Appium Bootcamp 3: Interrogating Your App

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.

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 `UIAStaticTex` 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`).

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.

```
[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">
```

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.