Appium Bootcamp 5: Writing and Refactoring Your Tests

Appium Bootcamp is a series of articles prepared by Selenium guru Dave Haeffner, and leading Appium contributor Matthew Edwards, for Sauce Labs. Dave also authors the Elemental Selenium website, which includes tips for using Selenium, and where you can sign up for his weekly email on the topic of Selenium testing. He is also the author of the Selenium Guidebook.

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.

```
# 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.

```
# 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.

```
# 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.

```
# 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.

```
# 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.

```
equire_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.