Getting Started with Appium

by Dave Haeffner

Preface

This getting started guide would not exist (or be nearly as good) if it weren't for the help of Matthew Edwards -- a leading Appium contributor.

He was looking for a way to make the getting started experience for Appium more approachable. What you have in front of you is the end result.

The examples in this guide were based on the exisitng Appium tutorials. I took some creative liberties, and Matthew helped keep me honest. While the prose are mine, this was a collaborative work between Matthew and myself.

We hope you enjoy it!

Cheers,
Dave H

Table of Contents

Before You Get Started

Appium is built to test mobile apps of all kinds (native, hybrid, and mobile web), has client libraries written in every popular programming language, it's open source, works on every prominent operating system, and best of all -- it works for iOS and Android.

But before you jump in with both feet, know that there is a bit of setup in order to get things up and running on your local machine.

A brief Appium primer

Appium is architected similarly to Selenium -- there is a server which receives commands and executes them on a desired node. But instead of a desktop browser, the desired node is running a mobile app on a mobile device (which can be either a simulator, an emulator, or a physical device).

In order for Appium to work, we will need to install the dependent libraries for each device we care about.

Initial Setup

Testing an iOS app? Here's what you'll need:

- Install Java (version 7 of the JDK or higher)
- Install Xcode
- Install Xcode Command-line Build Tools

For more info on supported Xcode versions, read this.

Testing an Android app? Here's what you'll need:

- Install Java (version 7 of the JDK or higher)
- Install the Android SDK (version 17 or higher)
- Install the necessary packages for your Android platform version in the Android SDK Manager
- Configure an Android Virtual Device (AVD) that mimics the device you want to test against

For more info on setting up the Android SDK and configuring an AVD, read this.

Next, you'll need to install Appium. Luckily, there's a handy binary for it (Appium.app for OSX and Appium.exe for Windows). This binary also happens to be a GUI wrapper for Appium.

Alternatively, if you want the absolute latest version of Appium and aren't afraid to get your hands dirty, then you can install Appium from source and run it from the command line.

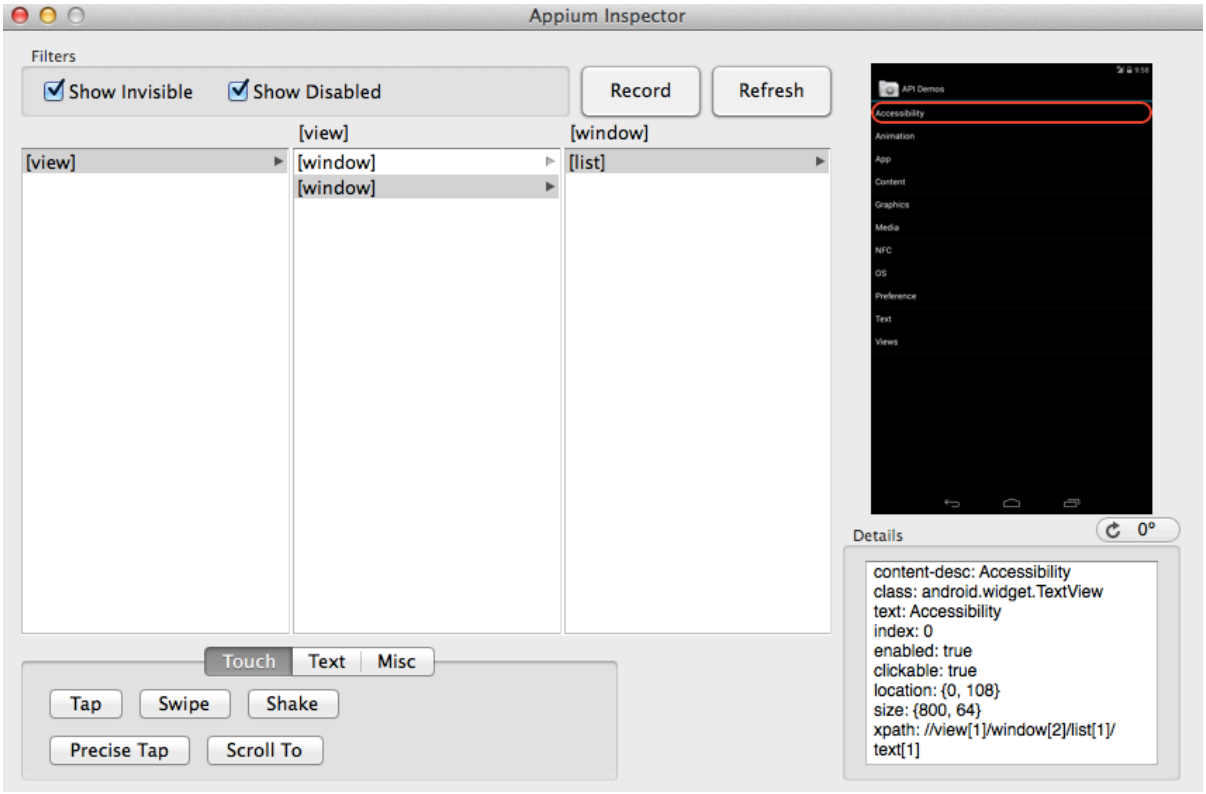
But if you're new to mobile testing, then the one-click installer is a better place to start.

An Appium GUI primer

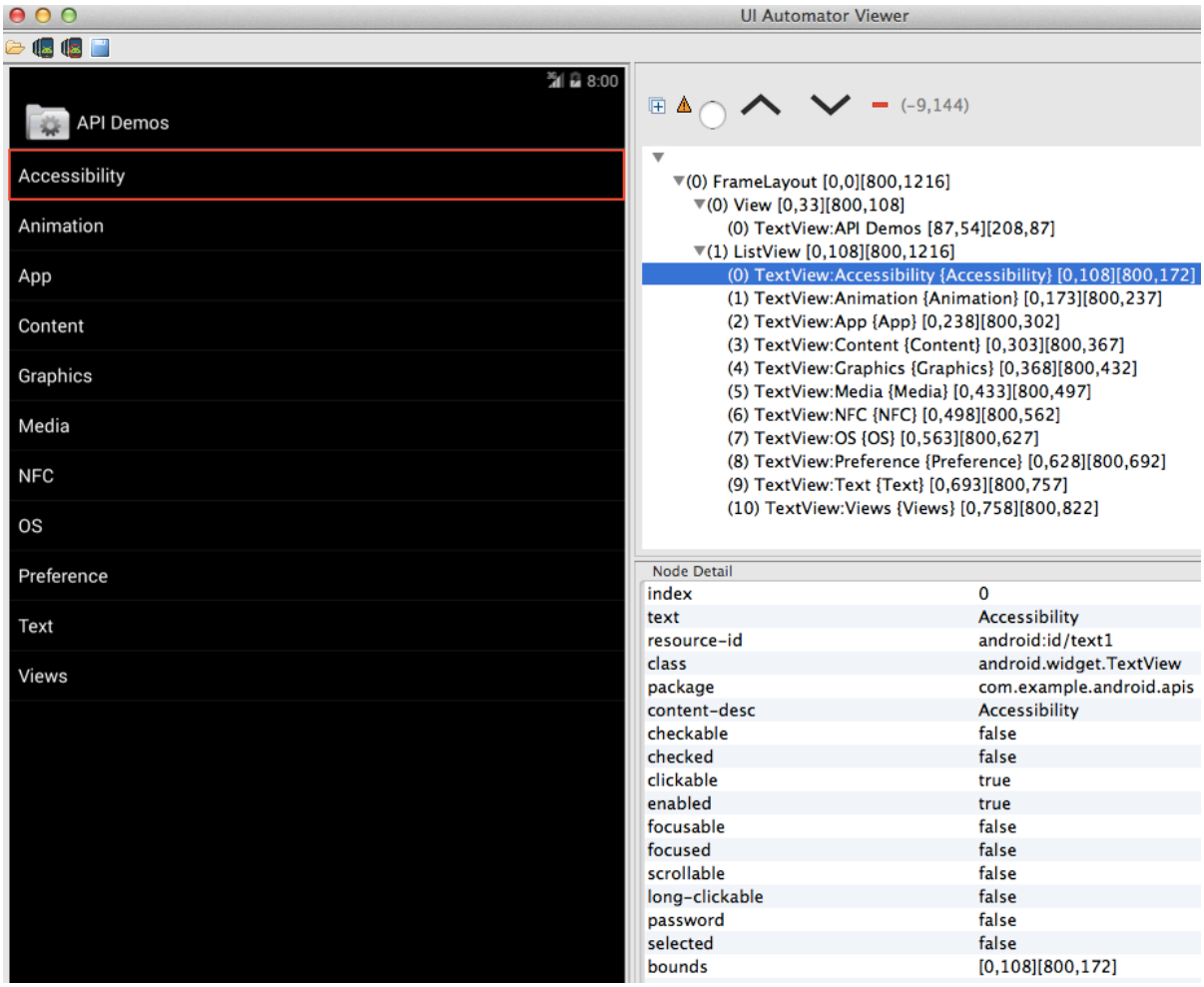
The Appium GUI is a one-click installer for the Appium server that enables easy configuration of your app and Appium.

Aside from the easy install, it adds a key piece of functionality -- an inspector. The inspector enables a host of functionality, most notably:

- shows you all of the elements in your app
- enables you to record and playback user actions



While the inspector works well for iOS, there are some problem areas with it in Android on Appium at the moment. To that end, the Appium team encourages the use of uiautomatorviewer (which is an inspector tool provided by Google that provides similar functionality to the Appium inspector tool). For more info on how to set that up, read this.



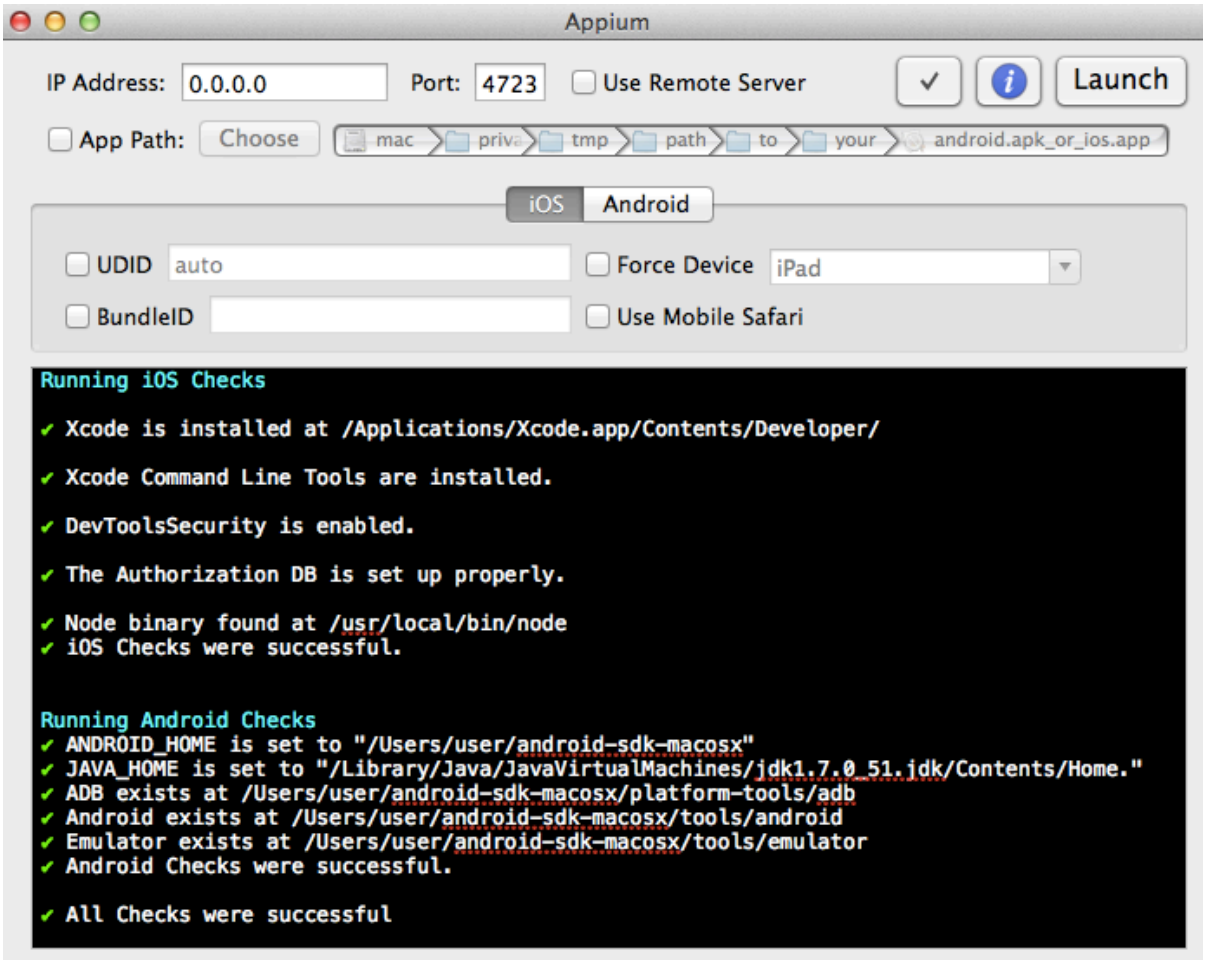
More on inspectors and how to use them in a later post.

It's worth noting that while we can configure our app within the Appium GUI, it's not necessary since we will be able to do it more flexibly in code. More on that in the next post.

Making Sure Appium Is Setup

After you have your Appium one-click installer up and running, you can verify your setup by using it's Doctor functionality. It is the button on the left of the `Launch` button. It is the one that looks like a doctor's stethoscope.

Click on that, and it should output information in the center console window of the Appium GUI.



If you don't see anything outputted, refer to this open issue.

What About A Programming Language?

Before you do a victory lap, you'll also want to have chosen a programming language to write your tests in, installed said programming language, and installed it's client bindings for Appium.

Currently, Appium has client bindings for Java, JavaScript, Objective C, .NET, PHP, Python, and Ruby.

The examples in this series will be written in Ruby. You can use version 1.9.3 or later, but it's advisable to use the latest stable version. For instructions on installing Ruby and the necessary client libraries (a.k.a. "gems"), read this.

Outro

Now that you have Appium setup with all of it's requisite dependencies, along with a programming language and Appium client bindings, we're ready to load up a test app and step through it.

Configuring Appium

In order to get Appium up and running there are a few additional things we'll need to take care of.

If you haven't already done so, install Ruby and setup the necessary Appium client libraries (a.k.a. "gems"). You can read a write-up on how to do that here.

Installing Necessary Libraries

Assuming you've already installed Ruby and need some extra help installing the gems, here's what you to do.

1. Install the gems from the command-line with `gem install appium_console`
2. Once it completes, run `gem list | grep appium`

You should see the following listed (your version numbers may vary):

```
appium_console (1.0.1)
appium_lib (4.0.0)
```

Now you have all of the necessary gems installed on your system to follow along.

An Appium Gems Primer

`appium_lib` is the gem for the Appium Ruby client bindings. It is what we'll use to write and run our tests against Appium. It was installed as a dependency to `appium_console`.

`appium_console` is where we'll focus most of our attention in the remainder of this and the next post. It is an interactive prompt that enables us to send commands to Appium in real-time and receive a response. This is also known as a record-eval-print loop (REPL).

Now that we have our libraries setup, we'll want to grab a copy of our app to test against.

Sample Apps

Don't have a test app? Don't sweat it. There are pre-compiled test apps available to kick the tires with. You can grab the iOS app here and the Android app here. If you're using the iOS app, you'll want to make sure to unzip the file before using it with Appium.

If you want the latest and greatest version of the app, you can compile it from source. You can find instructions on how to do that for iOS here and Android here.

Just make sure to put your test app in a known location, because you'll need to reference the path to it next.

App Configuration

When it comes to configuring your app to run on Appium there are a lot of similarities to Selenium -- namelythe use of Capabilities (e.g., "caps" for short).

You can specify the necessary configurations of your app through caps by storing them in a file called `appium.txt`.

Here's what `appium.txt` looks like for the iOS test app to run in an iPhone simulator:

```
[caps]
platformName = "ios"
app = "/path/to/UICatalog.app.zip"
deviceName = "iPhone Simulator"
```

And here's what `appium.txt` looks like for Android:

```
[caps]
platformName = "android"
app = "/path/to/api.apk"
deviceName = "Android"
avd = "training"
```

For Android, note the use of both `avd`. The `"training"` value is for the Android Virtual Device that we configured in the previous post. This is necessary for Appium to auto-launch the emulator and connect to it. This type of configuration is not necessary for iOS.

For a full list of available caps, read this.

Go ahead and create an appium.txt with the caps for your app (making sure to place it in the same directory as the Gemfile we created earlier).

Launching The Console

Now that we have a test app on our system and configured it to run in Appium, let's fire up the Appium Console.

First we'll need to start the Appium server. So let's head over to the Appium GUI and launch it. It doesn't matter which radio button is selected (e.g., Android or Apple). Just click the `Launch` button

in the top right-hand corner of the window. After clicking it, you should see some debug information in the center console. Assuming there are no errors or exceptions, it should be up ready to receive a session.

After that, go back to your terminal window and run `arc` (from the same directory as `appium.txt`). This is the execution command for the Appium Ruby Console. It will take the caps from `appium.txt` and launch the app by connecting it to the Appium server. When it's done you will have an emulator window of your app that you can interact with as well as an interactive command-prompt for Appium.

Outro

Now that we have our test app up and running, it's time to interrogate our app and learn how to interact with it.

Interrogating Your App

Writing automated scripts to drive an app in Appium is very similar to how it's done in Selenium. We first need to choose a locator, and use it to find an element. We can then perform an action against that element.

In Appium, there are two approaches to interrogate an app to find the best locators to work with. Through the Appium Console, or through an Inspector (e.g., Appium Inspector, uiautomatorviewer, or selendroid inspector).

Let's step through how to use each of them to decompose and understand your app.

Using the Appium Console

Assuming you've followed along with the last two posts, you should have everything setup and ready to run.

Go ahead and startup your Appium server (by clicking `Launch` in the Appium GUI) and start the Appium Ruby Console (by running `arc` in a terminal window that is in the same directory as your `appium.txt` file). After it loads you will see an emulator window of your app that you can interact with as well as an interactive prompt for issuing commands to Appium.

The interactive prompt is where we'll want to focus. It offers a host of readily available commands to quickly give us insight into the elements that make up the user interface of the app. This will help us easily identify the correct locators to automate our test actions against.

The first command you'll want to know about is `page`. It gives you access to every element in the app. If you run it by itself, it will output all of the elements in the app, which can be a bit unwieldy. Alternatively you can specify additional arguments along with it. This will filter the output down to just a subset of elements. From there, there is more information available that you can use to further refine your results.

Let's step through some examples of that and more for both iOS and Android.

An iOS Example

To get a quick birds eye view of our iOS app structure, let's get a list of the various element classes available. With the `page_class` command we can do just that.

```
[1] pry(main)> page_class
get /source
13x UIAStaticText
12x UIATableCell
4x UIAElement
2x UIAWindow
1x UIATableView
1x UIANavigationBar
1x UIAStatusBar
1x UIAApplication
```

`UIAStaticText` and all of the others are the specific class names for types of elements in iOS. You can see reference documentation for `UIAStaticText` here. If you want to see the others, go here.

With the `page` command we can specify a class name and see all of the elements for that type. When specifying the element class name, we can either specify it as a string, or a symbol (e.g., `'UIAStaticText'` or `:UIAStaticText`).

```
[2] pry(main)> page :UIAStaticText
get /context
post /execute
{
    :script => "UIATarget.localTarget().frontMostApp().windows()[0].getTree()"
}
UIAStaticText
   name, label, value: UICatalog
UIAStaticText
   name, label: Buttons, Various uses of UIButton
   id: ButtonsTitle   => Buttons
       ButtonsExplain => Various uses of UIButton
UIAStaticText
   name, label: Controls, Various uses of UIControl
   id: ControlsExplain => Various uses of UIControl
       ControlsTitle   => Controls
UIAStaticText
   name, label: TextFields, Uses of UITextField
   id: TextFieldExplain => Uses of UITextField
       TextFieldTitle   => TextFields
...
```

Note the `get` and `post` (just after we issue the command but before the element list). It is the network traffic that is happening behind the scenes to get us this information from Appium. The response to `post /execute` has a script string. In it we can see which window this element lives in (e.g., `windows()[0]`).

This is important because iOS has the concept of windows, and some elements may not appear in the console output even if they're visible to the user in the app. In that case, you could list the elements in other pages (e.g., `page window: 1`). `0` is generally the elements for your app, whereas `1` is where the system UI lives. This will come in handy when dealing with alerts.

Finding Elements

Within each element of the list, notice their properties -- things like `name`, `label`, `value`, and `id`. This is the kind of information we will want to reference in order interact with the app.

Let's take the first element for example.

```
UIAStaticText
   name, label, value: UICatalog
```

In order to find this element and interact with it, we can can search for it with a couple of different commands: `find`, `text`, or `text_exact`.

```
> find('UICatalog')
...
#<Selenium::WebDriver::Element:0x..fbe2eadf843018080 id="0">
```

```
> text('UICatalog')
...
#<Selenium::WebDriver::Element:0x..fd4a475024c1bbfe2 id="1">
```

```
> text_exact('UICatalog')
...
#<Selenium::WebDriver::Element:0x56d7cdb94d0de4a4 id="2">
```

We'll know that we successfully found an element when we see a `Selenium::WebDriver::Element` object returned.

It's worth noting that in the underlying gem that enables this REPL functionality, if we end our command with a semi-colon it will not show us the return object.

```
> find('UICatalog')
# displays returned value

> find('UICatalog');
# returned value not displayed
```

To verify that we have the element we expect, let's access the name attribute for it.

```
> find('UICatalog').name
...
"UICatalog"
```

Finding Elements by ID

A better approach to find an element would be to reference its id, since it is less likely to change than the text of the element.

```
UIAStaticText
   name, label: Buttons, Various uses of UIButton
   id: ButtonsTitle   => Buttons
       ButtonsExplain => Various uses of UIButton
```

On this element, there are some IDs we can reference. To find it using these IDs we can use the `id` command. And to confirm that it's the element we expect, we can ask it for its name attribute.

```
> id('ButtonsTitle').name
...
"Buttons, Various uses of UIButton"
```

For a more thorough walk through and explanation of these commands (and some additional ones) go here. For a full list of available commands go here.

An Android Example

To get a quick birds eye view of our Android app structure, let's get a list of the various element classes available. With the `page_class` command we can do just that.

```
[1] pry(main)> page_class
get /source
12x android.widget.TextView
1x android.view.View
1x android.widget.ListView
1x android.widget.FrameLayout
1x hierarchy
```

`android.widget.TextView` and all of the others are the specific class names for types of elements in Android. You can see reference documentation for `TextView` here. If you want to see the others, simply do a Google search for the full class name.

With the `page` command we can specify a class name and see all of the elements for that type. When specifying the element class name, we can specify it as a string (e.g.,

`'android.widget.TextView'`).

```
[2] pry(main)> page 'android.widget.TextView'
get /source
post /appium/app/strings

android.widget.TextView (0)
  text: API Demos
  id: android:id/action_bar_title
  strings.xml: activity_sample_code

android.widget.TextView (1)
  text, desc: Accessibility
  id: android:id/text1

android.widget.TextView (2)
  text, desc: Animation
  id: android:id/text1
...
```

Note the `get` and `post` (just after we issue the command but before the element list). It is the network traffic that is happening behind the scenes to get us this information from Appium. `get /source` is to download the source code for the current view and `post /appium/app/strings` gets the app's strings. These app strings will come in handy soon, since they will be used for some of the IDs on our app's elements; which will help us locate them more easily.

Finding Elements

Within each element of the list, notice their properties -- things like `text` and `id`. This is the kind of information we will want to reference in order interact with the app.

Let's take the first element for example.

```
android.widget.TextView (0)
  text: API Demos
  id: android:id/action_bar_title
  strings.xml: activity_sample_code
```

In order to find that element and interact with it, we can search for it by text or by id.

```
> text('API Demos')
...
#<Selenium::WebDriver::Element:0x..fdfd4a900132140f4 id="2">
```

```
> id('android:id/action_bar_title')
...
#<Selenium::WebDriver::Element:0x..f8ac6f9543fdba436 id="6">
```

We'll know that we successfully found an element when we see a `Selenium::WebDriver::Element` object returned.

It's worth noting that in the underlying gem that enables this REPL functionality, if we end our command with a semi-colon it will not show us the return object.

```
> text('API Demos')
# displays returned value

> text('API Demos');
# returned value not displayed
```

To verify we've found the element we expect, let's access the name attribute for it.

```
> text('API Demos').name
...
"API Demos"
```

Finding Elements by ID

A better approach to find an element would be to reference its ID, since it is less likely to change than the text of the element.

In Android, there are a two types of IDs you can search with -- a resource ID, and strings.xml. Resource IDs are best. But strings.xml are a good runner-up.

```
android.widget.TextView (10)
  text, desc: Text
  id: android:id/text1
  strings.xml: autocomplete_3_button_7
```

This element has one of each. Let's search using each with the `id` command.

```
# resource ID
> id('android:id/text1')
...
#<Selenium::WebDriver::Element:0x..fdfb437dde1faa7c2 id="25">

# strings.xml
> id('autocomplete_3_button_7')
...
#<Selenium::WebDriver::Element:0x..f9dc56786e74b5d48 id="26">
```

You can see a more thorough walk through of these commands here. For a full list of available commands go here.

Ending the session

In order to end the console session, input the `x` command. This will cleanly quit things for you. If a session is not ended properly, then Appium will think it's still in progress and block all future sessions from working. If that happens, then you need to restart the Appium server by clicking `Stop` and then `Launch` in the Appium GUI.

`x` only works within the console. In our test scripts, we will use `driver.quit` to kill the session.

Using An Inspector

With the Appium Ruby Console up and running, we also have access to the Appium Inspector. This is another great way to interrogate our app to find locators. Simply click the magnifying glass in the top-right hand corner of the Appium GUI (next to the `Launch` button) to open it. It will load in a new window.

Once it opens, you should see panes listing the elements in your app. Click on an item in the left-most pane to drill down into the elements within it. When you do, you should see the screenshot on the right-hand side of the window auto-update with a red highlight around the newly targeted element.

You can keep doing this until you find the specific element you want to target. The properties of the element will be outputted in the `Details` box on the bottom right-hand corner of the window.

It's worth noting that while the inspector works well for iOS, there are some problem areas with it in Android at the moment. To that end, the Appium team encourages the use of uiautomatorviewer (which is an inspector tool provided by Google that provides similar functionality to the Appium inspector tool). For more info on how to set that up, read this.

For older Android devices and apps with webviews, you can use the selendroid inspector. For more information on, go here.

There's loads more functionality available in the inspector, but it's outside the scope of this post. For more info I encourage you to play around with it and see what you can find out for yourself.

Outro

Now that we know how to locate elements in our app, we are ready to learn about automating some simple actions and putting them to use in our first test.

Your First Test

There are a good deal of similarities between Selenium and Appium tests. We will be using similar actions (like `click`) along with some kind of wait mechanism (e.g., an explicit wait) to make our tests more resilient. There will also be an assertion used to determine if our actions were successful or not.

In order to put these concepts to work, let's consider the basic structure of the test apps we've been working with. They are straightforward in that they both have text elements that, when clicked, take you to a dedicated page for that element (e.g., Accessibility triggers the Accessibility page). Let's step through our first set of test actions (in the console) that we'll use to automate this behavior; verifying that each element brings us to the correct page.

Let's dig in with some examples.

An iOS Example

The behavior of our app can be easily mapped to test actions by first using a text match to find the element we want, and clicking it. We can then make sure we are in the right place by performing another text match (this time an exact text match). When we wire this up to our test framework, this match will be responsible for passing or failing the test. More on that in the next post.

```
text('Buttons, Various uses of UIButton').click
text_exact 'Buttons'
```

The only problem with this approach is that it is not resilient. The global wait for each test action (a.k.a. an implicit wait) is set to 0 seconds by default. So if there is any delay in the app, the test action will not complete and throw an element not found exception instead.

To overcome these timing problems we can employ an explicit wait around our test actions (both the click and the exact text match). This is simple enough to do with the `wait` command.

```
wait { text('Buttons, Various uses of UIButton').click }
wait { text_exact 'Buttons' }
```

These test actions are resilient now, but they're inflexible since we were using statically coded values. Let's fix that by using dynamic values instead.

```
cell_1 = wait { text 2 }
cell_title = cell_1.name.split(',').first

wait { cell_1.click }
wait { text_exact cell_title }
```

Now we're finding the first text by it's index. Index `2` contains the first element (a.k.a. a cell), whereas index `1` is the table header. After that, we're extracting the name and dynamically finding the title. Now our test will continue to work if there are any text changes.

This is good, but now let's expand things to cover the rest of the app.

```
cell_names = tags('UIATableCell').map { |cell| cell.name }

cell_names.each do |name|
  wait { text_exact(name).click }
  wait { text_exact name.split(',').first }
  wait { back }
end
```

We first grab the names of each clickable cell, storing them in a collection. We then iterate through the collection, finding each element by name, clicking it, performing an exact match on the resulting page, and then going back to the main screen. This is repeated until each cell is verified.

This works for cells that are off the screen (e.g., out of view) since Appium will scroll them into view before taking an action against them.

An Android Example

Things are pretty similar to the iOS example. We perform a text match, click action, and exact text match.

```
text('Accessibility').click
text_exact 'Accessibility Node Provider'
```

We then make things resilient by wrapping them in an explicit wait.

```
wait { text('Accessibility').click }
wait { text_exact 'Accessibility Node Provider' }
```

We then make our selection more flexible by upgrading to dynamic values.

```
cell_1 = wait { text 2 }

wait { cell_1.click }
wait { find_exact 'Accessibility Node Provider' }
```

We then expand things to exercise the whole app by collecting all of the clickable elements and iterating through them.

```
cell_names = tags('android.widget.TextView').map { |cell| cell.name }

cell_names[1..-1].each do |cell_name|
  wait { scroll_to_exact(cell_name).click }
  wait_true { ! exists { find_exact cell_name } }
  wait { back }
  wait { find_exact('Accessibility'); find_exact('Animation')  }
end
```

A few things to note.

The first item in the `cell_names` collection is a header. To discard it, we use `cell_name[1..-1]` which basically says start with the second item in the collection (e.g., `[1`) and continue (e.g., `..`) all the way until the end (e.g., `-1]`).

In order to interact with cells that are off the screen, we will need to use the `scroll_to_exact` command, and perform a `click` against that (instead of a text match).

Since each sub-screen doesn't have many unique attributes for us to verify against, we can at the very least verify that we're no longer on the home screen. After that, we verify that we are brought back to the home screen.

Outro

Now that we have our test actions sussed out, we're ready to commit them to code and plug them into a test runner.

Writing and Refactoring Your Tests

Now that we've identified some test actions in our apps, let's put them to work by wiring them up in code.

We'll start with the iOS app and then move onto Android. But first, we'll need to do a quick bit of setup.

Quick Setup

Since we're setting up our test code from scratch, we'll need to make sure we have the necessary gems installed -- and done so in a way that is repeatable (which will come in handy for other team members and for use with Continuous Integration).

In Ruby, this is easy to do with Bundler. With it you can specify a list of gems and their versions to install and update from for your project.

Install Bundler by running `gem install bundler` from the command-line and then create a file called `Gemfile` with the following contents:

```
# filename: Gemfile

source 'https://rubygems.org'

gem 'rspec', '~> 3.0.0'
gem 'appium_lib', '~> 4.0.0'
gem 'appium_console', '~> 1.0.1'
```

After creating the `Gemfile` run `bundle install`. This will make sure `rspec` (our testing framework), `appium_lib` (the Appium Ruby bindings), and `appium_console` (our interactive test console) are installed and ready for use in this directory.

Capabilities

In order to run our tests, we will need to specify the capabilities of our app. We can either do this in our test code, or we can leverage the `appium.txt` files we used for the Appium Console.

Let's do the latter approach. But first, we'll want to create two new folders; one for Android and another for iOS. Once they're created, let's place each of the `appium.txt` files into their respective folders.

```
Gemfile
Gemfile.lock
android
   appium.txt
ios
    appium.txt
```

Be sure to update the `app` capability in your `appium.txt` files if you're using a relative path.

Writing Your First Test

With our initial setup taken care of, let's create our first test file (a.k.a. "spec" in RSpec). The test actions we identified in the previous post were focused on navigation in the app. So let's call this spec file `navigation_spec.rb` and place it in the `ios` folder.

```
Gemfile
Gemfile.lock
android
   appium.txt
ios
    appium.txt
    navigation_spec.rb
```

Now let's write our test to launch Appium for iOS and perform a simple navigation test.

```ruby
# filename: ios/navigation_spec.rb

require 'appium_lib'

describe 'Home Screen Navigation' do

  before(:each) do
    appium_txt = File.join(Dir.pwd, 'appium.txt')
    caps = Appium.load_appium_txt file: appium_txt
    Appium::Driver.new(caps).start_driver
    Appium.promote_appium_methods RSpec::Core::ExampleGroup
  end

  after(:each) do
    driver_quit
  end

  it 'First cell' do
    cell_1 = wait { text 2 }
    cell_title = cell_1.name.split(',').first

    wait { cell_1.click }
    wait { text_exact cell_title }
  end

end
```

In RSpec, `describe` denotes the beginning of a test file, whereas `it` denotes a test. So what we have is a test file with a single test in it.

In this test file, we are starting our Appium session before each test (e.g., `before(:each)`) and ending it after each test (e.g., `after(:each)`). More specifically, in `before(:each)`, we are finding the path to the iOS `appium.txt` file and then loading it. After that we start the Appium session and promote the Appium commands so they will be available for use within our test. We then issue `driver_quit` in `after(:each)` to cleanly end the Appium session. This is equivalent to submitting an `x` command in the Appium console.

The commands in our test (`it 'First cell' do`) should look familiar from the last post. We're finding the first cell, grabbing it's title, click on the cell, and then looking to see if the title appeared on the inner screen.

After saving this file, let's change directories into the `ios` folder (e.g., `cd ios`), and run the test (assuming your Appium Server is running -- if not, load up the Appum GUI and click `Launch`) with `rspec navigation_spec.rb`. When it's running, you will see the iOS simulator launch, load up the test app, click the first cell, and then close.

This is a good start, but we can clean this code up a bit by leveraging some simple page objects and a central configuration.

A Page Objects Primer

Automated tests can quickly become brittle and hard to maintain. This is largely due to the fact that we are testing functionality that will constantly change. In order to combat this, we can use page objects.

Page Objects are simple objects that model the behavior of an application. So rather than writing your tests directly against your app, you can write them against these objects. This will make your test code more reusable, maintainable, and easier to fix when the app changes.

You can learn more about page objects here and here.

Refactoring Your First Test

Let's create a new directory called `pages` within our `ios` directory and create two new files in it: `home.rb` and `inner_screen.rb`. And while we're at it, let's create a new folder to store our test files (called `spec` -- which is a folder RSpec will know to look for at run time) and move our `navigation_spec.rb` into it.

```
Gemfile
Gemfile.lock
android
   appium.txt
ios
    appium.txt
    pages
       home.rb
       inner_screen.rb
    spec
        navigation_spec.rb
```

Let's open up `ios/pages/home.rb` to create our first page object.

```ruby
# filename: ios/pages/home.rb

module Pages
  module Home
    class << self

      def first_cell
        @found_cell = wait { text 2 }
        self
      end

      def title
        @found_cell.name.split(',').first
      end

      def click
        @found_cell.click
      end

    end
  end
end

module Kernel
  def home
    Pages::Home
  end
end
```

Since the Appium commands are getting promoted for use (instead of passing around a driver object), storing our page objects in a module is a cleaner approach (rather than keeping them in a class that we would need to instantiate).

To create the `Home` module we first wrap it in another module called `Pages`. This helps prevent any namespace collisions as well simplify the promotion of Appium methods.

In `Home`, we've created some simple static methods to mimic the behavior of the home screen (e.g., `first_cell`, `title`, `click`). By storing the found cell in an instance variable (e.g., `@found_cell`) and returning `self`, we will be able to chain these methods together in our test (e.g., `first_cell.title`). And in order to cleanly reference the page object in our test, we've made the `home` method available globally (which references this module).

Now let's open up `ios/pages/inner_screen.rb` and create our second page object.

```ruby
# filename: pages/inner_screen.rb

module Pages
  module InnerScreen
    class << self

      def has_text(text)
        wait { text_exact text }
      end

    end
  end
end

module Kernel
  def inner_screen
    Pages::InnerScreen
  end
end
```

This is the same structure as our previous page object. In it, we're performing an exact text search.

Let's go ahead and update our test to use these page objects.

```ruby
# filename: ios/spec/navigation_spec.rb

require 'appium_lib'
require_relative '../pages/home'
require_relative '../pages/inner_screen'

describe 'Home Screen Navigation' do

  before(:each) do
    appium_txt = File.join(Dir.pwd, 'appium.txt')
    caps = Appium.load_appium_txt file: appium_txt
    Appium::Driver.new(caps).start_driver
    Appium.promote_appium_methods RSpec::Core::ExampleGroup
    Appium.promote_singleton_appium_methods Pages
  end

  after(:each) do
    driver_quit
  end

  it 'First cell' do
    cell_title = home.first_cell.title
    home.first_cell.click
    inner_screen.has_text cell_title
  end

end
```

We first require the page objects (note the use of `require_relative` at the top of the file). We then promote the Appium methods to our page objects (e.g., `Appium.promote_singleton_appium_methods Pages`). Lastly, we update our test.

Now when we run our test from within the `ios` directory (e.g., `cd ios` then `rspec`) then it will run just the same as it did before.

Now the test is more readable and in better shape. But there is still some refactoring to do to round things out. Let's pull our test setup out of this test file and into a central config that we will be able to leverage for both iOS and Android.

Central Config

In RSpec, we can configure our test suite from a central location. This is typically done in a file called `spec_helper.rb`. Let's create a folder called `common` in the root of our project and add a `spec_helper.rb` file to it.

```
Gemfile
Gemfile.lock
android
   appium.txt
common
   spec_helper.rb
ios
    appium.txt
    pages
       home.rb
       inner_screen.rb
    spec
        navigation_spec.rb
```

Let's open up `common/spec_helper.rb`, add our test setup to it, and polish it up.

```ruby
# filename: common/spec_helper.rb

require 'rspec'
require 'appium_lib'

def setup_driver
  return if $driver
  caps = Appium.load_appium_txt file: File.join(Dir.pwd, 'appium.txt')
  Appium::Driver.new caps
end

def promote_methods
  Appium.promote_singleton_appium_methods Pages
  Appium.promote_appium_methods RSpec::Core::ExampleGroup
end

setup_driver
promote_methods

RSpec.configure do |config|

  config.before(:each) do
    $driver.start_driver
  end

  config.after(:each) do
    driver_quit
  end

end
```

After requiring our requisite libraries, we've created a couple of methods that get executed when the file is loaded. One is to setup (but not start) Appium and another is to promote the methods to our page objects and tests. This approach is taken to make sure that only one instance of Appium is loaded at any one time.

We then configure our test actions so they run before and after each test. In them we are starting an Appium session and then ending it.

In order to use this central config, we will need to require it (and remove the unnecessary bits) in our test.

```
# filename: ios/spec/navigation_spec.rb

require_relative '../pages/home'
require_relative '../pages/inner_screen'
require_relative '../../common/spec_helper'

describe 'Home Screen Navigation' do

  it 'First cell' do
    cell_title = home.first_cell.title
    home.first_cell.click
    inner_screen.has_text cell_title
  end

end
```

Note the order of the `require_relative` statements -- they are important. We need to load our page objects before we can load our `spec_helper`, or else the test won't run.

If we run the tests from within the `ios` directory with `rspec`, we can see everything execute just like it did before.

Now that we have iOS covered, let's wire up an Android test, some page objects, and make sure our test code to supports both devices.

Including Android

It's worth noting that in your real world apps you may be able to have a single set of tests and segmented page objects to help make things run seamlessly behind the scenes for both devices. And while the behavior in our Android test app is similar to our iOS test app, it's design is different enough that we'll need to create a separate test and page objects.

Let's start by creating `spec` and `pages` folders within the `android` directory and then creating page objects in `pages` (e.g., `home.rb` and `inner_screen.rb`) and a test file in `spec` (e.g., `navigation_spec.rb`).

```
Gemfile
Gemfile.lock
android
   appium.txt
   pages
      home.rb
      inner_screen.rb
   spec
       navigation_spec.rb
common
   spec_helper.rb
ios
    appium.txt
    pages
       home.rb
       inner_screen.rb
    spec
        navigation_spec.rb
```

Now let's open and populate our page objects and test file.

```
module Pages
  module Home
    class << self

      def first_cell
        @found_cell = wait { text 2 }
        self
      end

      def click
        @found_cell.click
      end

    end
  end
end

module Kernel
  def home
    Pages::Home
  end
end
```

This page object is similar to the iOS one except there's no title search (since we won't be needing

it).

```
module Pages
  module InnerScreen
    class << self

      def has_text(text)
        wait { find_exact text }
      end

    end
  end
end

module Kernel
  def inner_screen
    Pages::InnerScreen
  end
end
```

In this page object we're performing a search for an element by text (similar to the iOS example), but using `find_exact` instead of `text_exact` because of how the app is designed (we need to perform a broader search that will search across multiple attributes, not just the text attribute).

Now let's wire up our test.

```
require_relative '../pages/home'
require_relative '../pages/inner_screen'
require_relative '../../common/spec_helper'

describe 'Home Screen Navigation' do

  it 'First cell' do
    home.first_cell.click
    inner_screen.has_text 'Accessibility Node Provider'
  end

end
```

Now if we `cd` into the `android` directory and run our test with `rspec` it should launch the Android emulator, load the app, click the first cell, and then end the session. The emulator will remain open, but that's something we'll address in a future post.

One More Thing

If we use the console with the code that we have right now, we won't be able to reference the page objects we've created -- which will be a bit of a pain if we want to reference them when debugging test failures. Let's fix that.

Let's create a new file in our `android/spec` and `ios/spec` directories called `requires.rb`. We'll move our require statements out of our test files and into these files instead.

```
Gemfile
Gemfile.lock
android
   appium.txt
   pages
      home.rb
      inner_screen.rb
   spec
       navigation_spec.rb
       requires.rb
common
   spec_helper.rb
ios
    appium.txt
    pages
       home.rb
       inner_screen.rb
    spec
        navigation_spec.rb
        requires.rb
```

Here's what one of them should look like:

```
# filename: ios/spec/requires.rb

# require the ios pages
require_relative '../pages/home'
require_relative '../pages/inner_screen'

# setup rspec
require_relative '../../common/spec_helper'
```

Next, we'll want to update our tests to use this file.

```
require_relative 'requires'

describe 'Home Screen Navigation' do

  it 'First cell' do
    cell_title = home.first_cell.title
    home.first_cell.click
    inner_screen.has_text cell_title
  end

end
```

```
# filename: android/spec/navigation_spec.rb

require_relative 'requires'

describe 'Home Screen Navigation' do

  it 'First cell' do
    home.first_cell.click
    inner_screen.has_text 'Accessibility Node Provider'
  end

end
```

Now that we have a central `requires.rb` for each device, we can tell the Appium Console to use it. To do that, we'll need to add some additional info to our `appium.txt` files.

```
# filename: ios/appium.txt

[caps]
deviceName = "iPhone Simulator"
platformName = "ios"
app = "../../../apps/UICatalog.app.zip"

[appium_lib]
require = ["./spec/requires.rb"]
```

```
# filename: android/appium.txt

[caps]
platformName = "android"
app = "../../../apps/api.apk"
avd = "training"
deviceName = "Android"

[appium_lib]
require = ["./spec/requires.rb"]
```

This new `require` value is only used by the Appium Console. Now if we run `arc` from either the `ios` or `android` directories, we'll be able to access the page objects just like in our tests.

And if we run our tests from either directory, they will still work as directed.

Outro

Now that we have our tests, page objects, and central configuration all sorted, it's time to look at wrapping our test execution and make it so we can run our tests in the cloud.

Running Your Tests

Now that we have our tests written, refactored, and running locally it's time to make them simple to launch by wrapping them with a command-line executor. After that, we'll be able to easily add in the ability to run them in the cloud.

Quick Setup

`appium_lib` comes pre-wired with the ability to run our tests in Sauce Labs, but we're still going to need two additional libraries to accomplish everything; `rake` for command-line execution, and `sauce_whisk` for some additional tasks not covered by `appium_lib`.

Let's add these to our `Gemfile` and run `bundle install`.

```
# filename: Gemfile

source 'https://rubygems.org'

gem 'rspec', '~> 3.0.0'
gem 'appium_lib', '~> 4.0.0'
gem 'appium_console', '~> 1.0.1'
gem 'rake', '~> 10.3.2'
gem 'sauce_whisk', '~> 0.0.13'
```

Simple Rake Tasks

Now that we have our requisite libraries let's create a new file in the project root called `Rakefile` and add tasks to launch our tests.

```
# filename: Rakefile

desc 'Run iOS tests'
task :ios do
  Dir.chdir 'ios'
  exec 'rspec'
end

desc 'Run Android tests'
task :android do
  Dir.chdir 'android'
  exec 'rspec'
end
```

Notice that the syntax in this file reads a lot like Ruby -- that's because it is (along with some Rake specific syntax). For a primer on Rake, read this.

In this file we've created two tasks. One to run our iOS tests, and another for the Android tests. Each task changes directories into the correct device folder (e.g., `Dir.chdir`) and then launches the tests (e.g., `exec 'rspec'`).

If we save this file and run `rake -T` from the command-line, we will see these tasks listed along with their descriptions.

```
> rake -T
rake android  # Run Android tests
rake ios      # Run iOS tests
```

If we run either of these tasks (e.g., `rake android` or `rake ios`), they will execute the tests locally for each of the devices.

Running Your Tests In Sauce

As I mentioned before, `appium_lib` comes with the ability to run Appium tests in Sauce Labs. We just need to specify a Sauce account username and access key. To obtain an access key, you first need to have an account (if you don't have one you can create a free trial one here). After that, log into the account and go to the bottom left of your dashboard; your access key will be listed there.

We'll also need to make our apps available to Sauce. This can be accomplished by either uploading the app to Sauce, or, making the app available from a publicly available URL. The prior approach is easy enough to accomplish with the help of `sauce_whisk`.

Let's go ahead and update our `spec_helper.rb` to add in this new upload capability (along with a couple of other bits).

```ruby
# filename: common/spec_helper.rb

require 'rspec'
require 'appium_lib'
require 'sauce_whisk'

def using_sauce
  user = ENV['SAUCE_USERNAME']
  key  = ENV['SAUCE_ACCESS_KEY']
  user && !user.empty? && key && !key.empty?
end

def upload_app
  storage = SauceWhisk::Storage.new
```

```
  app = @caps[:caps][:app]
  storage.upload app

  @caps[:caps][:app] = "sauce-storage:#{File.basename(app)}"
end

def setup_driver
  return if $driver
  @caps = Appium.load_appium_txt file: File.join(Dir.pwd, 'appium.txt')
  if using_sauce
    upload_app
    @caps[:caps].delete :avd # re: https://github.com/appium/ruby_lib/issues/241
  end
  Appium::Driver.new @caps
end

def promote_methods
  Appium.promote_singleton_appium_methods Pages
  Appium.promote_appium_methods RSpec::Core::ExampleGroup
end

setup_driver
promote_methods

RSpec.configure do |config|

  config.before(:each) do
    $driver.start_driver
  end

  config.after(:each) do
    driver_quit
  end

end
```

Near the top of the file we pull in `sauce_whisk`. We then add in a couple of helper methods (`using_sauce` and `upload_app`). `using_sauce` checks to see if Sauce credentials have been set properly. `upload_app` uploads the application from local disk and then updates the capabilities to reference the path to the app on Sauce's storage.

We put these to use in `setup_driver` by wrapping them in a conditional to see if we are using Sauce. If so, we upload the app. We're also removing the `avd` capability since it will cause issues with our Sauce run if we keep it in.

Next we'll need to update our `appium.txt` files so they'll play nice with Sauce.

```
# filename: android/appium.txt

[caps]
appium-version = "1.2.0"
deviceName = "Android"
platformName = "Android"
platformVersion = "4.3"
app = "../../../apps/api.apk"
avd = "training"

[appium_lib]
require = ["./spec/requires.rb"]
```

```
# filename: ios/appium.txt

[caps]
appium-version = "1.2.0"
deviceName = "iPhone Simulator"
platformName = "ios"
platformVersion = "7.1"
app = "../../../apps/UICatalog.app.zip"

[appium_lib]
require = ["./spec/requires.rb"]
```

In order to work with Sauce we need to specify the `appium-version` and the `platformVersion`. Everything else stays the same. You can see a full list of Sauce's supported platforms and configuration options here.

Now let's update our Rake tasks to be cloud aware. That way we can specify at run time whether to run things locally or in Sauce.

```
desc 'Run iOS tests'
task :ios, :location do |t, args|
  location_helper args[:location]
  Dir.chdir 'ios'
  exec 'rspec'
end

desc 'Run Android tests'
task :android, :location do |t, args|
  location_helper args[:location]
  Dir.chdir 'android'
  exec 'rspec'
end

def location_helper(location)
  if location != 'sauce'
    ENV['SAUCE_USERNAME'], ENV['SAUCE_ACCESS_KEY'] = nil, nil
  end
end
```

We've updated our Rake tasks so they can take an argument for the location. We then use this argument value and pass it to `location_helper`. The `location_helper` looks at the location value -- if it is not set to `'sauce'` then the Sauce credentials get set to `nil`. This helps us ensure that we really do want to run our tests on Sauce (e.g., we have to specify both the Sauce credentials AND the location).

Now we can launch our tests locally just like before (e.g., `rake ios`) or in Sauce by specifying it as a location (e.g., `rake ios['sauce']`)

But in order for the tests to fire in Sauce Labs, we need to specify our credentials somehow. We've opted to keep them out of our `Rakefile` (and our test code) so that we can maintain future flexibility by not having them hard-coded; which is also more secure since we won't be committing them to our repository.

Specifying Sauce Credentials

There are a few ways we can go about specifying our credentials.

Specify them at run-time

```
SAUCE_USERNAME=your-username SAUCE_ACCESS_KEY=your-access-key rake ios['sauce']
```

Export the values into the current command-line session

```
export SAUCE_USERNAME=your-username
export SAUCE_ACCESS_KEY=your-access-key
```

Set the values in your bash profile (recommended)

```
# filename: ~/*.bash_profile

...
export SAUCE_USERNAME=your-username
export SAUCE_ACCESS_KEY=your-access-key
```

After choosing a method for specifying your credentials, run your tests with one of the Rake task and specify `'sauce'` for the location. Then log into your Sauce Account to see the test results and a video of the execution.

Making Your Sauce Runs Descriptive

It's great that our tests are now running in Sauce. But it's tough to sift through the test results since the name and test status are nondescript and all the same. Let's fix that.

Fortunately, we can dynamically set the Sauce Labs job name and test status in our test code. We just need to provide this information before and after our test runs. To do that we'll need to update the RSpec configuration in `common/spec_helper.rb`.

```
# filename: common/spec_helper.rb

...
RSpec.configure do |config|

  config.before(:each) do |example|
    $driver.caps[:name] = example.metadata[:full_description] if using_sauce
    $driver.start_driver
  end

  config.after(:each) do |example|
    if using_sauce
      SauceWhisk::Jobs.change_status $driver.driver.session_id, example.exception.nil?
    end
    driver_quit
  end

end
```

In `before(:each)` we update the `name` attribute of our capabilities (e.g., `caps[:name]`) with the

name of the test. We get this name by tapping into the test's metadata (e.g., `example.metadata[:full_description]`). And since we only want this to run if we're using Sauce we wrap it in a conditional.

In `after(:each)` we leverage `sauce_whisk` to set the job status based on the test result, which we get by checking to see if any exceptions were raised. Again, we only want this to run if we're using Sauce, so we wrap it in a conditional too.

Now if we run our tests in Sauce we will see them execute with the correct name and job status.

Outro

Now that we have local and cloud execution covered, it's time to automate our test runs by plugging them into a Continuous Integration (CI) server.

Automating Your Test Runs

To make our tests as useful as possible, we'll want to automate when they get run. To do that, we'll use a Continuous Integration (CI) Server.

A Continuous Integration Server Primer

A Continous Integration server (a.k.a. CI) is responsible for merging code that is actively being developed into a central place (e.g., "trunk" or "master") frequently (e.g., several times a day, or on every code commit) to find issues early so they can be addressed quickly -- all for the sake of releasing working software in a timely fashion.

With CI, we can automate our test runs so they can happen as part of the development workflow. The lion's share of tests that are typically run on a CI Server are unit (and potentially integration) tests. But we can very easily add in our automated mobile tests.

There are numerous CI Servers available for use today. Let's pick one and step through an example.

A CI Example

Jenkins is a fully functional, widely adopted, and open-source CI server. It's a great candidate for us to step through.

Let's start by setting it up on the same machine as our Appium Server. Keep in mind that this isn't the "proper" way to go about this -- it's merely beneficial for this example. To do it right, the Jenkins server (e.g., master node) would live on a machine of its own.

Quick Setup

A simple way to get started is to grab the latest Jenkins war file. You can grab it from the Jenkins homepage, or from this direct download link.

Once downloaded, launch it from your terminal.

```
java -jar /path/to/jenkins.war
```

You will now be able to use Jenkins by visiting `http://localhost:8080/` in your browser.

Running Tests Locally

After loading Jenkins in the browser, we'll create a Job and configure it to run our Appium tests. Let's start with the Android tests first.

1. Click `New Item` in the top-left corner
2. Type a name into the `Item name` input field (e.g., `Appium Android`)
3. Select `Build a free-style software project`
4. Click `OK`

This will load a configuration screen for the Jenkins Job.

1. Scroll down until you reach the `Build` section (near the bottom of the page)
2. Click `Add build step`
3. Select `Execute shell`
4. Input the following into the `Command` input box

```
cd /path/to/your/appium/test/code
bundle update
rake android
```

In this set of commands we are telling Jenkins to change directories to our test code, make sure we have the necessary libraries, and then launch the Android tests.

Click `Save` at the bottom of the page, make sure your Appium Server is running (if not, load up the Appium GUI and click `Launch`), and click `Build Now` on the left-hand side of the Jenkins Job screen.

Once it's running, you can click on the job under `Build History`, and then click `Console Output` (from the left-hand panel). In it, you should see something similar to this:

```
Started by user anonymous
Building in workspace /Users/tourdedave/.jenkins/jobs/Appium Android/workspace
[workspace] $ /bin/sh -xe
/var/folders/yt/h7v9k6px7jl68q81c9sqrd9h0000gn/T/hudson6140596697737249507.sh
+ cd /Users/tourdedave/Dropbox/_dev/appium/appium-getting-started/code-examples/7/1
+ bundle update
Fetching gem metadata from https://rubygems.org/...........
Fetching additional metadata from https://rubygems.org/..
Resolving dependencies...
Using rake 10.3.2
Using awesome_print 1.2.0
Using json 1.8.1
Using mini_portile 0.6.0
Using nokogiri 1.6.3.1
Using ffi 1.9.3
Using childprocess 0.5.3
Using multi_json 1.10.1
Using rubyzip 1.1.6
Using websocket 1.0.7
```

```
Using selenium-webdriver 2.42.0
Using blankslate 2.1.2.4
Using parslet 1.5.0
Using toml 0.1.1
Using appium_lib 4.0.0
Using bond 0.5.1
Using coderay 1.1.0
Using method_source 0.8.2
Using slop 3.6.0
Using pry 0.9.12.6
Using numerizer 0.1.1
Using chronic_duration 0.10.5
Using spec 5.3.4
Using appium_console 1.0.1
Using diff-lcs 1.2.5
Using mime-types 1.25.1
Using rdoc 4.1.1
Using rest-client 1.6.8
Using rspec-support 3.0.3
Using rspec-core 3.0.3
Using rspec-expectations 3.0.3
Using rspec-mocks 3.0.3
Using rspec 3.0.0
Using sauce_whisk 0.0.13
Using bundler 1.6.2
Your bundle is updated!
+ rake android
.

Finished in 38.39 seconds (files took 1.52 seconds to load)
1 example, 0 failures
Finished: SUCCESS
```

Making Sure We Have A Clean Finish

We now have a working job in Jenkins. But we're not there yet. While the job was runnning you should have seen the Android Emulator open, load the test app, and perform the test actions. Unfortunately, after the job completed, the emulator didn't close.

Closing the Android Emulator is something that Appium doesn't handle, so we'll need to account for this in our Jenkins build configuration. Otherwise, we won't leave things in a clean state for future test runs.

The simplest way to close the emulator is by issuing a `kill` command against the name of the process (ensuring that the command always returns `true`). That way we cover our bases in case there is more than one emulator process running or if we try to kill a process that doesn't exist.

So let's go ahead and add the `kill` command to our existing commands under the `Build` section of our job. For good measure, let's add it before and after our test execution commands.

To get back to the job configuration screen, click `Configure` from the main job screen.

```
killall -9 emulator64-x86 || true

cd /path/to/your/appium/test/code
bundle update
rake android

killall -9 emulator64-x86 || true
```

Now let's save the job and build it again. The job will run just like before, but now the emulator will close after the test run completes.

Creating Another Job

Now let's create a second job to run our tests against iOS.

To save a step, let's create a copy of our existing job and modify the build commands as needed.

1. Click the Jenkins logo at the top of the screen (it will take you to the main page)
2. Click `New Item` in the top-left corner
3. Type a name into the `Item name` input field (e.g., `Appium iOS`)
4. Select `Copy existing Item`
5. Start to type in the name of the other job in the `Copy from` input field (e.g., `Appium Android`)
6. Select the job from the drop-down as it appears
7. Click `OK`

This will take us to a configuration screen for the new (copied) job. Let's scroll down to the `Build` section and modify the `Command` input field under `Execute Shell`.

```
killall -9 "iPhone Simulator" &> /dev/null || true
killall -9 instruments &> /dev/null || true

cd /path/to/your/appium/test/code
bundle update
rake ios

killall -9 "iPhone Simulator" &> /dev/null || true
killall -9 instruments &> /dev/null || true
```

Similar to the Android job, we're using `kill` to end a process (in this case two processes) and

making sure the command returns `true` if it doesn't exist. This protects us in the event that the test suite doesn't complete as planned (leaving a simulator around) or if the simulator doesn't close instruments cleanly (which can happen).

If we save this and build it, then we will see the iPhone Simulator load, launch the app, run the tests, and then close the simulator.

Running Tests On Sauce

We've covered running things locally on the CI server, now let's create a job to run our tests on Sauce.

Let's create another copy of the `Appium Android` job and modify the build commands.

Since we're not going to be running locally, we can remove the `kill` line. We'll then specify our Sauce credentials (through environment variables) and update the `rake` command to specify `'sauce'` as a location. When we're done, our `Command` window should look like this:

```
export SAUCE_USERNAME=your-username
export SAUCE_ACCESS_KEY=your-access-key

cd /path/to/your/appium/test/code
bundle update
rake android['sauce']
```

If we save this and build it, our tests will now run on Sauce Labs. And you can view them as they happen on your Sauce Labs Account Page.

An iOS job would be identical to this, except for the job name (e.g., `Appium iOS Sauce`) and the `rake` incantation (which would be `rake ios['sauce']`.

Outro

Now that we have our Appium tests wired up for automatic execution, we're now able to configure them to run based on various triggers (e.g., other CI jobs, a schedule, etc.). Find what works for you and your development team's workflow, and make it happen.

Finding Information On Your Own

Now that you're up and running with Appium locally, in the cloud, and on a CI solution, it's best to show you where you can find more information. Below is a collection of some great resources to help you find your way when it comes to mobile testing.

Community Support

- Appium Android Tutorial
- Appium iOS Tutorial

These are the official tutorials for the Appium project for Android and iOS. They served as inspiration and a base for this getting started series. They are great follow-on material since they cover various topics in more depth, and include Java examples as well.

- Appium's Google Discussion Group

If you have an issue or a question, this is a great place to turn to. Before posting an issue, be sure to read through the Appium Troubleshooting docs and search the group to see if your question has already been asked/answered.

- Appium Chat Channel

In addition to the Google Discussion Group, you can hop on the Appium HipChat chat room and ask questions from others in the Appium community.

- Live panel Q&A with the core Appium committers

This is a follow-up post answering loads of questions from a webinar from just after thet Appium 1.0 release. It's chocked full of a lot of great information.

- GTAC 2013: Appium: Automation for Mobile Apps

In this video, Jonathan Lipps (Appium's Chief Architect) explains mobile automation with Appium.

- The Appium Book

This is an open-source book that is a work in progress; authored by Jonathan Lipps. It's working title is "Appium: Mobile Automation Made Awesome".

Some Android Specific Resources

- GTAC 2013: Breaking the Matrix - Android Testing at Scale
- GTAC 2013: Breaking the Matrix Q&A

- How the Google+ Team Tests Mobile Apps

These links (a video, Q&A, and a blog post) cover how Google approaches Android testing.

- GTAC 2013: Android UI Automation

uiautomator is a crucial component of Android test automation. In this video, the engineers behind it talk about it's future.

- GTAC 2013: Espresso: Fresh Start to Android UI Testing

This video is a walk through Google's newest Android testing framework. This isn't directly related to Appium, but it contains some useful information.

Some iOS Specific Resources

- Automating User Interface Testing With Instruments, by Apple at 2010 WWDC
- Documentation for UI Automation JavaScript
- Apple's iOS Accessibility Guide

Appium relies on Apple's UI Automation support, and these are some solid resources for understanding it better.

Professional Support

Sauce Labs offers support for Appium as part of their hosted cloud offering. If you are a Sauce customer and encounter an issue when using their platform with Appium, be sure to open a support ticket.

If you're using Appium and you think you've found a bug specific to either Android or iOS, then let Google and/or Apple know. In either case it's best to make sure that the bug is not an Appium issue before filing an issue.

For Google, file an issue here.

For Apple, file an issue here. Apple keeps all bugs private, so it's worth also filing a duplicate issue here.

Straight To The Source

- How to search the Appium source code (Android)
- How to search the Appium source code (iOS)

These are great instructions on how to search through the Appium source code to find more information.

Some Other Resources

- Appium Stack Overflow issues

There are over 600 Appium questions posted on Stack Overflow for you to peruse.

- Cheat Sheet for Top Mobile App Controls

Xamarin has a free cheat sheet comparing popular mobile app controls. Definitely worth a look.

Outro

Now you're ready, armed with all the information you need to continue your mobile testing journey.

Happy Testing!