Appium Bootcamp 4: Your First Test

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.

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.

```
sh 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.