Calabash-Android Exercises

Objective

These exercises will teach you how to write and execute Calabash Android features.

- Installing and Setting up Calabash

- Writing 'proper' step definitions using the Calabash Android Client API

- Advanced APIs

Prerequisites

You can run Calabash Android on Windows, Mac or Linux.

We recommend that you use Ruby 1.9.X or 2.x. For OS X Ruby is already installed. On Windows you can use http://rubyinstaller.org/

You need the following configured on your computer:

- The Android SDK
- The ANDROID_HOME environment variable should point to your Android SDK.
- Your Android phone should be connected via USB and allow USB-debugging. If you don't have an Android phone you can use an emulator but a physical device is usualy the faster and more reliable. Alternatively Genymotion is quite good.

On OS X run the script: `./install-calabash-local-osx.rb`

or

```
mkdir -p ~/.calabash && GEM_HOME=~/.calabash gem install cucumber calabash-android
```

Edit your .bash_profile and set the GEM_HOME and GEM_PATH environment variables. It is also necessary to update the $PATH to include `~/.calabash/bin`. The following snippet shows an example of this:

```
export GEM_HOME=~/.calabash
export GEM_PATH=~/.calabash
export PATH="$PATH:$HOME/.calabash/bin"
```

For details, see: http://developer.xamarin.com/guides/testcloud/calabash/osx-installation/

On Windows just do `gem install cucumber calabash-android`

You should have calabash-android 0.5.4 installed.

Exercise - Playing around with the Calabash API on Android

The console is an interactive ruby environment where you can play with the Calabash API and start automating your app.

1. Time to try the calabash-android console. Open a terminal and go into your project directory. Connect your Android device or start an emulator. Then in the terminal, enter:

```
./console.rb android
```

or

```
calabash-android console prebuilt/Android-debug.apk
```

2. Make sure the app and the test server is installed by running `reinstall_apps`.
3. Now start the test server (and the app) by running: `start_test_server_in_background`
4. You now have a console that you can use to experiment with the API. Let's try using the query function from Calabash Android. Try the following expressions:

```
query("android.widget.ImageView")
query("android.widget.TextView")
query("android.widget.TextView index:0")
query("android.widget.TextView marked:'Sign in'")
```

Can you make any sense of the output? Discuss it with your team. Ask us if in doubt.
Here are a bunch of more queries to try. For each of them, explain to each other what is happening?

```
query("*")
query("android.widget.TextView", :text) #What is this??
query("android.widget.TextView text:'Sign in'")
```

The console is very nice for exploring your app. Often when working with Calabash, you'll be spending a lot of time in the console. Then, once you're happy with the queries in the console, you'll copy those queries into a more permanent step definition. This is the next step.

5. Touching. Let's introduce the `touch` function: Anything you can query, you can also touch using the `touch` function.

First dismiss the keyboard if it is showing (the back button)

Try running this command in the console:

```
touch("android.widget.TextView marked:'Add self-hosted site'")
```

```
enter_text("android.widget.EditText id:'nux_password'", 'badpass')
```

there is also `keyboard_enter_text`

Alternatively you can call setText directly on the view objects (keyboard entry is preferred):

```
query("android.widget.EditText index:0", setText: 'calabash')
query("android.widget.EditText id:'nux_password'", setText: 'Calabash321')
query("android.widget.EditText id:'nux_url'", setText: 'Calabash')
```

A useful query on Android is often

```
    query("*", :id)
```

Can you guess what that does?

The Query system is quite advanced for finding views. You can learn more about the query syntax here:

http://developer.xamarin.com/guides/testcloud/calabash/calabash-query-syntax/

Exercise 2 - Waiting

Proper waiting is important in automated testing. A test often needs to wait for something. Examples could be waiting for login to complete before proceeding, waiting for an activity indicator to stop spinning or waiting for an animation to complete.

The simplest way to wait is using Ruby's built-in `sleep` function: this will pause the test for a number of seconds, e.g., `sleep(3)`

There are a number of problems with this

- for example on login, what if the server responds slowly? Test may fail even though everything is OK.

- What if the server responds quickly and you are logged in after one second. With `sleep(3)`, You're wasting two seconds waiting for nothing!

So too many calls to sleep lead to fragile and slow tests. This is why calabash has a number of functions to support efficient waiting.

Start the calabash-android console and start the app from the console.

1. Now try typing the following into the console

```
    wait_for_element_exists "* id:'nux_url'"
```

2. Now manually enter user/pass and touch "Sign In" (before 10 seconds have passed!)
3. What happened? Try again.

The following waiting functions are useful: `wait_for_elements_exist`, `wait_for_elements_do_not_exist`

Notice that these functions take arrays as their first argument - they wait for several 'things'. They actually take an optional second argument to control other aspects of waiting. More on that shortly.

4. The most low-level waiting primitive is the `wait_for` function. It works as follows:

```
    wait_for(timeout: 10) do
        res = query("android.widget.TextView", :text)
        res.include?("Create account")
    end
```

:timeout => 10 means wait for at most 10 seconds or fail. The **do..end** block will be run repeatedly as long as the block produces the result **false** or **nil**. (Remember the last value in a block is what the block produces).

You can use the `wait_for` function as a building-block to make your own waiting functions.

Exercise 3 - Put it all together - Writing a Scenario

The next goal is to use the actual Calabash Android Ruby API to write a scenario.

There are several reasons why you want to use the Ruby API instead of predefined-steps.

- More robust tests
- More powerful/expressive
- Faster tests
- Faster development experience

Let's see how that is done

OK, everything is good - we can search for UI components using `query` and we can touch components using `touch`. Let's do more of this to get experience!

1. Read through the Scenario which logs in to a Self-Hosted site using

   username 'calabash'

   password 'password'

   Site address: ec2-54-82-18-238.compute-1.amazonaws.com/wordpress

Notice how this is using the Page Object Pattern to reuse features and step definitions across platforms.

Once you understand this, try writing a new scenario by extending the page object methods. The best way to work is usually playing around in the console and then copying selected queries and actions into the Ruby methods in the page objects. Be aware that timings are usually different when you're in the console and running the full test.

2. Make screenshots by calling the `screenshot_embed` function where you need screenshots.
3. Try generating an HTML report using Cucumber

```
./run.rb android --format html --out report.html
```

Open the report. (Notice embedded screenshots)

Exercise 4 - Hybrid support

The WordPress app is a 'hybrid' app. Some of the screens are implemented using web views and HTML, CSS and JavaScript loaded from wordpress.com.

Make sure you're logged in - then go to "View site"

1. Fire up the console and type query("webView") - this reveals a web view.
2. Look at the web content and play with it.

```
query("webView css:'a'")
```

3. If you needed to you could, fill in text into input fields you can use keyboard_enter_text or you will have to use the set_text action with the rather cryptic method call

```
performAction "set_text", "css", "input[id=user_email]","karl@lesspainful.com"
```

4. You can filter on css queries: query("webView css:'a' textContent:'Google'")
5. With Calabash you can evaluate arbitrary JavaScript inside the web view.

```
    performAction "execute_javascript", "return 2+2"
    performAction "execute_javascript", "return document.body.innerHTML"
```

Exercise 5 - Predicates

Let's go back to the `query` function. It is used to inspect the current visible view. Often queries match based on content descriptions or accessibility identifiers. Sometimes you also match on text:

```
    query("* marked:'Posts'")   ---   query("android.widget.TextView text:'Posts'")
```

1. Sometimes you need to find text that matches a certain pattern instead of making an exact match. Try the following in console:

```
    query("android.widget.TextView {text BEGINSWITH '...'}")
    query("android.widget.TextView {text ENDSWITH '...'}")
    query("android.widget.TextView {text CONTAINS '..'}")
    query("android.widget.TextView {text CONTAINS[c] '..'}")
    query("android.widget.TextView {text LIKE '..*..'}")
```

(Note: you can use any single selector,e.g., content description or id - you don't have to use text.)