Way2Automation
Expert level Automation courses
W2A

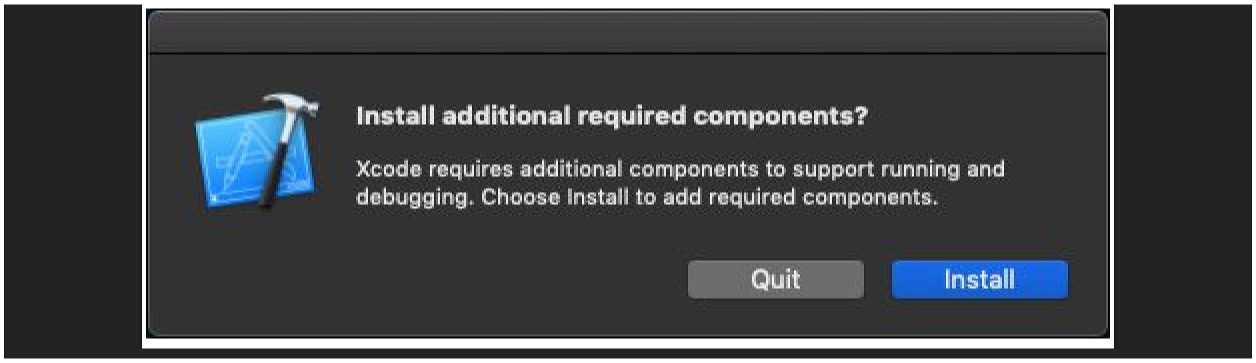
How To Test On Real iOS Devices With Appium (Important Notes By Appium Developer and Contributor Jonathan Lipps)

Getting set up with real iOS devices can be a long and involved process, this first edition will cover all the preliminary work you need to do to make sure Appium, your app, and your device will all be able to talk to each other. Stay tuned for next week when Jonah will show us how to put it all together.

This edition of Appium Pro is about getting started with a real iOS device test starting from scratch. Real iOS device testing can only take place on a Mac. (NB: This tutorial was written with OS X v10.14, Xcode version 10.0, automating an iPhone X S running iOS version 12.0.1. While the tutorial should remain largely correct for other versions, your mileage may vary!)

Organize Our Work Area

First things first, let's clean up our workspace and get ready to start a new project. Let's make sure all our software is up to date. To begin with, check that we have the latest version of Xcode installed. Then, since we will be using Homebrew, let's run brew update. Next let's launch Xcode in case it hasn't been launched before. (You will need to accept the End User License Agreement and may have to click a button to "install additional components").


Install additional required components?

Xcode requires additional components to support running and debugging. Choose Install to add required components.

Next we will install libimobiledevice, an open source package which is able to communicate with iOS devices. The only official applications made by Apple for communicating with iOS devices are Xcode and iTunes. These applications weren't built to easily allow programmatic use, which is why Appium uses libimobiledevice for certain operations.

brew install libimobiledevice

Appium also uses a package called ios-deploy for transferring iOS apps onto your device, so let's install that too.
brew install ios-deploy

WDA itself requires an iOS dependency manager called Carthage. Since Appium will be automatically building the WDA app, we need to install Carthage so it is available to the WDA bootstrap process.

brew install carthage

OK, now we've installed some things that Appium needs, but let's make sure we have Appium installed as well. I'm going to install the Appium Desktop App. Download the latest Appium-x.x.x.dmg file from the releases page. Open the file and drag the Appium.app icon into your Applications folder.

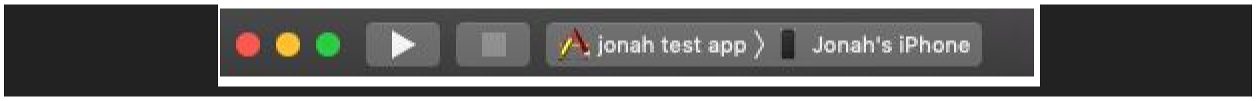
Finally, with the housecleaning on our Mac now finished, let's make sure our iOS device is set up:

- Turn the iOS device on
- Plug the device into your Mac
- Unlock the device
- Wait for a dialog to appear on the device and choose to "Trust" the Mac

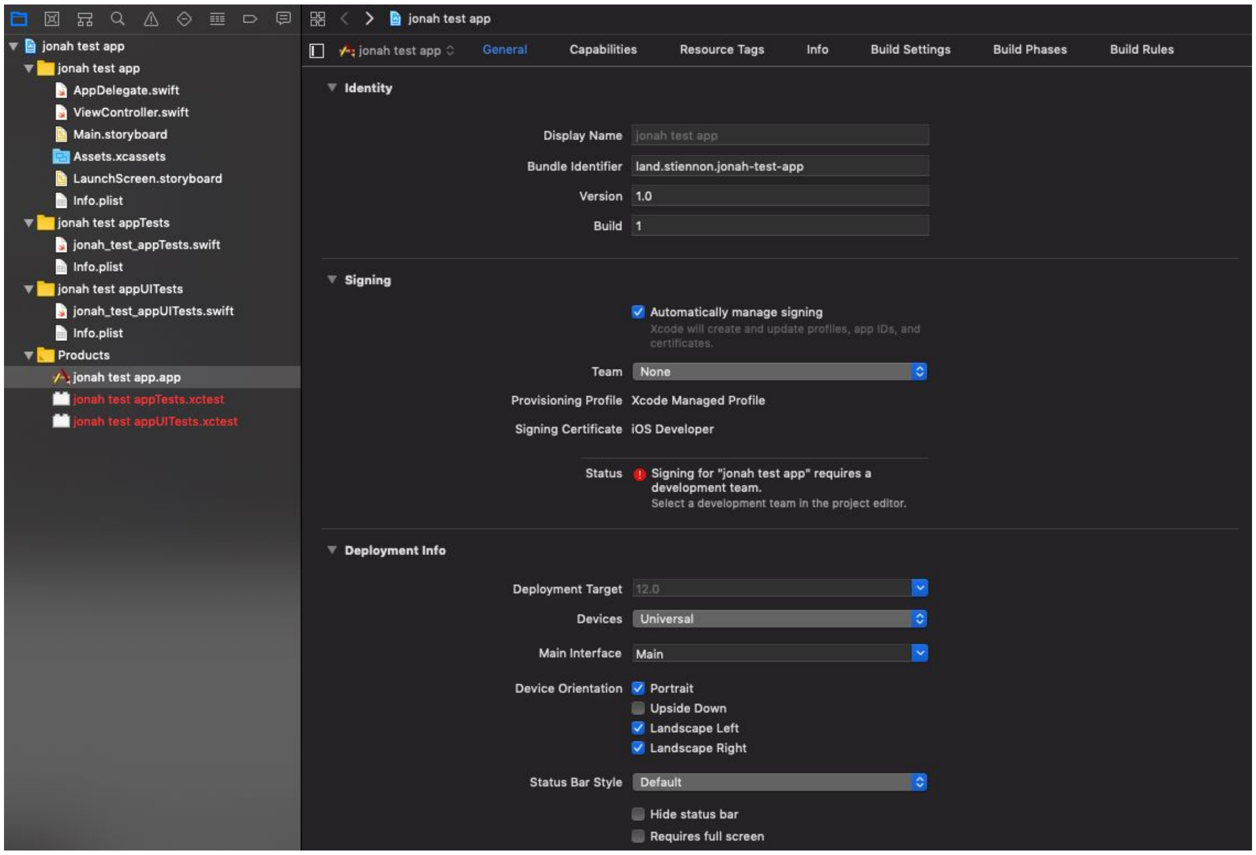
Running tests on the device will always require you to unlock it. There are parts of this process where if the device is locked, things may get into a bad state and the only way to continue will be to unplug the device and plug it back in. In order to avoid any further issues (and annoyances), let's go into the Settings app, select Display & Brightness, select Auto-Lock, and set it to never, to ensure that the screen never locks on us mid-test. (Obviously this means it's a good idea to keep your phone plugged in, and maybe set the brightness down a bit to save some electrons).

Run App Manually Using Xcode

Before using Appium to run tests on our device, let's make sure we can manually run our app on the device using Xcode. Opening the Xcode workspace for the app we want to test, we choose the device we want to run the app on by clicking the icon to the right of our app name by the ▸ button in the upper left hand corner. Xcode will usually automatically select our device once plugged in, but sometimes we have to select it ourselves.



My app is so new that I haven't set up Xcode with my developer profile. If you're like me, Apple will want us to register with them before I go any further. If I click the play button, it tells me I need a provisioning profile. Clicking on my app in the file browser on the left side of Xcode I'm brought to the main settings for the project.



Identity

Display Name: jonah test app
Bundle Identifier: land.stiennon.jonah-test-app
Version: 1.0
Build: 1

Signing

✓ Automatically manage signing
Xcode will create and update profiles, app IDs, and certificates.

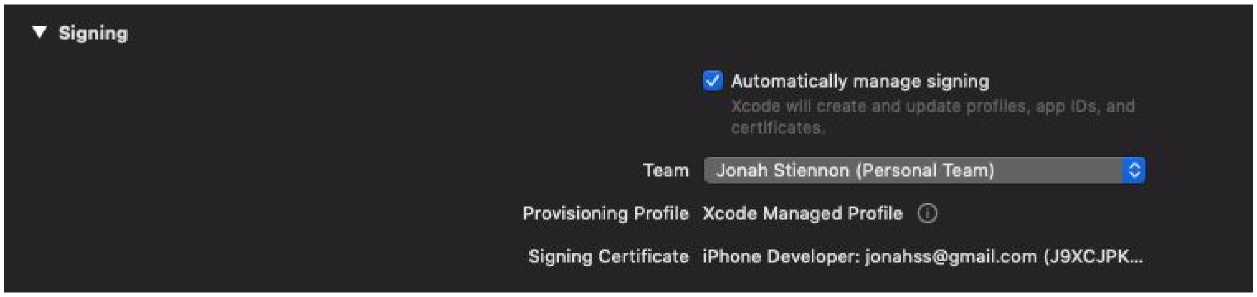
Team: None
Provisioning Profile: Xcode Managed Profile
Signing Certificate: iOS Developer

Status: Signing for "jonah test app" requires a development team.
Select a development team in the project editor.

Deployment Info

Deployment Target: 12.0
Devices: Universal
Main Interface: Main
Device Orientation: ✓ Portrait, Upside Down, ✓ Landscape Left, ✓ Landscape Right
Status Bar Style: Default
Hide status bar
Requires full screen

There's a warning sign in the "signing" section. Select the dropdown next to Team choose Add an Account. If I worked on a team with other people, at this point I'd be asking them to add me to their team which should already be set up to run this app. Sign in with our Apple ID, click on "manage certificates" and click the + button to add a new certificate. Xcode is helpful and handles the rest. Now we can go back to our app settings page and choose our name from the dropdown.

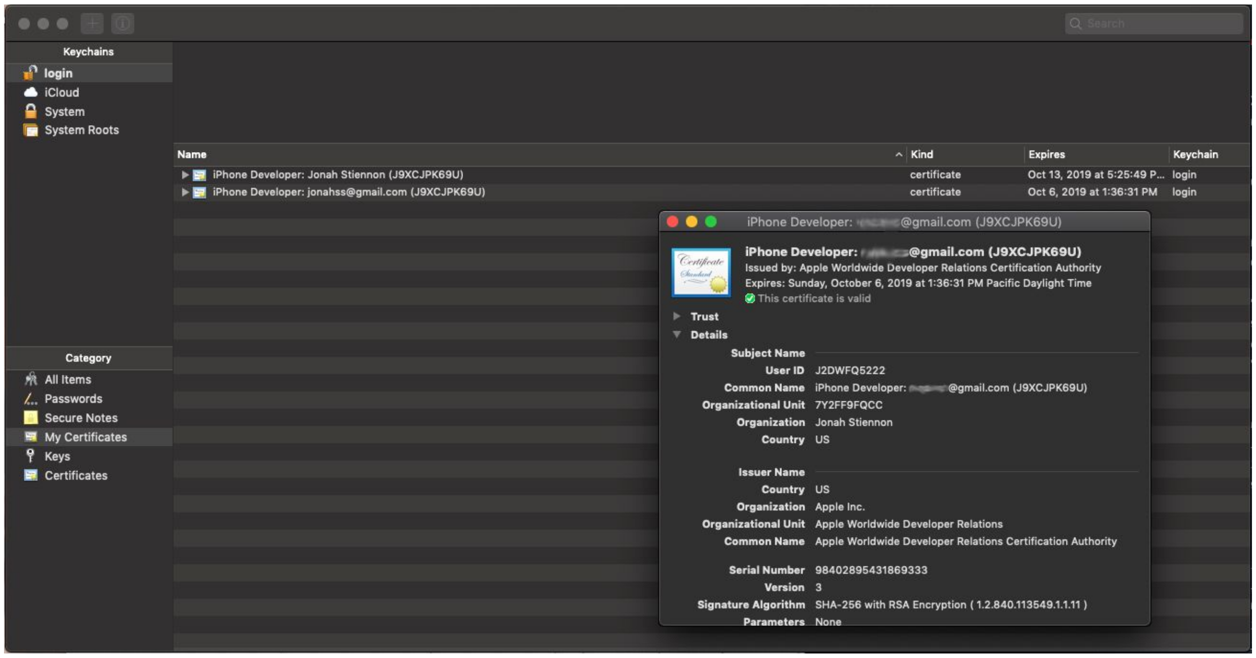


Signing

✓ Automatically manage signing
Xcode will create and update profiles, app IDs, and certificates.

Team: Jonah Stiennon (Personal Team)
Provisioning Profile: Xcode Managed Profile
Signing Certificate: iPhone Developer: jonahss@gmail.com (J9XCJPK...

The certificate we created has an important piece of information we will need later. We need the "Organizational Unit" which is a way to identify our development team. Let's open the Keychain Access app on our Mac and choose My Certificates in the lower lefthand corner. Now the main panel will display our new developer certificate. Double-click on the certificate we just created to see its details. Make a note of the ten digit ID next to Organizational Unit.



Name	Kind	Expires	Keychain
iPhone Developer: Jonah Stiennon (J9XCJPK69U)	certificate	Oct 13, 2019 at 5:25:49 P...	login
iPhone Developer: jonahss@gmail.com (J9XCJPK69U)	certificate	Oct 6, 2019 at 1:36:31 PM	login

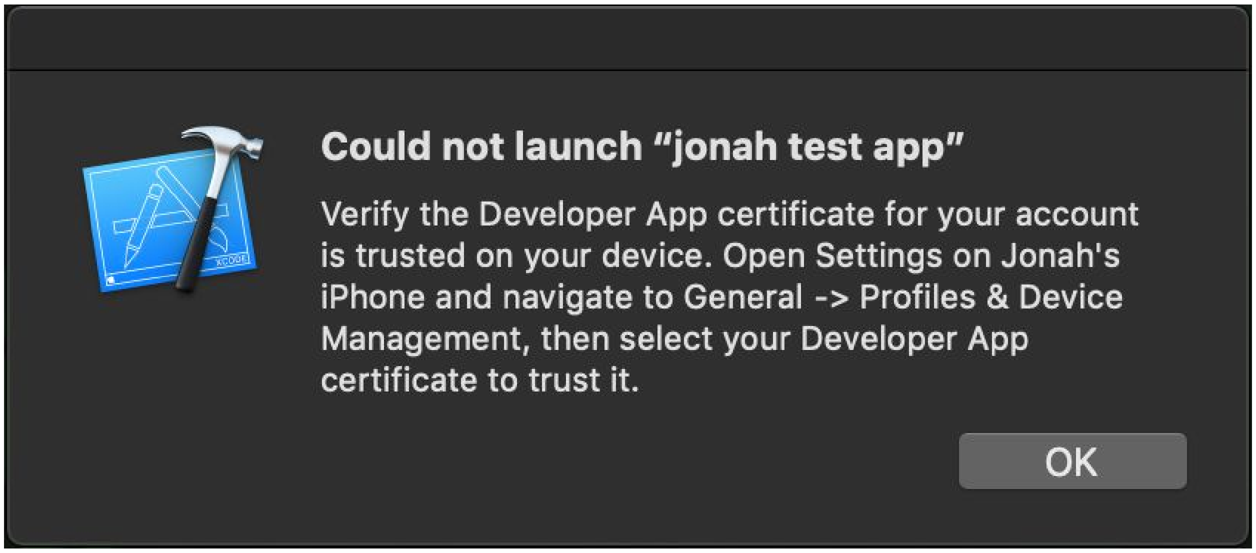
iPhone Developer: [redacted]@gmail.com (J9XCJPK69U)

iPhone Developer: [redacted]@gmail.com (J9XCJPK69U)
Issued by: Apple Worldwide Developer Relations Certification Authority
Expires: Sunday, October 6, 2019 at 1:36:31 PM Pacific Daylight Time
This certificate is valid

▶ Trust
▼ Details

Subject Name	
User ID	J2DWFQ5222
Common Name	iPhone Developer: [redacted]@gmail.com (J9XCJPK69U)
Organizational Unit	7Y2FF9FQCC
Organization	Jonah Stiennon
Country	US
Issuer Name	
Country	US
Organization	Apple Inc.
Organizational Unit	Apple Worldwide Developer Relations
Common Name	Apple Worldwide Developer Relations Certification Authority
Serial Number	98402895431869333
Version	3
Signature Algorithm	SHA-256 with RSA Encryption (1.2.840.113549.1.1.11)
Parameters	None

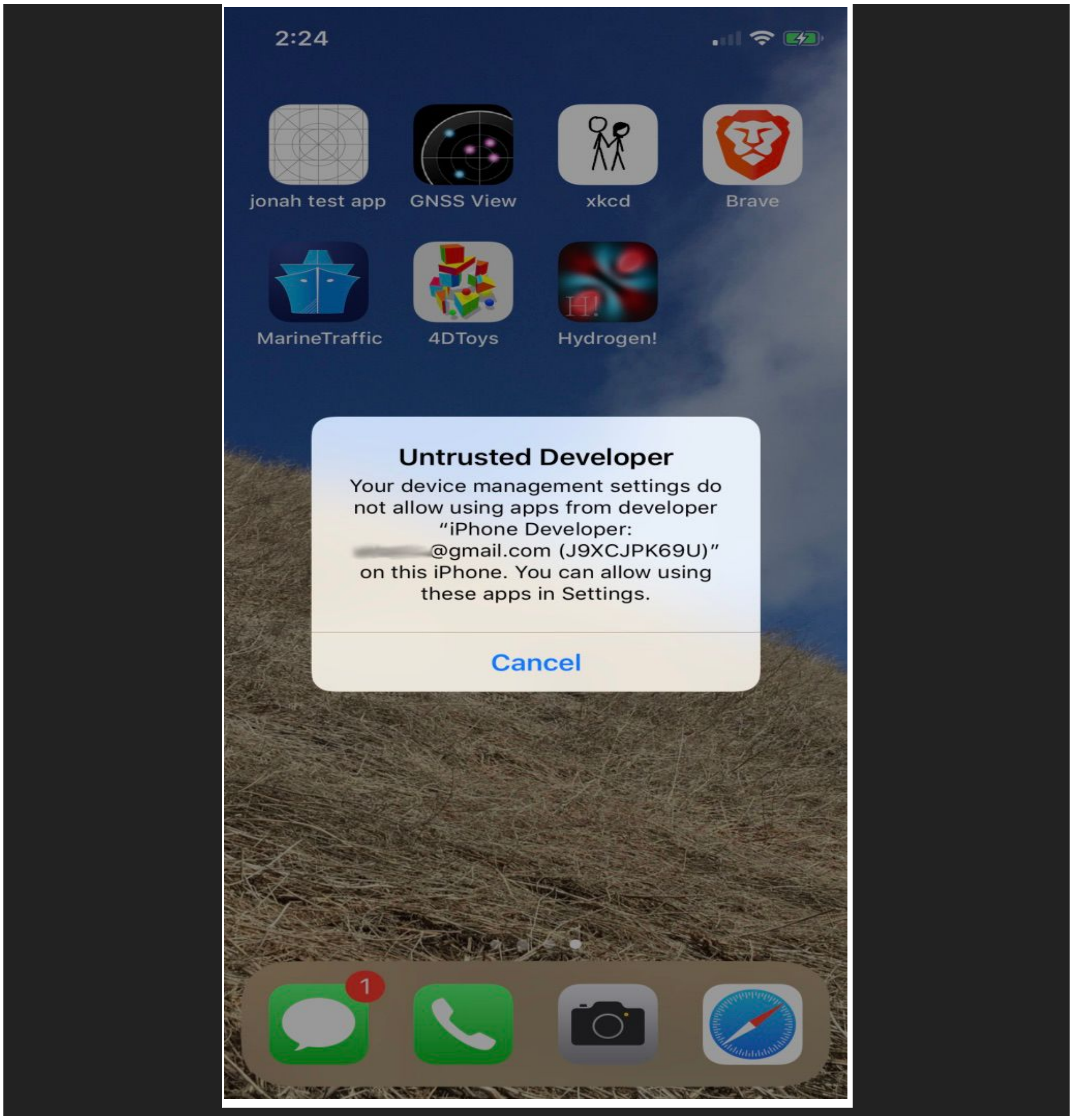
Going back to Xcode, clicking the ▸ button again in the upper lefthand corner will attempt to run our app on the device, but another error is logged.



Could not launch "jonah test app"

Verify the Developer App certificate for your account is trusted on your device. Open Settings on Jonah's iPhone and navigate to General -> Profiles & Device Management, then select your Developer App certificate to trust it.

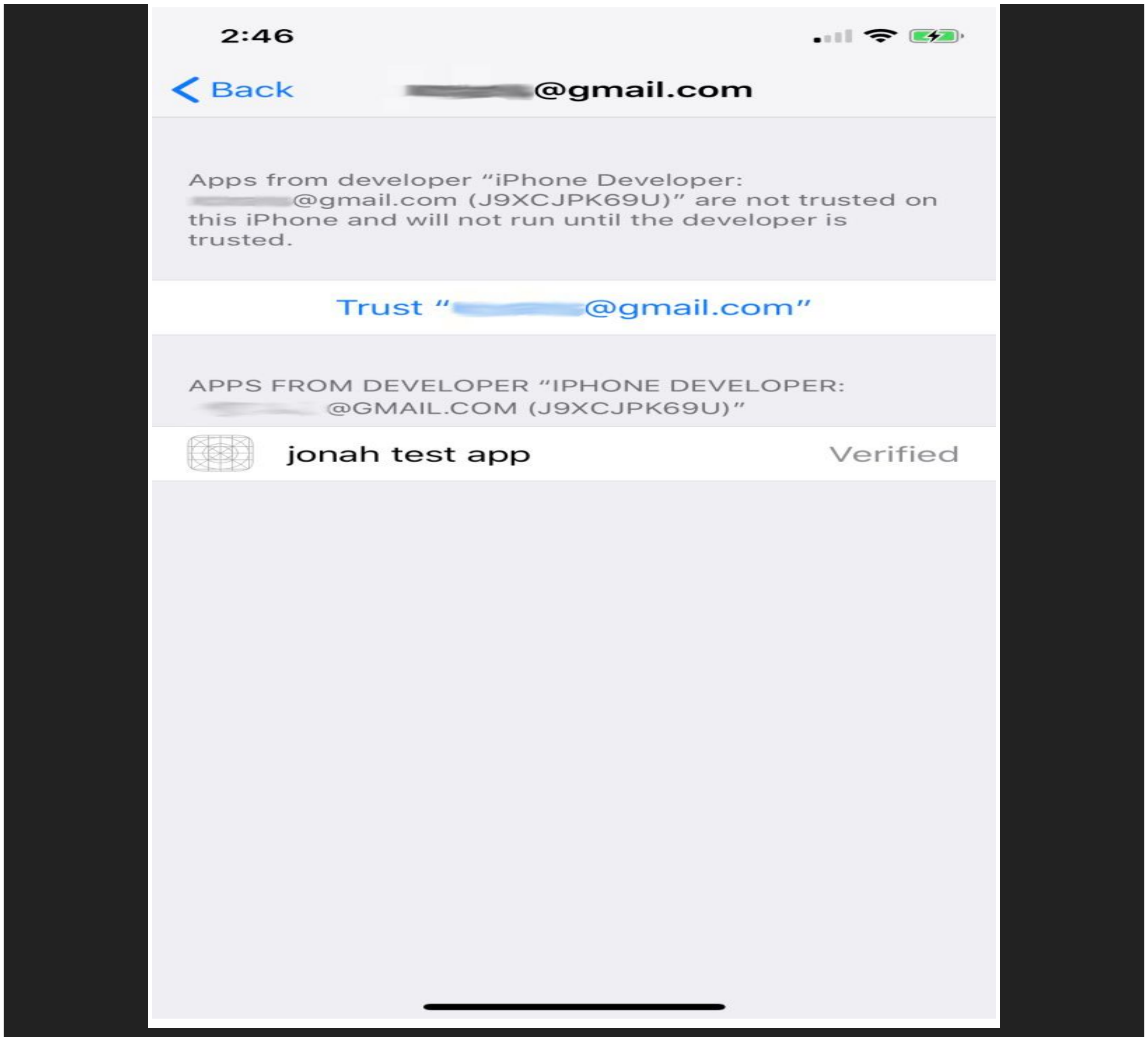
Xcode is telling us that in order for the device to run our app, the device has to trust our developer certificate. Normally, an iOS device will only run apps from the App Store. An app which is placed on the device but didn't come from the app store could be malicious. The device checks the certificate that the app was compiled with and will only trust official app store certificates. If we look at our device, we can see the app has been installed, it is only when the device tries to run the app that we encounter this error. As an interesting experiment, try manually tapping on the app on the device, this time the device (rather than Xcode) tries to instruct us how to trust the certificate.



Untrusted Developer

Your device management settings do not allow using apps from developer "iPhone Developer: [redacted]@gmail.com (J9XCJPK69U)" on this iPhone. You can allow using these apps in Settings.

We want the device to trust our certificate which we added to Xcode in the previous step, so now we will tell the device to trust it. Let's do exactly as it instructed, open the Settings app on the device, tap on General, tap on Device Management and pick the Developer App Certificate which we created earlier.



2:46

< Back [redacted]@gmail.com

Apps from developer "iPhone Developer: [redacted]@gmail.com (J9XCJPK69U)" are not trusted on this iPhone and will not run until the developer is trusted.

Trust "[redacted]@gmail.com"

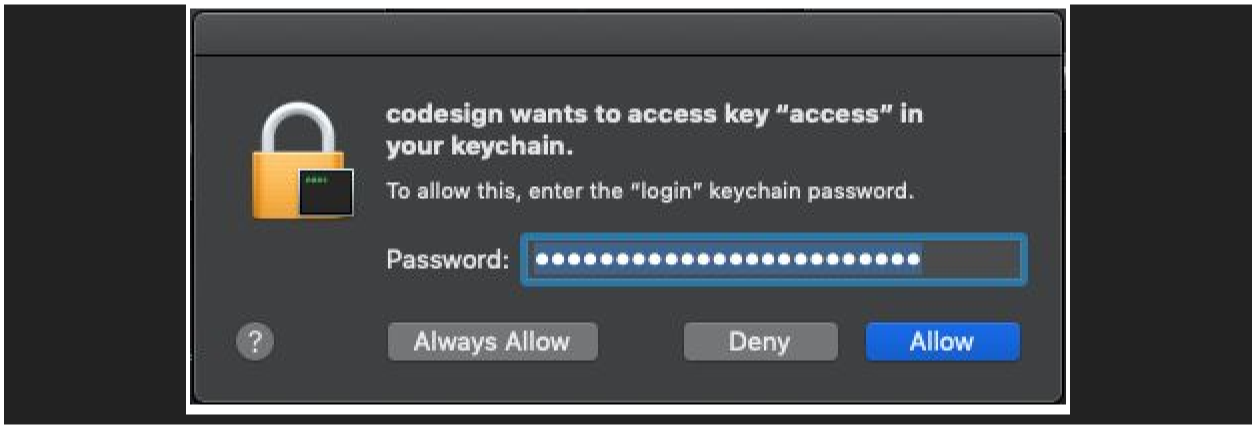
APPS FROM DEVELOPER "IPHONE DEVELOPER: [redacted]@GMAIL.COM (J9XCJPK69U)"

jonah test app Verified

Then tap the blue link to trust the app we're trying to test.

By clicking the ▸ button in Xcode, the app now runs on our device!

There is a rather confusing issue I ran into, where Xcode asks us for our password while building our app. Typing the wrong password will cause the alert to shake disagreeably, but typing the correct password and hitting the return key seems to do nothing.

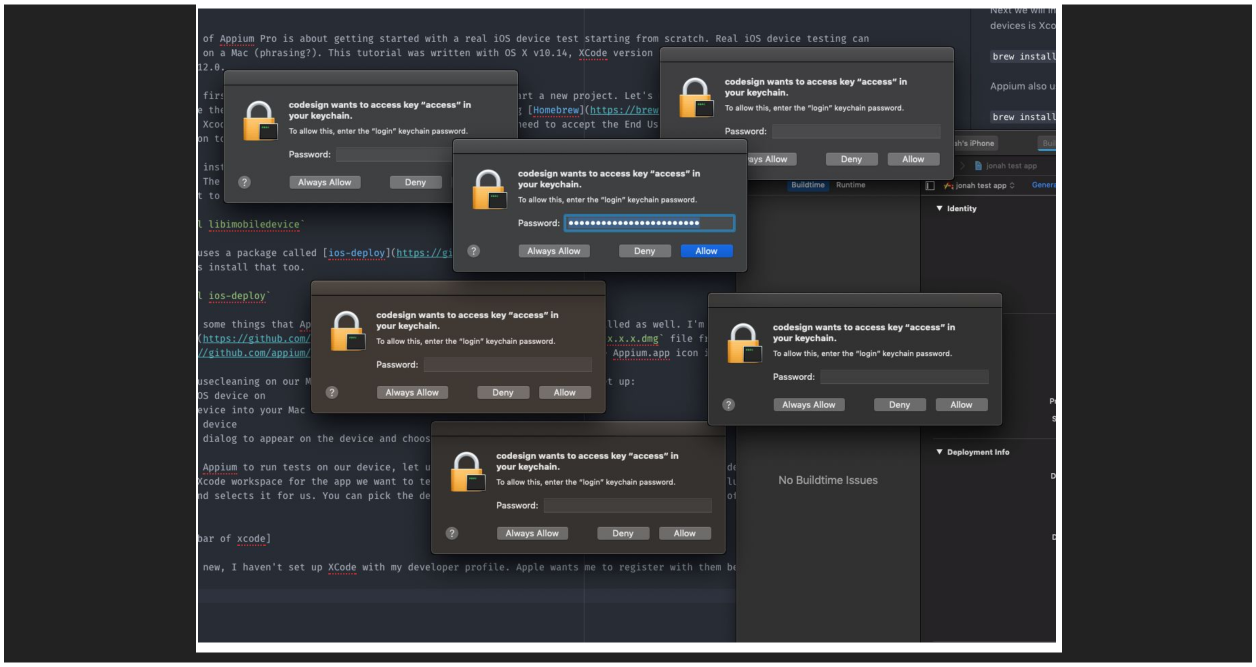


codesign wants to access key "access" in your keychain.

To allow this, enter the "login" keychain password.

Password: ••••••••••••••••••••••

The sneaky issue going on here is that the default button activated by the return key is Allow, but we actually want to click Always Allow. Accidentally choosing Allow results in an identical popup appearing immediately. Dragging the window reveals another window underneath it. I really got myself into a mess this way.



codesign wants to access key "access" in your keychain.

To allow this, enter the "login" keychain password.

Password:

Don't worry, the simple but annoying solution is to patiently type your password and click Always Allow over and over until you work your way through all windows.

At this point, we have:

1. Gotten a fresh set of system dependencies
2. Linked our device with Xcode
3. Linked our developer account with Xcode
4. Linked our device with the developer certificate used to sign our app
5. Launched our app manually on our device

Assuming this has all worked, we will be in good shape for Appium to automate our app

How Appium Automates Real iOS Devices

You don't need to understand how Appium controls iOS devices in order to run and write your tests, but I haven't found a written explanation of this anywhere else, so am including it for those who are curious. To get started testing right away, skip to the next section, but if you're wondering why we've had to install some of the things we need, read on.

User Interface Testing of any device relies upon the ability to launch an app and have another program inspect and interact with what the app displays on the screen. iOS is very strict about security and works hard to prevent one app from looking at what is going on in another app. In order to provide this necessary feature for testing apps, Apple built the XCUITest framework. The way it works is that Xcode has the ability to build a special app called an "XCUITest-Runner" app. The XCUITest-Runner app has access to a special set of system functions which can look at the user interface elements of another app and interact with them. These special functions are called the XCUITest API, and sparse documentation for them can be found here. https://developer.apple.com/documentation/xctest/user_interface_tests (For the rest of this explanation, we'll refer to the app we are trying to test as the "app under test" or AUT, and the XCUITest-Runner app as the "runner app".)

When running tests, Xcode installs both the AUT and the runner app on the device. The runner app is a special package which includes the actual tests we wrote, and the name of the AUT. Xcode tells the device to launch the runner app, and the runner app launches the AUT. From this point forward, both the runner app and the AUT are active on the device. The runner app is invisible and works in the background, while the AUT is displayed on screen.

After the AUT launches, the runner app goes through its list of tests and runs each test, looking at the user interface of the AUT and tapping, swiping, typing into it, etc... It does this using the special XCUITest API functions.

UI tests are usually compiled into the runner app which is loaded onto the device, but Appium needs to be able to control the device as you send commands to it, rather than following a predetermined script. What we need is a test which can become any test, rather than following a prescribed set of user actions. Some bright minds at Facebook came up with a way to do this and published a special test called WebDriverAgent (or WDA).

https://github.com/facebook/WebDriverAgent

Essentially, WDA is a runner app which opens a connection to the outside world and waits for commands to be sent to it, calling the relevant XCUITest API methods for each command. The creators of WDA chose to use the eponymous WebDriver protocol for the format of these commands, the same protocol that Appium already uses for its test commands.

When your Appium tests run, they are using an Appium client to send WebDriver commands to the Appium server. The Appium server installs both your app and WDA on the iOS device, waits for WDA to start, then forwards your test commands to WDA. WDA in turn executes XCUITest API functions on the device, corresponding to the commands it receives from the Appium server. In this way, we are able to arbitrarily interact with the user interface of an iOS device.

Whenever you see a reference to WDA in the Appium logs, this is referring to WebDriverAgent running on the device.

The Capabilities

We have the app running on our device, now let's write a simple automated test which will launch our app and look for a particular set of words on the screen. Because all of our Apple setup has been done (hopefully) correctly, all that we need to do from the Appium side of things is use the right set of capabilities:

```
DesiredCapabilities capabilities = new DesiredCapabilities();

capabilities.setCapability("platformName", "iOS");

capabilities.setCapability("platformVersion", "12.0.1");

capabilities.setCapability("deviceName", "iPhone 8");

capabilities.setCapability("udid", "auto");

capabilities.setCapability("bundleId", "<your bundle id>");

capabilities.setCapability("xcodeOrgId", "<your org id>");

capabilities.setCapability("xcodeSigningId", "iPhone Developer");

capabilities.setCapability("updatedWDABundleId", "<bundle id in scope of
provisioning profile>");
```

The trick here is knowing how to fill all of these out!

1. **platformName** is iOS (as you would no doubt expect)
2. **platformVersion** is the version of iOS our app is running, 12.0.1 in my case.
3. **deviceName** does not actually matter for us, since we have plugged in a real device and will select that device using the udid desired capability. Appium still requires us to supply a value for deviceName, though, so I put iPhone 8.
4. **udid** is the unique ID of the device we want to run our test on. We could find our device udid by running the command

instruments -s devices in the terminal, but since we only have a single device plugged in, we can put auto and Appium will automatically find the udid of the device for us and use it.

5. **bundleId** is the special iOS-internal name of our app, which is set in the same app-settings form where we selected our Team in Xcode. It is a unique way of identifying any app. In my case it is land.stiennon.jonah-test-app, but you should put in a value that is correct for your app..
6. **xcodeOrgId** is the "Organizational Unit" value we made a note of earlier. It is the ID of the Developer Team which signed the certificate used to create the app.
7. **xcodeSigningId** is the first part of the "Common Name" associated with the developer certificate. Since Xcode set this up for us, it is almost always iPhone Developer but could be something different for you if you are automating a different iOS device.
8. **updatedWDABundleId** is used by Appium to trick your device into allowing Appium to install WDA on it. You might have wondered, given how many hoops you had to jump through to get your app running on a device, how Appium is able to get WDA on the device. The short answer is that, typically, it can't. WDA's bundle ID (com.facebook.webdriveragent) will not show up as an App ID in any of your provisioning profiles, so the app would not be allowed to run on your device. But given that Appium has the WDA code, it can actually change the bundle ID on

the fly, so that when WDA is built and signed it will be allowed past Apple's security restrictions. What this means is that you must supply a bundle ID value that is allowed by an App ID in your provisioning profile. We typically recommend using wildcard App IDs (like com.test.*), so that we can give a new bundle ID to WDA of com.test.webdriveragent. You could also give it the same bundle ID as your app, but that could cause some confusion in the system later on. If you prefer, you can omit this capability and simply open up the WDA project inside of Appium, and make all these modifications yourself using Xcode.

The Test

Your actual test code, of course, is up to you to define! You'll simply instantiate a Driver and run your test as in any other case. All the heavy lifting is done by Appium in response to the capabilities above and in the context of correctly-signed apps and correctly-provisioned devices. Here's a step-by-step guide of what to do:

1. Start the Appium server (by opening the Appium Desktop app or using the CLI).
2. Run the Java test you wrote including the capabilities above, and watch as the iOS device opens your app!

3. Make sure the device is unlocked, and if it asks you to "Trust the Computer", tap the button to trust our Mac. You'll need to do this the first time.
4. If anything goes wrong, check the logs which Appium prints.

Note that the **bundleId** capability can only be used for apps which are already installed on our iOS device. We installed our app manually using Xcode, so the app is already there. If we make changes to the app code, we will have to click the ▶ button in Xcode in order to install the latest version of our code on the device. Then we can run our Appium tests again.

Alternatively, we could use the app capability and set it to the path of an .ipa file on disk. This must be an app archive generated in Xcode and signed correctly.

SAMPLE CODE

```
import io.appium.java_client.ios.IOSDriver;

import java.net.MalformedURLException;

import java.net.URL;

import org.junit.After;

import org.junit.Before;

import org.junit.Test;

import org.openqa.selenium.remote.DesiredCapabilities;


public class Edition041_iOS_Real_Device {

    private IOSDriver driver;


    @Before

    public void setUp() throws MalformedURLException {

        DesiredCapabilities capabilities = new DesiredCapabilities();

        capabilities.setCapability("platformName", "iOS");

        capabilities.setCapability("platformVersion", "12.0.1");

        capabilities.setCapability("deviceName", "iPhone 8");

        capabilities.setCapability("udid", "auto");

        capabilities.setCapability("bundleId", "<your bundle id>");

        capabilities.setCapability("xcodeOrgId", "<your org id>");

        capabilities.setCapability("xcodeSigningId", "iPhone Developer");

        capabilities.setCapability("updatedWDABundleId", "<bundle id in scope of
provisioning profile>");
```

```
        driver = new IOSDriver<>(new URL("http://localhost:4723/wd/hub"), capabilities);
    }

    @After
    public void tearDown() {
        if (driver != null) {
            driver.quit();
        }
    }

    @Test
    public void testFindingAnElement() {
        driver.findElementByAccessibilityId("Login Screen");
    }
}
```