to the actual test itself.

This process of separating concerns and creating them through a factory object, then passing that object in for reuse, is known as Dependency

Injection, or "DI." The open-source framework that Sauce Labs has created to enable this is callled All The Things Digital - https://github.com/SauceCon2017-demo/allTheThingsDigital. Currently All The Things Digital is a tool for Java and TestNG but plans exist to support other webdriver languages. For more information, watch the talk on the topic at SauceCon 2017 https://www.youtube.com/watch?v=saD0gwLWm3k.

The cycle time to get code on physical devices will be slower than the time to get code on emulators and simulators, and physical devices will be harder to scale, so test runs will take longer. Still, with the right architectural design, it will be possible to re-use automated tests on the physical devices, perhaps running in a slower but still continuous loop.

FITTING THESE TOOLS ON YOUR TEAM

Each of these tool options fits in a specific place during the development cycle, and will probably be used by particular people on your technical team.

Simulators come into play during development. A developer does not have to leave XCode to get a running version of the product. Once that person thinks they have enough new code to test, they click a button to select what device and operating system they want to use. This builds the software, and then opens up a new test environment that looks just like the screen of a mobile device. The simulator offers the ability to see the latest code in action. Once the simulator is running, that developer can see the new text field, enter strings into it, click submit, and then see how the app behaves and whether not the data saves correctly. But, simulators are also missing much of what real mobile devices offer and they don't scale well.

This is where emulators enter the picture.

The main use case for emulators is mobile test automation. Each sprint, a team will build and modify their bank of automated tests. Of course, this also means that each sprint the set of tests is getting a little bit more complicated and taking a little longer to complete. Testers can solve this problem by using simulators. They start by splitting up the suite of mobile UI tests into groups. This enables what is called parallelization. Rather than kicking off one long test suite at the end of each build, a new emulator is started dynamically based on how many test suites there currently are. Each of these emulators comes live and runs a group of tests at the same time. When done this way, test suites take minutes instead of hours. They also have the benefit of providing a brand new, clean test environment every time the test suite is run.

The development group can't walk around with a simulator or emulator. They can't do real gestures, taps, and device rotation. Testers can't watch device resources drain while the app is running, they don't know what will happen when connectivity drops, and they will not be able to feel the change in temperature. Real mobile devices, sometimes augmented with cloud service providers, can make the physical mobile testing problems much more approachable.

Emulators are great for quickly providing a variety of test devices, and making test automation complete in a reasonable amount of time, but this is still no replacement for a device in hand.

Automating tests for these devices can be a challenge - not only in switching out the device, but also in dealing with responsive design. Activities like log in and search can actually have different behaviors at different resolutions. Smaller screen resolutions associated with phone and tablets have less screen real estate, so a login button might require the user to first select from a menu. That requires a different test - or at least a different way of writing test programs.

WHAT TO DO TOMORROW

Like any test strategy, development and test groups should blend the tool and approaches they want based on how their development flow works today, what software they are developing and the technology stack being used, and the available skill set.

Teams with a strong culture of automation, such as those that are moving towards DevOps and Continuous Delivery will use simulators and emulators heavily. Each development cycle will be accompanied by some new automated tests, or some old tests updated for new functionality. Each time the app is built, a new group of emulators gets spun up as an automation test bed. The emulator bank is nearly endlessly scalable, so as the number of tests grow, so do the number of available emulators to run them on. The remaining testing is done on a combination of devices in house, and real devices in house to catch issues that automation can't.

Teams further off from continuous delivery may focus more of their time on hands on testing. They will spend their time analyzing which devices their customers are likely to have so that the right devices can be purchased for a test lab. The development group may do some cursory testing -- submit

forms, and navigate through through the app -- using a simulator, and then testers spend most of their time with a mobile device in their hands. In this context, automation is generally built by testers when they have spare cycles.

Every mobile testing option -- simulator, emulator, or real device -- has its place in the mobile development cycle. The trick is analyzing where a development group is right now, and finding the right amount of each to sprinkle into their test approach.

ABOUT SAUCE LABS

Sauce Labs ensures the world's leading apps and websites work flawlessly on every browser, OS and device. Its award-winning Continuous Testing Cloud provides development and quality teams with instant access to the test coverage, scalability, and analytics they need to rapidly deliver a flawless digital experience. Sauce Labs is a privately held company funded by Toba Capital, Salesforce Ventures, Centerview Capital Technology, IVP, Adams Street Partners and Riverwood Capital. For more information, please visit saucelabs.com.

saucelabs.com/signup/trial

FREE TRIAL

SAUCELABS

SAUCE LABS INC. - HQ
116 NEW MONTGOMERY STREET, 3RD FL
SAN FRANCISCO, CA 94105 USA

SAUCE LABS EUROPE GMBH
C/O WEWORK STRALAUER
ALLEE 6 10245 BERLIN DE

SAUCE LABS INC. - CANADA
134 ABBOTT ST #501
VANCOUVER, BC V6B 2K4 CANADA