Calabash-iOS Exercises

Objective

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

- Installing and Setting up Calabash
- Writing step definitions with predefined steps
- Writing 'proper' step definitions using the Calabash iOS Client API
- Advanced APIs

Prerequisites

You have to use a Mac and you have to Have XCode installed (version 6.1+).

You must also install Calabash iOS which uses Ruby: On OS X, you can install rbenv (see below) or use the OS X built-in ruby.

For built-in Ruby on OS X Mavericks and above run this command

```
./install-calabash-local-osx.rb
```

or

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

You've installed Calabash for iOS. Now to setup Calabash for easy use in future sessions:

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"
```

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:

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

On Mavericks, people sometimes get a compiler error. In that case install with the following ARCHFLAGS

```
export ARCHFLAGS=-Wno-error=unused-command-line-argument-hard-error-in-future
gem install calabash-cucumber
or
sudo ARCHFLAGS=-Wno-error=unused-command-line-argument-hard-error-in-future gem
install calabash-cucumber
(all on one line)
```

Exercise - Playing around with the Calabash API on iOS

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-ios console. Open a Terminal and go into your project directory. Then in terminal, enter:

```
    ./console.rb ios
or
    APP_BUNDLE_PATH=prebuilt/WordPress-cal.app calabash-ios console
then in the console:
 > start_test_server_in_background
#which should launch the app
```

2. You now have an iOS console that you can use to experiment with the API. Let's try using the query function from Calabash iOS. Try the following expressions:

```
    query("button index:0")
    query("label index:0")
    query("button marked:'Add Self-Hosted Site'")
```

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("button")  #What is this?
    query("button", :accessibilityLabel) #What is this?
    label("button") #What is this?
    query("label", :text) #What is this?
    query("button marked:'Add Self-Hosted Site'") #this again?
```

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.

3. Touching. Let's introduce the `touch` function: Anything you can query, you can also touch using the `touch` function. For example when you are on the "Home" screen, try running this command in calabash-ios console:

```
touch("UIButton marked:'Add Self-Hosted Site' ")
```

or simply

```
tap_mark "Add Self-Hosted Site"
```

4. Type `classes("view")` in the console. Can you guess what this does? (Check the docs). The results will include 'WPWalkthroughTextField' which is a subclass of iOS's UITestField

When you type a query like 'button' this translates to UIButton (the iOS class documented here).

- Similarly with 'label' -> UILabel, and 'textField' -> UITextField

- 'view' -> UIView (the superclass of all UIKit view classes), etc.

If you type in a query starting with capital letters it will be interpreted as a classname, e.g., UIButton.

5. In the Calabash console. Bring up keyboard (if not visible). Now, in the console, try typing:

```
keyboard_enter_text "karl"
```

Try using

```
done
```

in the console.

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.

Pop-up the console and run start_test_server_in_background.

1. From the Home screen, enter "karl" and "karl" (but don't Sign in). Now try typing the following into the console

```
wait_for_element_exists ("* marked:'Need Help?'")
```

6. Now in the simulator manually touch "Sign in" (before 10 seconds have passed!)

7. What happened? Try, `wait_for_element_exists ("* marked:'Need Help?'")` again.

The following waiting functions are useful: `wait_for_element_exists`, `wait_for_elements_do_not_exist`, e.g.,

```
wait_for_elements_do_not_exist( ["button marked:'Sign In'"] )
```

Notice that some of 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.

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

```
wait_for(timeout: 10) do
    res = query("label",:text)
    res.include?("Need Help?")
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).

Exercise 3 - Put it all together - Writing a Scenario

The next goal is to use the actual Calabash iOS 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

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.

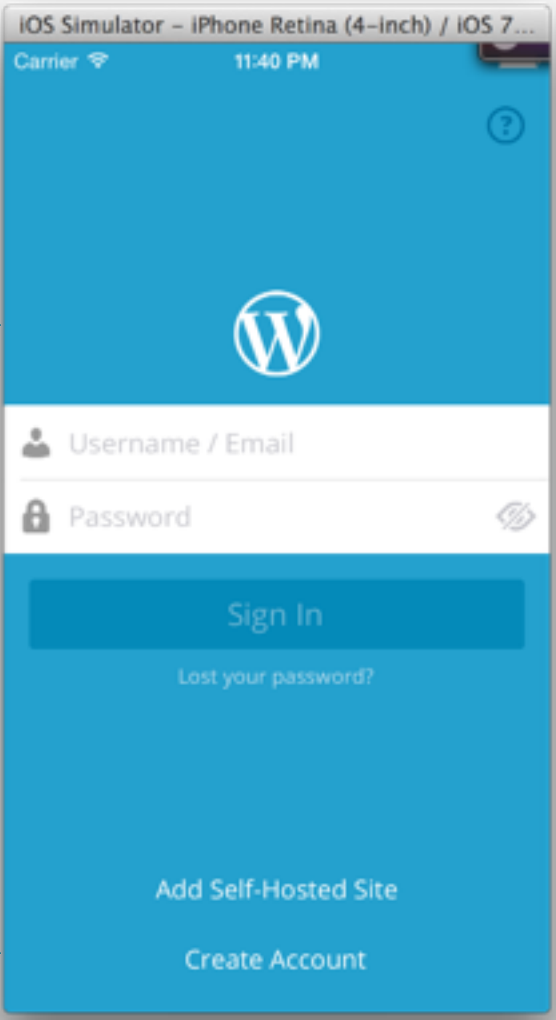
Once you understand this, try writing a new scenario by extending the page object methods. The best way to work is usually playing around in the console and then copying selected queries and actions into the Ruby methods in the page objects. Be aware that timings are usually different when you're in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

Open the report. (Notice embedded screenshots)

Important note: once logged in, wordpress remembers you even if app is closed. You can select Reset Content and Settings in the iOS Simulator menu.



Checkpoint C

Checkpoint question: If you're done with the above, you've probably ended up asserting you're on the Freshly Pressed screen. Did you assert the text 'Freshly Pressed' was there?

Did you assert that this text was the title of the navigation bar? How can we do that? Please read the documentation on the query language: https://github.com/calabash/calabash-ios/wiki/05-Query-syntax. Discus the query syntax page with your team mates.

Now can you assert that 'Posts' is the title of the navigation bar?

Exercise 8 - Scroll views and tables

Let's write a new feature concerning Posts. This is the "Me" tab bar button. Create a new feature file: posts.feature:

```
Feature: Posts
  Scenario: Posts available
    Given I am logged in
    And I am on Posts
    Then all the "calabashtraining" blog posts are available
```

Let's tackle this step by step. Start by implementing 'I am logged in' and 'I am on Posts'. This should be relatively easy given the last exercise.

To implement 'all the "calabashtraining" blog posts are available' we need some new stuff. For the sake of this exercise, "all blog posts" means the blog posts titled: "Calabash" and "Hello Utrecht", and "Example".

Let's write this in Ruby as:

```
target_titles = ['Calabash', 'Hello Utrecht', 'Example']
```

Fire up the console on the Posts screen.

1. Inside console, use query to figure out the class of the table-view cells representing posts.
2. Now figure out a query to find the titles of all visible posts (Hint: if X is a custom view class then the query "view:'X'" finds all visible views of type 'X').
3. In console, try using the following API method: scroll_to_row. You can also look at the documentation for the `scroll` method, and play with that. Finally for more complex tables, you can read about the `scroll_to_cell` method.
4. Combine `scroll_to_row` with either `wait_for` or a loop-ing construct to solve the exercise (i.e. to check that all `target_titles` are accessible).

Exercise 9 - 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.

Navigate to "Me" > "Calabash Blog" > "View Site"

1. Fire up the console and type query("webView") - this reveals a web view.
2. With Calabash you can query for css selectors in web views. Try running a few queries like

query("webView css:'a'") (search for Hello Utrecht) or query("webView css:'input'") (under leave a reploy).

3. Anything you can query, you can also touch: touch("webView css:'input'").
4. If a keyboard is shown, you can use the regular keyboard_enter_text.
5. Try running: touch("webView css:'a'") – what now? Think about this for a second.
6. You can filter on css queries: query("webView css:'a' textContent:'Hello Utrecht'").count
7. Even dynamic properties like 'value'. Create a comment - enter something, e.g., 'karl'. Then run: query("webView css:'textarea' value:'karl'")
8. With Calabash you can evaluate arbitrary JavaScript inside the web view. This is a low-level action which is generally not needed, but can be an useful escape-hatch. Try these:

```
query("webView", :stringByEvaluatingJavaScriptFromString => '2+2')


js = 'document.body.innerHTML'

query("webView", :stringByEvaluatingJavaScriptFromString => js)
```

Exercise 10 - Predicates and Advanced Queries.

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

```
query("view marked:'Posts'")  ---  query("label text:'Posts'")
```

Finds all views with accessibility label (or identifier) "Posts" (usually accessibility labels are unique).

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

```
query("label {text BEGINSWITH 'Hello'}")
query("label {text ENDSWITH 'Utrecht'}")
query("label {text CONTAINS 'trech'}")
query("label {text CONTAINS[cd] 'utrecht'}")
query("label {text LIKE 'He* Utrecht'}")
```

(Note: is is a subset of NSPredicates: https://developer.apple.com/library/ios/#documentation/Cocoa/Reference/Foundation/Classes/NSPredicate_Class/Reference/NSPredicate.html)

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

2. Remember the query

```
query("label", :text)
```

Notice the second argument to query – :text. This is actually maps to an Objective-C selector which is performed on each object that results from the query (i.e. all visible labels in the current view). You can put any simple selector like :text there. Take a look at:

http://developer.apple.com/library/ios/#DOCUMENTATION/UIKit/Reference/UILabel_Class/Reference/UILabel.html

And try extracting other properties from labels. For example can you get a representation of the color of a label?

Often knowing the iOS classes and their properties is extremely helpful when writing more advanced iOS tests. The docs from Apple are quite useful here.

3. Now navigate to the Posts screen. There is some special support for tables in Calabash. Particularly NSIndexPaths which are used to find particular cells in a table.

```
query("view:'NewPostTableViewCell' indexPath:2,0") #row, section
```

In general query("tableViewCell indexPath:x,y") will return the cell for section number y and row number x.

4. In its most general form, query supports calling selectors which take arguments and also chaining selectors. For example

```
query("tableView",:numberOfSections)
query("tableView",{:numberOfRowsInSection => 0})
```

Try experimenting with this. For example, can you find selectedViewController in the tab bar? Take a look at the delegate of the tabBar. What is its class? Find the documentation for this class.

For ruby 1.9+ you can even write

```
query("tableView", numberOfRowsInSection: 0)
```

which looks just like an Objective-C selector.

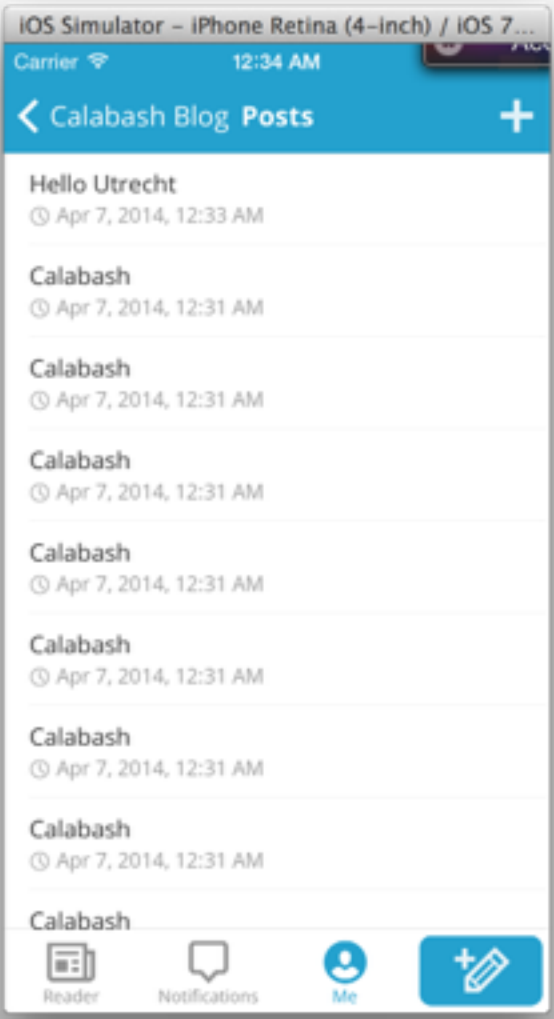
This is an advanced technique but can be useful! You can even do nasty stuff like

```
query("label", {:setText => "HI!"})
```

but you probably don't want to do this in your tests!

Checkpoint Last

Congrats! You made it!



Hello Utrecht
Apr 7, 2014, 12:33 AM

Calabash
Apr 7, 2014, 12:31 AM

Calabash
Apr 7, 2014, 12:31 AM

Calabash
Apr 7, 2014, 12:31 AM

Calabash
Apr 7, 2014, 12:31 AM

Calabash
Apr 7, 2014, 12:31 AM

Calabash
Apr 7, 2014, 12:31 AM

Calabash
Apr 7, 2014, 12:31 AM

Calabash