# Calabash 101

## Basics, Getting Started and Advanced Tips

testdroid.

# INTRODUCTION

Calabash is a test automation framework that enables mobile developers and pretty much anyone without coding skills to create and execute automated acceptance tests for Android and iOS apps. Calabash works by enabling automatic UI interactions within an application such as pressing buttons, inputting text, validating responses, etc.

While this is a great first step in automated UI acceptance test automation, the real benefits can be gained when Calabash tests are executed on real mobile devices. This is very easy and Calabash tests can be configured to run on hundreds of different Android and iOS devices, providing real-time feedback and validation across many different form factors, OS versions, OEM customizations and hardware specs.

As one of the earliest supported frameworks in Testdroid Cloud, Calabash has been widely used by mobile developers and testers. And with many years of experiences of supporting it, we have decided to compose an ebook regarding Calabash and share it with all of you.

For Calabash beginners, this ebook will shed light on what is Calabash and how you can utilize it. And for those who have been using Calabash many years, the ebook can also give you a deep understanding of this open source framework and refresh your knowledge about Calabash with some of our new tricks and tips.

This ebook provides with you a tutorial of best practices with Calabash, how to get started with it, how to combine Calabash approach across other frameworks, and some useful tips.

# CHAPTER 1

# The Basics of Calabash framework

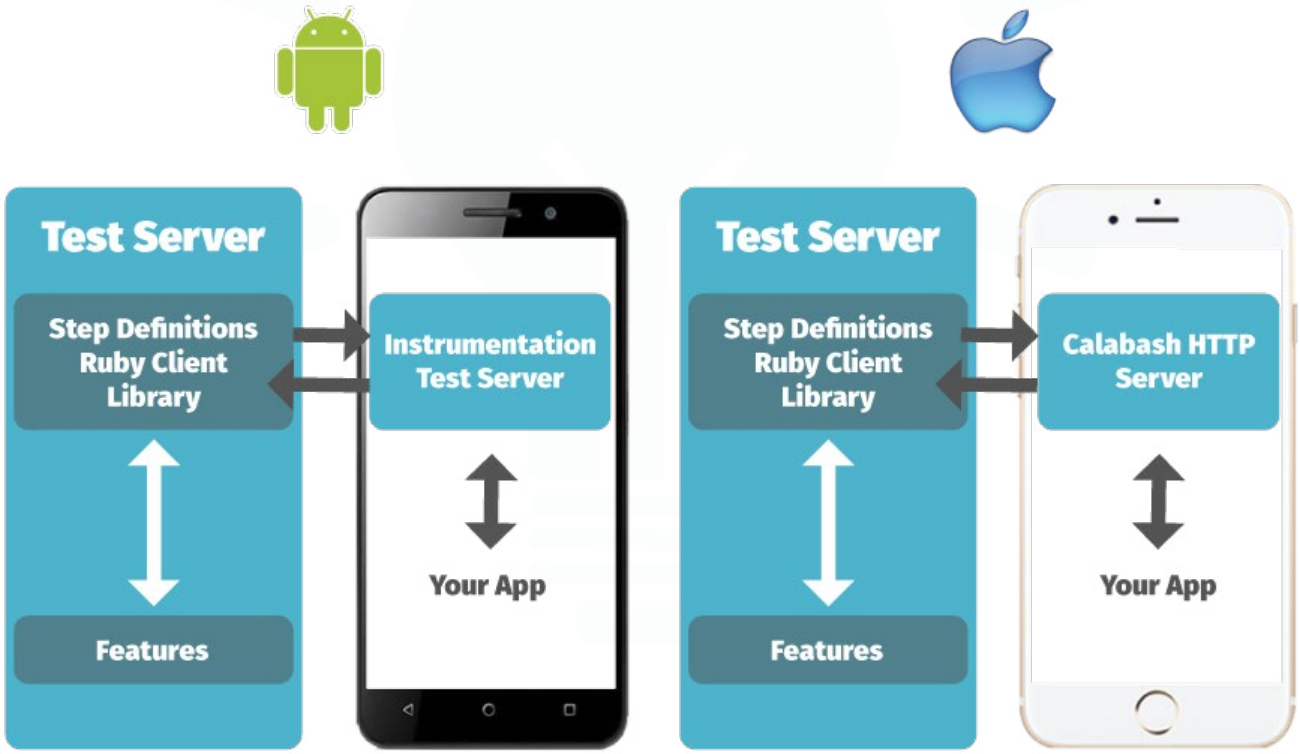


Share this eBook!

Before the nitty-gritty of Calabash, this chapter briefly explains what is Calabash and how to get the best out of Calabash and presents a diagram on how it works with Android and iOS.

## Calabash Framework Definition

Calabash is quite popular choice for test automation framework and works fairly well with both Android and iOS. Considered as Behavior-Driven Development (BDD) test automation framework, it basically means a philosophy of outside-in development. It's conceptually similar to Test-Driven Development (and is in fact based on it), but takes it one step further, in that instead of creating tests that describe the shape of APIs, application behaviors are specified.




The idea of Calabash isn't much different from the practice of creating functional specifications for software and then building to the spec, but it takes functional specifications and makes them executable, by providing a means in which to test the actual specifications. And

just as typical with specification creation, BDD is meant to be a process in which multiple stakeholders weigh in to create a common understanding of what is to be built. The benefit of the BDD approach is that it helps to ensure that the right software is built (as opposed to the software being built right). The intent is that the software is designed from the perspective of the business owner. That way, there is little interpretation required as to how the software should behave.

## Ensure You Get The Best Out of Calabash

Use only real mobile devices with Calabash. First of all, it's always recommended that you rely on real mobile devices in your development and testing. This enables you to get accurate information on how your app does on the same devices that end-users use. Having an authentic environment where you run your Calabash tests must include the various version of OS, OEM customizations (especially in case of Android), hardware with different setups, chipsets and amount of memory, and real-environment conditions (e.g. integration with back-end, network).

**Test coverage needs to be adequate.** Writing tests with Calabash is very easy and actually fun job to do. The language syntax is human-readable and you don't need coding experience or skills that are typically required for software developers. Writing, configuring and setting up tests can be also nicely scaled for different software versions (e.g. OS versions, and those OEM customizations) but you'll likely confront more fragmentation when it comes to running Calabash tests on different hardware.

**Real-user conditions.** When you have those real devices in your testing roster you also want to make sure there is enough capacity and power with your surrounding testing infrastructure. This means having capable network and if you want to use mobile network, you should equip devices with SIM cards. We constantly enable these sort of setups in our private cloud. There is availability of any Android and iOS devices, with what-ever infrastructure required for the setup, and all runs smoothly with your Agile development tools and environment.

**Integrated with CI/CD.** Calabash, like other test automation frameworks, can be easily integrated with continuous integration / continuous delivery tools, such as Jenkins. In addition, there is a comprehensive API for app testing that can be hooked with any additional development tool, testing framework or tool, and test management software.

# CHAPTER 2

# How to Setup and Get Started with Calabash



Share this eBook!

One of the greatest things with Calabash is that tests work seamlessly on both platforms if your application is identical. As this is not always the case you might need to break down test scripts for both platforms.

This chapter will discover how to support both major platforms with your Calabash tests and walk you through of very basic setup, installations and how to get started with Calabash.

## Calabash Prerequisites and Calabash Installation

Calabash requires Ruby to be installed on your machine. First, check whether you have Ruby installed on your machine:

```
$ ruby -v
```

We typically recommend the latest and greatest versions, but some older ones (e.g. 1.8.7 or 1.9.1) should work fine too. However, the latest versions are proven to work without any friction (especially with iOS) so if possible please upgrade to the latest one. We also recommend the use of Bundler to ensure you get all dependencies properly installed:

```
$ gem install bundler
$ bundle install
```

In addition, if you plan to manage several versions of Ruby, it's recommended to use tools like rbenv. If you are not using Mac or Linux, you can also install Ruby from RubyInstaller.org for Windows.

# ANDROID

If you want to use Calabash for Android, make sure you have the latest and greatest SDK installed. Depending on which desktop you use, it's also always recommended to create an environmental variable <code>ANDROID_HOME</code> to point the location of unzipped or installed SDK folder.

```
$ sudo gem install calabash-android
```

You should also have command calabash-android in your path.

## iOS

For the host machine, it's recommended to have (at least) Mac OS 10.10 (Yosemite) or 10.11 (El Capitan) with Xcode 6.x or 7.x. More information about Xcode 7.x installation can be found at Apple's developer site.

To get Calabash for iOS installed, use the following command line:

```
$ gem install calabash-cucumber
```

After this, you have commands calabash-ios and cucumber in your path and 11 gems will be installed (including tilt, rack, rack-protection, sinatra, sim_launcher, command_runner_ng, CFPropertyList, run_loop, geocoder, edn, and calabash-cucumber).
If you are using Mac OS X, you could also use cURL to fetch all files to your machine:

```
$ curl -sSL https://raw.githubusercontent.com/calabash/install/master/install-osx.sh
| bash
```

This will install both calabash-android and calabash-ios, and if everything goes smoothly, you should see something as follows on your console:

```
Preparing Ruby 2.1.6-p336...
################################################################################ 100.0%
Installing gems, this may take a little while...
################################################################################ 100.0%
Preparing sandbox...
################################################################################ 100.0%
Done! Installed:
calabash-ios:       0.18.2
calabash-android:   0.6.0
Execute 'calabash-sandbox update' to check for gem updates.
Execute 'calabash-sandbox' to get started!
```



Share this eBook!

## Troubleshooting

If you are installing Calabash for the first time, you should restart the terminal after installation, as not all done during installation might not be in use. Simply restarting the bash session will do the trick. After restarting the console, you can quickly verify this by calling calabash-android or calabash-ios to see if anything gets printed on console.

Now again, depending on which desktop OS you use you might need to install specific versions of Ruby. In case with Ruby on Ubuntu or Mac please check this troubleshooting documentation.

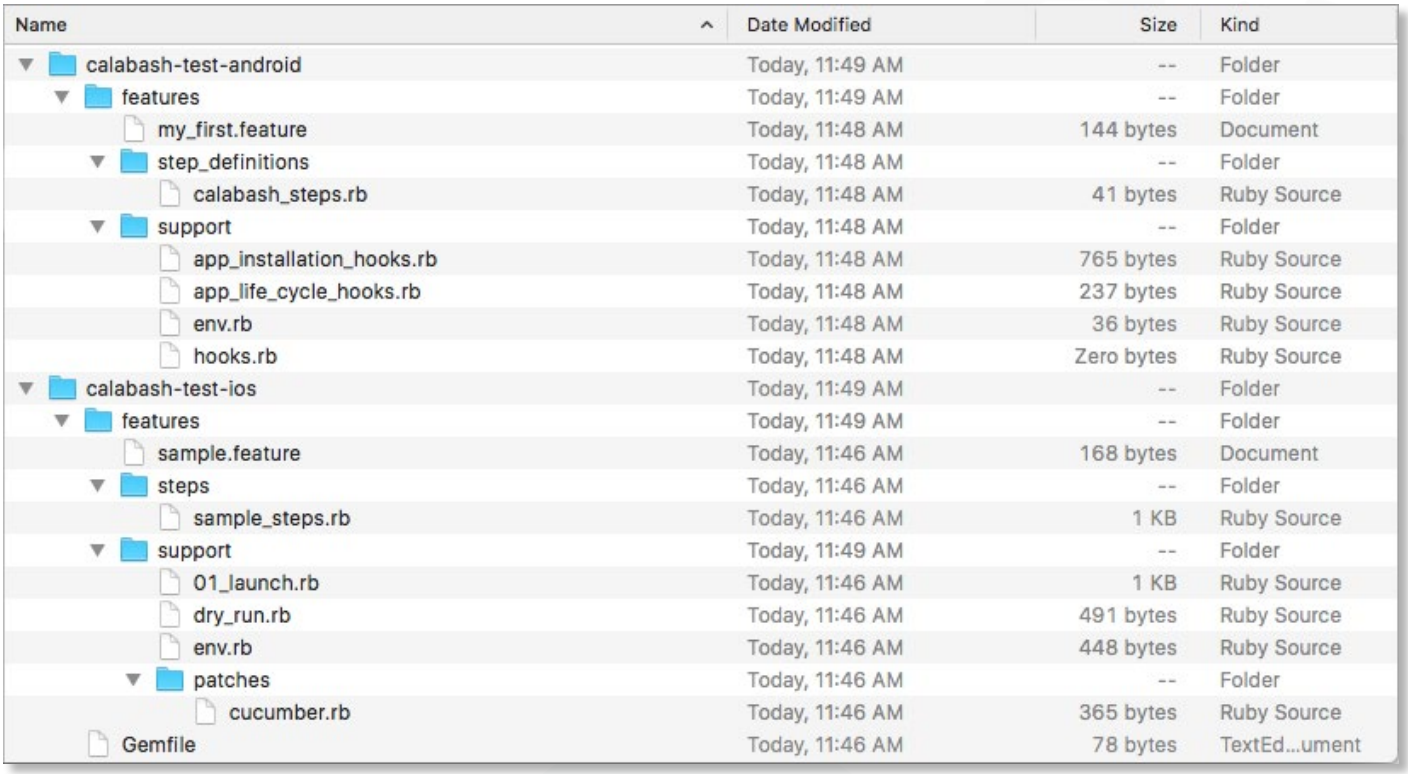
## The Basic Calabash Setup

After you have verified that the environment is properly configured, we can move to generate a basic file structure for your first Calabash test.

Create a temp directory for both Calabash Android and iOS tests:

```
$ mkdir calabash-test-android
$ cd calabash-test-android
$ calabash-android gen
...
$ cd ..
$ mkdir calabash-test-ios
$ cd calabash-test-ios
$ calabash-ios gen
```

You'll be asked few questions, plus to press enter, and as a result the following folder structure should have been created:



| Name | Date Modified | Size | Kind |
|---|---|---|---|
| calabash-test-android | Today, 11:49 AM | -- | Folder |
| features | Today, 11:49 AM | -- | Folder |
| my_first.feature | Today, 11:48 AM | 144 bytes | Document |
| step_definitions | Today, 11:48 AM | -- | Folder |
| calabash_steps.rb | Today, 11:48 AM | 41 bytes | Ruby Source |
| support | Today, 11:48 AM | -- | Folder |
| app_installation_hooks.rb | Today, 11:48 AM | 765 bytes | Ruby Source |
| app_life_cycle_hooks.rb | Today, 11:48 AM | 237 bytes | Ruby Source |
| env.rb | Today, 11:48 AM | 36 bytes | Ruby Source |
| hooks.rb | Today, 11:48 AM | Zero bytes | Ruby Source |
| calabash-test-ios | Today, 11:49 AM | -- | Folder |
| features | Today, 11:49 AM | -- | Folder |
| sample.feature | Today, 11:46 AM | 168 bytes | Document |
| steps | Today, 11:46 AM | -- | Folder |
| sample_steps.rb | Today, 11:46 AM | 1 KB | Ruby Source |
| support | Today, 11:49 AM | -- | Folder |
| 01_launch.rb | Today, 11:46 AM | 1 KB | Ruby Source |
| dry_run.rb | Today, 11:46 AM | 491 bytes | Ruby Source |
| env.rb | Today, 11:46 AM | 448 bytes | Ruby Source |
| patches | Today, 11:46 AM | -- | Folder |
| cucumber.rb | Today, 11:46 AM | 365 bytes | Ruby Source |
| Gemfile | Today, 11:46 AM | 78 bytes | TextEd...ument |

Now let's look at those files what is automatically generated for your first test foundation.

## Examples

Calabash for Android generates you my_first.feature file that is an example of a Calabash feature:

```
Feature: Login feature

      Scenario: As a valid user I can log into my app
      When I press "Login"
      Then I see "Welcome to coolest app ever"
```

The feature describes a basic behavior (scenario) or several of them in readable and easy to understand language. These 'scenarios' can be grouped together logically under the Feature definition. As in this example, Feature definitions are typically given a name and optional short description of the behavior.

The generated Android test bundle of files also contains empty file for Step Definitions in calabash_steps.rb.

The iOS example includes more details on Steps. These steps usually begin with one of the keywords: Given, When, Then, And, and But, however, they don't have to, they can use * in place of those keywords. Cucumber language does not distinguish between these keywords or *. Instead, they are meant to provide a language hint based on what will happen.

The sample.features found in the root of calabash-test-ios contain the basic example of Calabash Features and Scenario use:

```
Feature: Sample Feature

Scenario: Sample Scenario
Given the app has launched
And I have done a specific thing
When I do something
Then something should happen
```

These keywords are defined in sample_steps.rb. For example, Given looks as follows:

```
Given(/^the app has launched$/) do
 wait_for do
  !query("*").empty?
 end
end
```

And then And which includes a basic dummy example of what could go inside the given keyword. These Step definitions are somewhat like code behind these scenarios. These functions are the ones that translate all readable and human-friendly test into runnable actions.

```
And(/^I have done a specific thing$/) do
      # Example: Given I am logged in
      #  wait_for do
      #    !query("* marked:'username'").empty?
      #  end
      #
      #  touch("* marked:'username'")
      #  wait_for_keyboard
      #  keyboard_enter_text("cleveruser27")
      #
      #  touch("* marked:'password'")
      #  wait_for_keyboard
      #  keyboard_enter_text("pa$$w0rd")
```

```
#
#  wait_for_element_exists("* marked:'Login'")
#  touch("* marked:'Login'")
did_something = true

unless did_something
   fail 'Expected to have done something'
  end
end
```

This example includes some steps of how test would be progressed and what would happen in terms of test execution and progress. Naturally, these step definitions can be configured and build to be very specific for your application and test.

# CHAPTER 3

# How to Create the Right and Reusable Calabash Tests



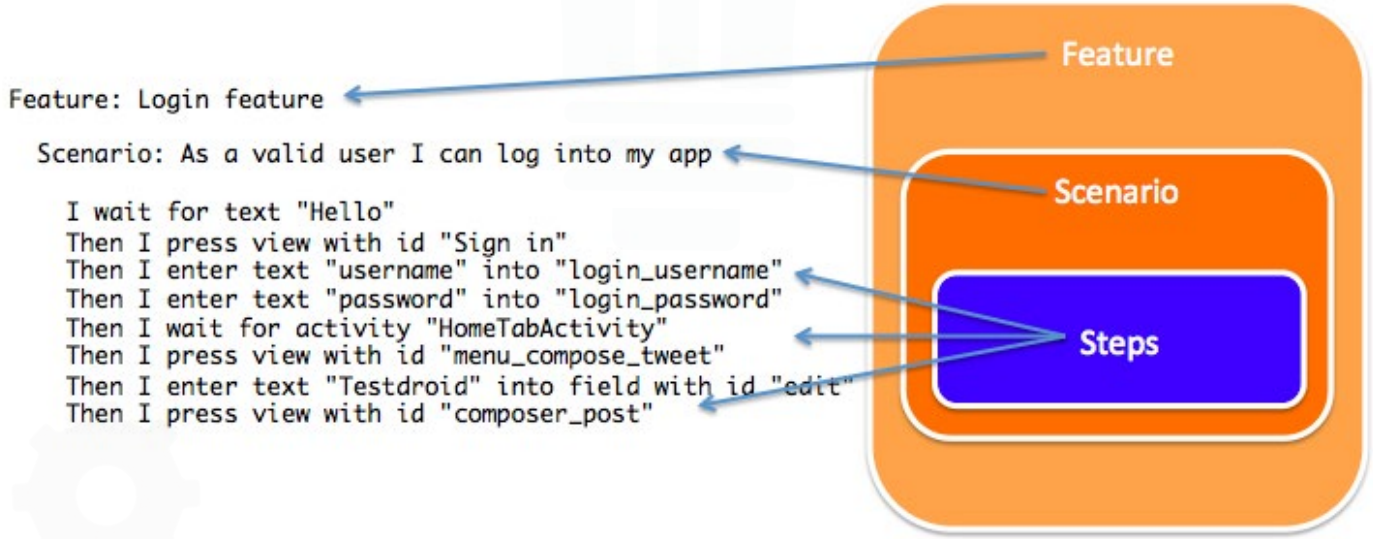
Share this eBook!

Now if you have followed the above example, you have properly generated the file structure for your first Calabash tests on Android and iOS. There are lots of different (best) practices of how to create your Calabash tests, and this chapter will present an example of basis to illustrate how these can test templates can be generated and further developed.

## Calabash Architecture for Test Creation

There are differences when it comes to Calabash running on Android and iOS. The basic test and test creation flow, however, remains the same. The recommended process to get test scripts done varies but the rule of thumb is that you first create your features. All this information can be found in example below and even example provides a basic example of feature.

Then you test your feature and execute it to see what it actually does. Then you can create those Step Definitions. Step definition is basically a Ruby code that gets executed when you run your test. The following picture illustrates the relation between Feature, Scenario and Steps.



## Scenarios

A scenario specifies a single behavior or use case within a given feature that is comprised of various Steps. For example, the following scenario describes the behavior of ensuring that a credit card input field has the correct length of digits:

```
Scenario: Run whole app
      Given my app is running
      And I touch the “Start” button
      Then I take picture
      Then I press “More Info”
```

Steps usually begin with one of the keywords Given, When, Then, And, and But, however, they don't have to, they can use * in place of those keywords. In fact, Cucumber does not distinguish among them (or *). They are instead meant to provide a language hint based on cause and effect to the stakeholders as to what is being described.

As such, simply recognizing their language implications are enough to use them effectively. However for a detailed examination of these keywords, see the Cucumber Wiki entry on them.

## Features

Feature is rarely defined by a single behavior. For this very reason, Scenarios can be grouped together logically under a Feature Definition. Feature definitions are typically given a name and an optional, short description. For example:

```
Feature: Test the entire app

Scenario: Log in to app
Given my app is running
Then I take picture
Then I use the native keyboard to enter “username@domain.com” into the
“your name” text field
Then I use the native keyboard to enter “myPassword123” into the “password” text
field
Then I press “Sign In”
```

```
Scenario: As a valid user I can start using the app
      I wait for text “Hello”
      Then I wait for activity “HomeTabActivity”
      Then I press view with id “menu_compose_tweet”
      Then I enter text “Testdroid” into field with id “edit”
      Then I press view with id “composer_post”
```

## Steps and Step Definitions

Step definitions are like code-behind for the scenarios defined in step definition scripts. They provide the glue that makes them runnable in the application. Their function is to translate readable texts into runnable actions. For example, step definitions for used example could be as follows:

```
require ‘calabash-android/calabash_steps’

Then /^I set screen to portrait$/ do
perform_action(‘set_activity_orientation’, ‘portrait’)
end

Then /^I set screen to landscape$/ do
perform_action(‘set_activity_orientation’, ‘landscape’)
end

Then /^I hide keyboard$/ do
hide_soft_keyboard()
end
```

We’ll take a look at steps “I set screen to portrait/landscape” and “I hide keyboard” examples below.

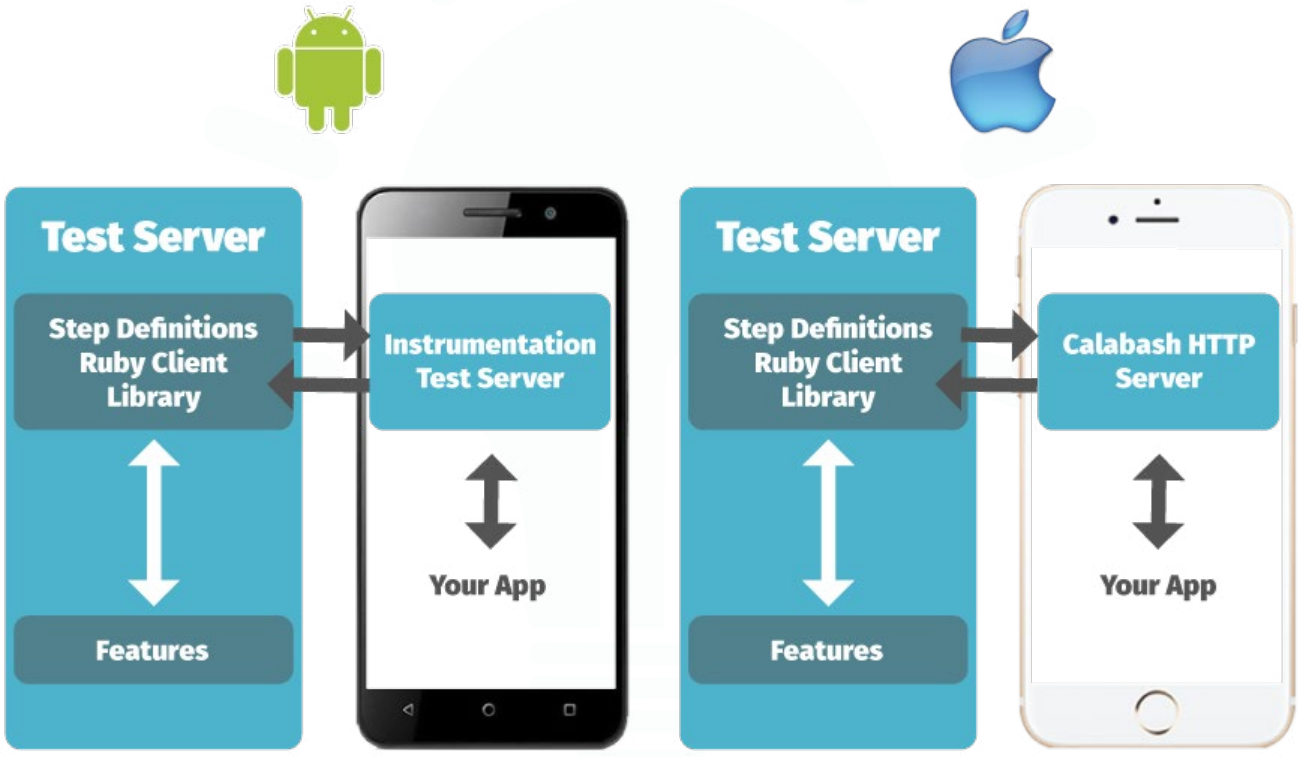
## Calabash Architecture / Infrastructure - How Calabash Works on Real Devices

The great part of Calabash is that it is a cross-platform framework and works really well for both major platforms, Android and iOS.

On Android, Calabash uses Android instrumentation test server (separated application) that is based on ActivityInstrumentationTestCase2. All Android tests are executed on the

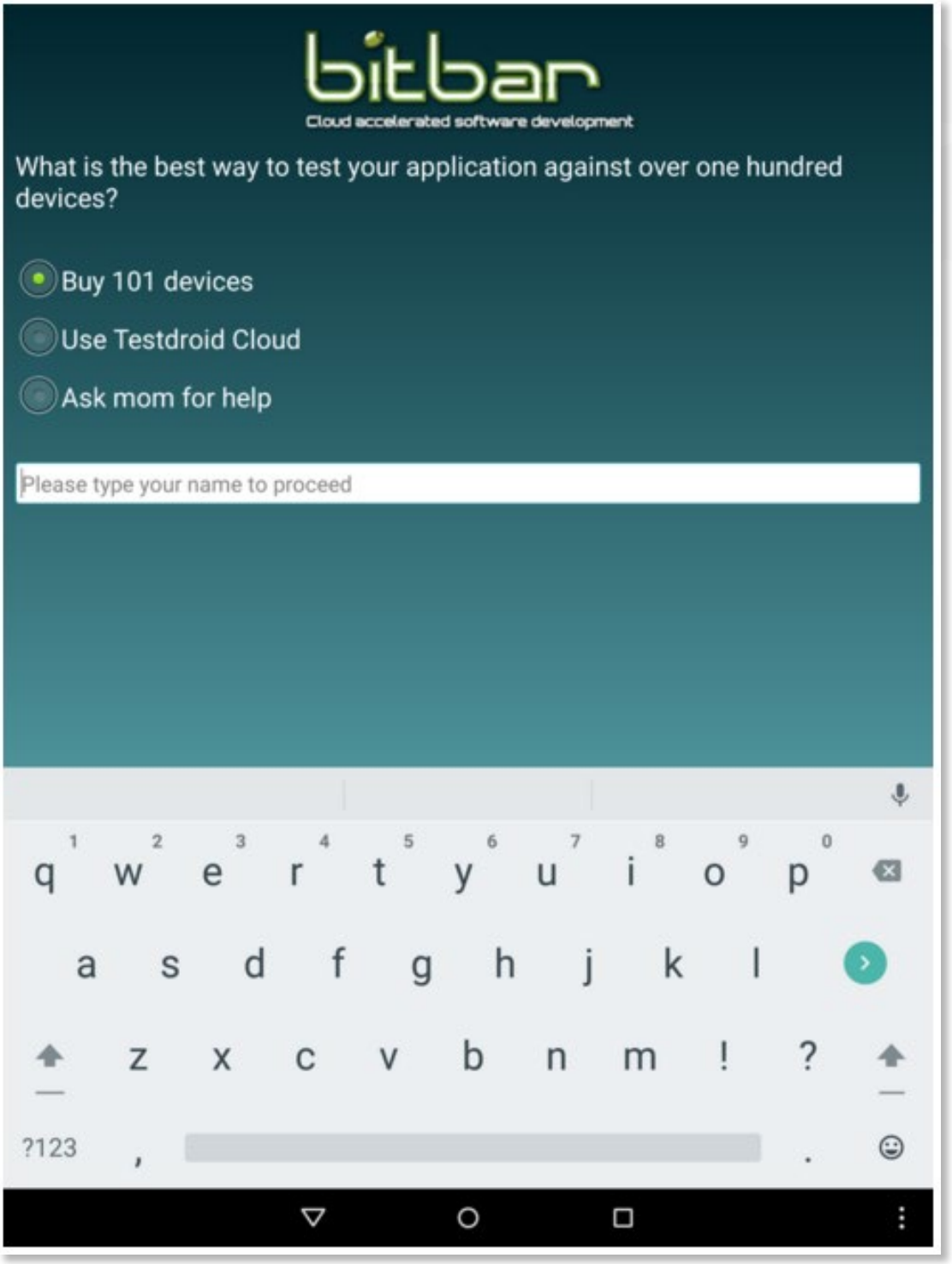
server side and users can flexibly use predefined or custom steps. Looking at the picture below, Ruby client library translates Features and commands defined for those features to instrumentation calls. If application uses Webview, the execution is injected through JavaScript.

With iOS, Calabash installs a HTTP server as an instrumentation package that then listens commands from Calabash server. Tests are also executed on server side and each test case is described in Cucumber. Slightly different than with Android, the Calabash iOS converts Cucumber commands to 'FRANK' method calls that then get executed. The webview support is done the same way than with Android.




## Getting Started with Test Creation

The BitbarSampleAppTests.zip file contains a functional test script. You can easily create similar file set for using calabash-android gen command. If you are looking for Calabash-iOS Tests you can basically use the same test package but you need to fetch the .IPA file.




As the application is relatively simple, we'll start by creating a Feature, written in Cucumber. The possible scenarios for user interaction in the above example would be that user 1) clicks through the radio buttons, 2) types a name in input field, 3) hides (or after hiding, asks to show) the keyboard, 4) changes the orientation of the screen, plus asking test to take screenshots at any given moment.

```
Feature: Test UI Components

Scenario: Test Radio Buttons
Then I press view with id “radio0”
Then I take screenshot
Then I press view with id “radio1”
Then I take screenshot
Then I press view with id “radio2”
Then I take screenshot

Scenario: Input Text and Change Orientation
Then I set screen to portrait
Then I take screenshot
Then I set screen to landscape
Then I wait
Then I enter text “Hello Calabash” into field with id “editText1”

Scenario: Test Keyboard
Then I take screenshot
Then I hide keyboard
Then I take screenshot
Then I wait
Then I show keyboard
Then I take screenshot

Scenario: Click Button
Then I press view with id “button1”
```

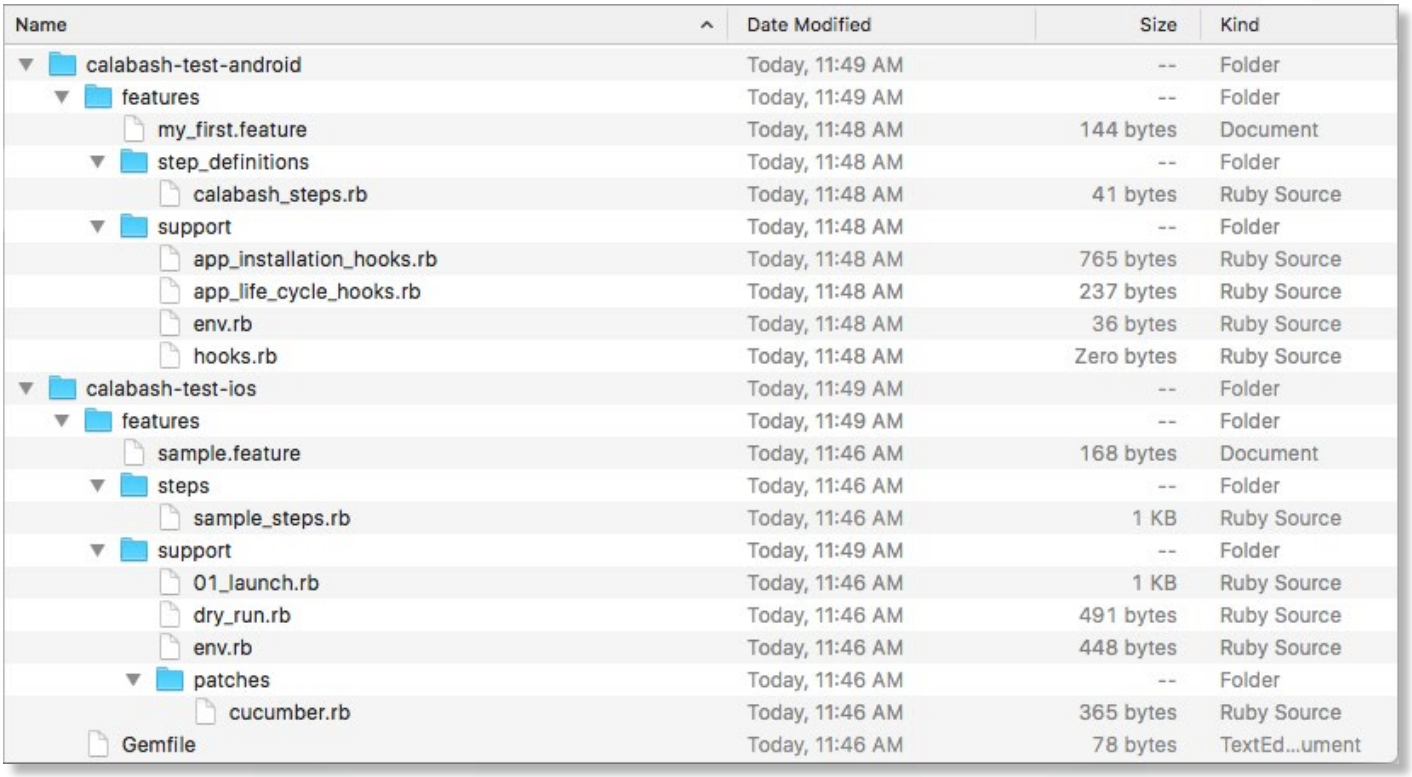
If you look at my_first.feature it basically contains the same steps as shown breakdown in the scenarios above.

## Running Calabash Tests and Interpreting the Results

The reporting and test results are important to understand how application does under the testing. These reports and results should yield all fine-grained details about the test execution, performance, and provide users actionable and rich information about tests and test runs. Furthermore, it’s important to get that test result data as real-time as possible, and those are stored based on each application regression, test executed against the specific version of app, and that you have a way to compare results based on each app and test against the older ones.

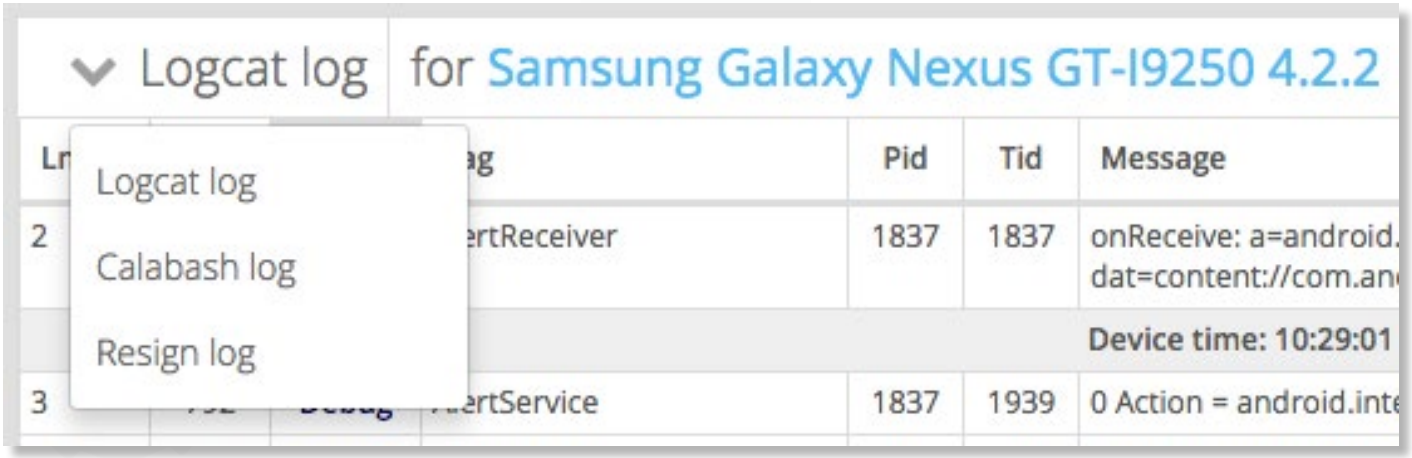
First package your test zip file to include all files that has been either generated (e.g. calabash-android gen) and modified/added while you created the test. The folder structure

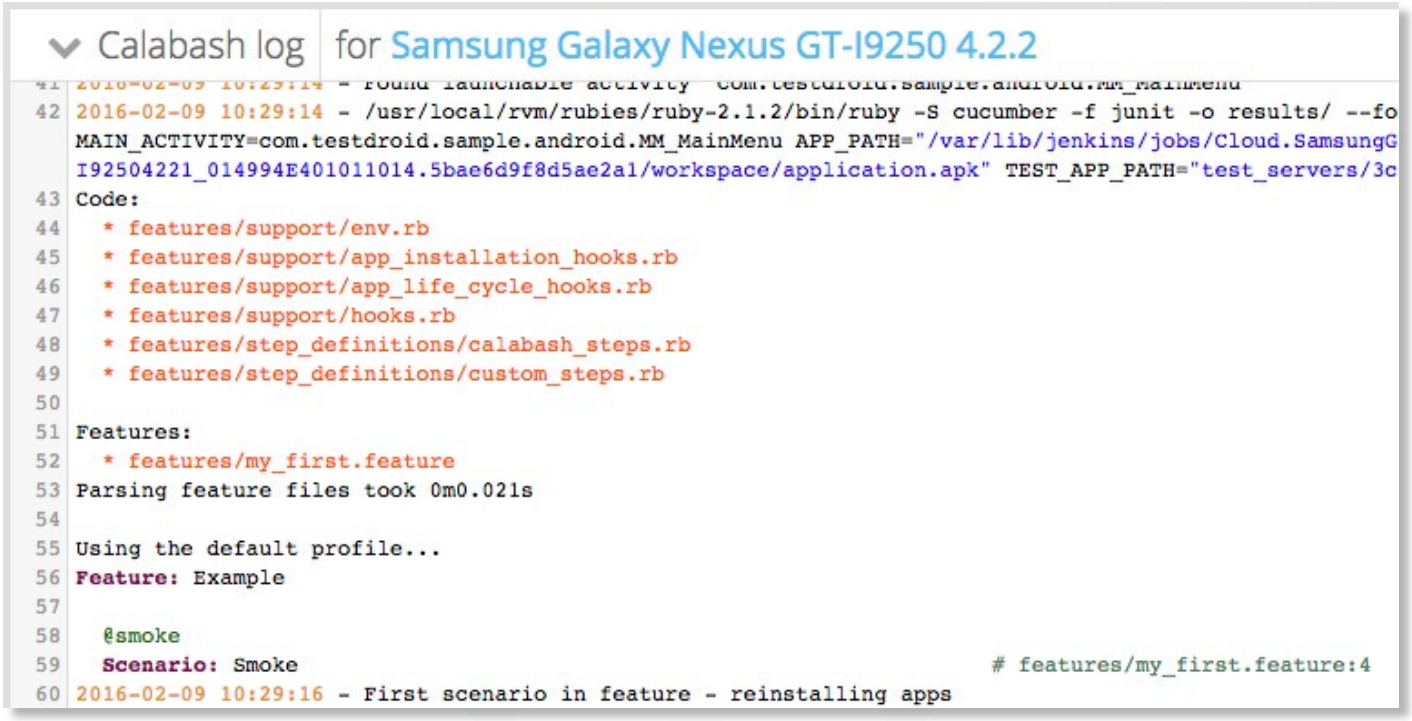
(regardless if you have added new tests, steps, features etc.) should look like this:



| Name | Date Modified | Size | Kind |
|---|---|---|---|
| calabash-test-android | Today, 11:49 AM | -- | Folder |
| features | Today, 11:49 AM | -- | Folder |
| my_first.feature | Today, 11:48 AM | 144 bytes | Document |
| step_definitions | Today, 11:48 AM | -- | Folder |
| calabash_steps.rb | Today, 11:48 AM | 41 bytes | Ruby Source |
| support | Today, 11:48 AM | -- | Folder |
| app_installation_hooks.rb | Today, 11:48 AM | 765 bytes | Ruby Source |
| app_life_cycle_hooks.rb | Today, 11:48 AM | 237 bytes | Ruby Source |
| env.rb | Today, 11:48 AM | 36 bytes | Ruby Source |
| hooks.rb | Today, 11:48 AM | Zero bytes | Ruby Source |
| calabash-test-ios | Today, 11:49 AM | -- | Folder |
| features | Today, 11:49 AM | -- | Folder |
| sample.feature | Today, 11:46 AM | 168 bytes | Document |
| steps | Today, 11:46 AM | -- | Folder |
| sample_steps.rb | Today, 11:46 AM | 1 KB | Ruby Source |
| support | Today, 11:49 AM | -- | Folder |
| 01_launch.rb | Today, 11:46 AM | 1 KB | Ruby Source |
| dry_run.rb | Today, 11:46 AM | 491 bytes | Ruby Source |
| env.rb | Today, 11:46 AM | 448 bytes | Ruby Source |
| patches | Today, 11:46 AM | -- | Folder |
| cucumber.rb | Today, 11:46 AM | 365 bytes | Ruby Source |
| Gemfile | Today, 11:46 AM | 78 bytes | TextEd...ument |

After you have created a package of it, just log in to Testdroid Cloud and upload your APK or IPA, and this zip file. If you are not familiar with the interface, there is a full-blown guide of how to use real mobile devices on cloud for testing. This guide will provide you all necessary steps to access devices, get tests executed, and finally provides you a comprehensive reports of both, test details and Calabash report.






Naturally, when your Calabash test script is more complex you probably want to check also performance aspects of the app running on specific device. The same view provides you the details of CPU consumption, memory allocation/deallocations, and even FPS metrics on how apps does.

# CHAPTER 4

# The Basics of Calabash Steps and Step Definitions



Share this eBook!

This chapter will go in depth on Calabash Steps, Step definitions and predefined steps that are set to be default with Calabash installations. There are few tips and tricks that are extremely handy and useful when it comes to using additional tools to help creating Calabash test scripts for your mobile app – and inspecting what should be put in Ruby code.

## Steps, Step Definitions and Predefined Steps

In mobile test automation user interactions are in the epicenter and drive the behavior and progress of the app execution. User naturally interacts with UI components, visual assets, that enable the control of the app. To get better understanding of how Calabash drives user interactions on these, we'll take a look at few different methods to interact with an application, for instance, gestures on regular UI elements and interactions on those (e.g. pinch, zoom, swipe).

The UI hierarchy is pretty trivial for developers who build the app, but for testers there are some inspection tools that can be very useful to further inspect on which types of characteristics these UI elements contain. For instance, what layers, fragments and other implementations may be difficult to inspect, but there are good tools that can be used to identify regular native UI components (e.g.Appium Inspector and uiautomatorviewer).

These types of inspection tools can help to find the name, description, value other useful attributes about the UI elements and objects. Let's look at some basics of the step definitions and how those would be described in Ruby code.

## Button Interactions

First of all, pressing a button (both on Android or iOS) is a basic procedure that test script and implementation must handle. For example, when user presses a back button, the script and implementation could look like something as follows

```
Then I go back
```

Naturally, there are some reserved buttons for both Android and iOS (e.g. Back, Home, Enter) that can then use perform_action function call to send the key press details:

### For Android:

```
Then /^I go back$/ do
  press_back_button
end

Then /^I press the enter button$/ do
  perform_action('send_key_enter')
end
```

### For iOS:

```
Then /^I go back$/ do
  touch("navigationItemButtonView first")
  sleep(DELAY_IN_SECONDS)
end
```

Another example with Radio buttons would be very similar, despite that the nature of Radio button is different (on/off) and it contains information whether button is already checked. In this example we could use the same syntax but the implementation part would use another Ruby call:

```
Then I press view with id "radio0"
```

The "radio0· is an ID of the Android UI component that test must tap on. The implementation using tap_when_element_exists would click the UI element when it is available/ready for a click, and the implementation would go as follows:

```
Then /^I press view with id "([^\"]*)"$/ do |id|
  tap_when_element_exists("* id:'#{id}'")
end
```

Now, it's slightly different how you do (implementation) this with iOS. The script itself would look the same, but the implementation part would be different:

```
Then /^I (?:press|touch) the "([^\"]*)" button$/ do |name|
      touch("Button touched: '#{name}'")
      sleep(DELAY_IN_SECONDS)
end
```

With iOS press/tap is called 'touch' but this would still use the name as a definition of which button is going to get pressed.

Other possible scenarios for touch would be a swipe where coordinates are given:

```
Then /^I (?:press|touch) on screen (\d+) from the left and (\d+) from the
top$/ do |x, y|
  touch(nil, { offset: { x: x.to_i, y: y.to_i } })
  sleep(DELAY_IN_SECONDS)
end
```

For more information on specific tap/click/press/touch variants and how to configure those can be found in Calabash step definitions.

## Entering Text into Text Field

For any use case where test script needs to enter text into text fields (e.g. login credentials or regular EditText elements) Calabash provides predefined steps out of the box. However, these can be easily tweaked to fit with your app. For instance:

```
Then I enter "Testing text input..." as "EditText1"
```

The script would enter "Testing text input..." as content into 'EditText1· UI element, and the implementation would go as follows:

```
Then /^I enter "([^\"]*)" as "([^\"]*)"$/ do |text, field|
   enter_text("android.widget.EditText {field LIKE[c] '#{field}'}", text)
end
```

The same could be applied to IDs of UI elements:

```
Then /^I enter text "([^\"]*)" into field with id "([^\"]*)"$/ do |text, id|
enter_text("android.widget.EditText id:'#{id}'", text)
end
```

If you are interested to find out more about UI elements and inspect those – for example with Android – you can use uiautomatorviewer to get quickly all relevant information for your script. As default, uiautomatorviewer is installed with Android Studio.

## Gestures

Gestures are natural for users with mobile devices and Calabash supports those as well. Let's take a swipe as an example. The test would call it as follows:

```
Then I swipe right
```

The implementation part is slightly different for Android and iOS, and in addition there are few other navigation/gesture implementation differences between these two platforms:

## For Android:

```
Then /^I swipe right$/ do
  perform_action('swipe', 'right')
end
```

## For iOS:

```
Then /^I swipe (left|right|up|down)$/ do |dir|
  swipe(dir)
  sleep(STEP_PAUSE)
end
```

The implementation for swipe with iOS looks more flexible as you only need one function to set the swipe direction. However, another example with scroll would be vice-versa when it comes to implementation:

```
Then I scroll up
```

## For Android:

```
Then /^I scroll down$/ do
scroll_down
end
```

### For iOS:

```
Then /^I scroll (left|right|up|down)$/ do |dir|
scroll("scrollView index:0", dir)
sleep(STEP_PAUSE)
end
```

## Waiting and Asserts

First, the classic example with waiting/delaying test execution on Calabash:

```
Then I waitThis basically gives 2 seconds of delay as stated in implementation:
Then /^I wait$/ do
sleep 2
end
```

Second example makes the test execution to wait until certain label/text is shown on the screen:

```
Then I wait to see "Hello!"
The same method can be also used to track if certain UI element (e.g. button) is
visible on the screen. The implementation goes as follows:
```

### For Android:

```
Then /^I wait to see "([^\"]*)"$/ do |text|
  wait_for_text(text)
end
```

### For iOS:

```
Then /^I wait to see "([^\"]*)"$/ do |expected_mark|
  wait_for(TIMEOUT) { view_with_mark_exists( expected_mark ) }
end
```

The Wait-To-See calls are definitely better than regular sleep as this can synchronize the test execution. As said, Waits can be also applied to certain elements, and here is an example of waiting implementation until Android button is visible:

```
Then /^I wait for the “([^\”]*)” button to appear$/ do |identifier|
  wait_for_element_exists(“android.widget.Button marked:’#{identifier}’”);
end
```

Asserting is very similar to waiting and cucumber language changes from “to see” to “should see”. For example:

```
Then I should see “Hello!”
```

This step checks the view for provided text on screen. If Calabash cannot find the identified text on the screen, the test step will fail.

### For Android:

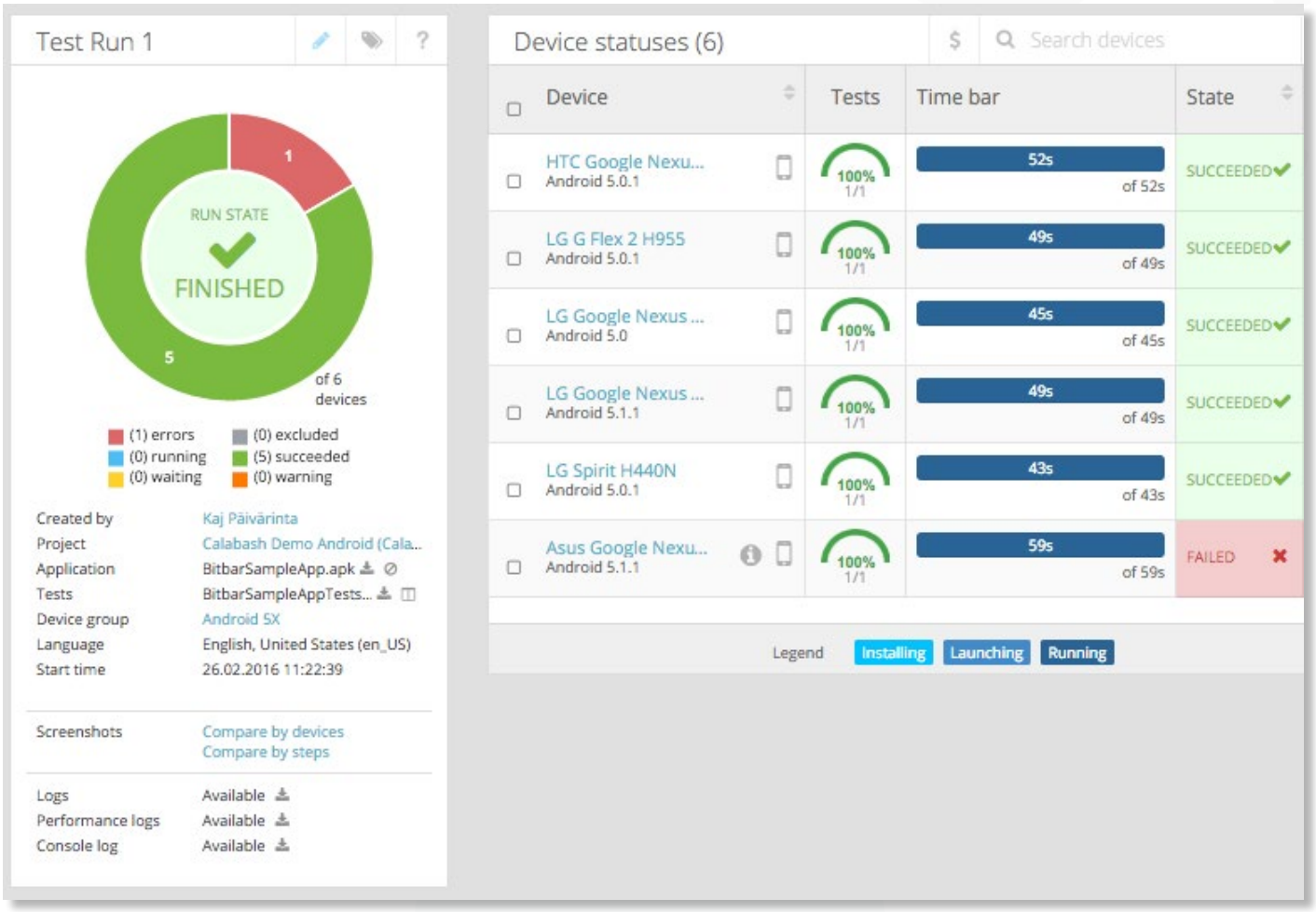
```
Then /^I should see “([^\”]*)”$/ do |text|
  wait_for_text(text, timeout: 5)
end
```

### For iOS:

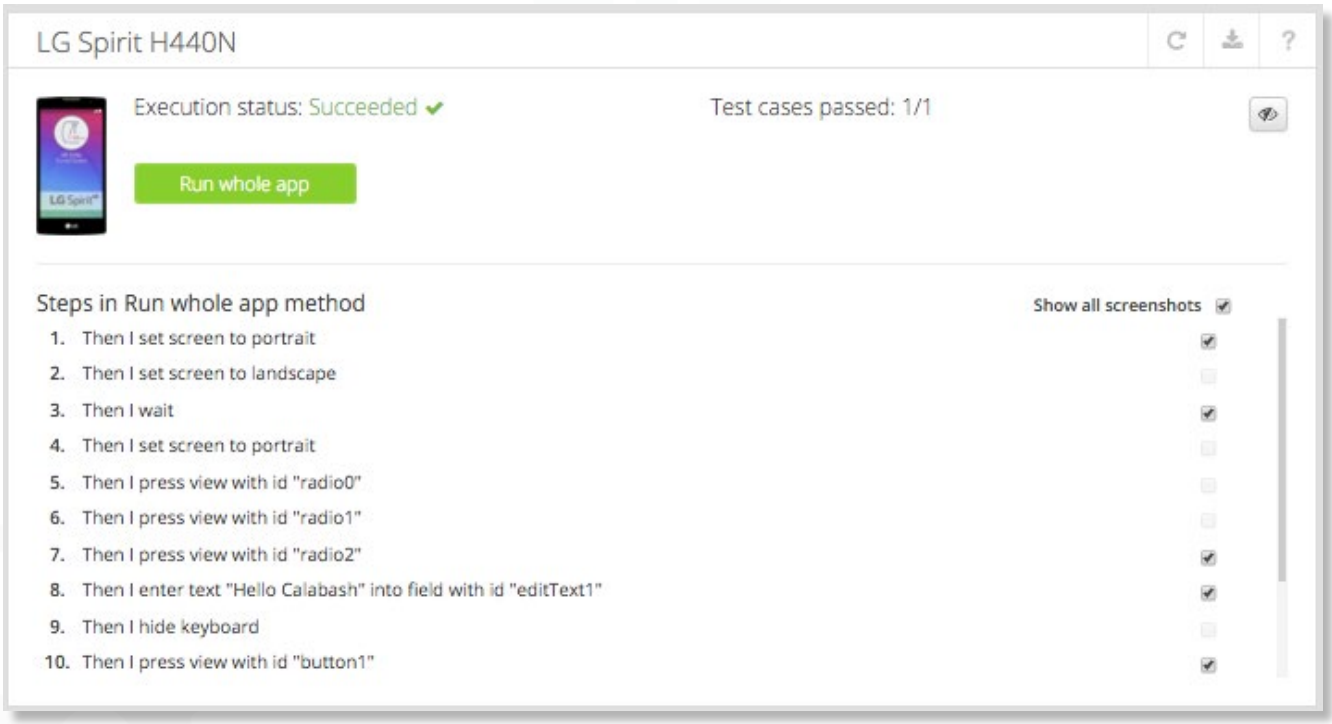
```
Then /^I should see “([^\”]*)”$/ do |expected_mark|
   res = (element_exists( “view marked:’#{expected_mark}’” ) or element_exists(
“view text:’#{expected_mark}’”))
  if not res
      screenshot_and_raise “No element found with mark or text: #{expected_mark}”
  end
end
```

## How Test Steps and Step Definitions Show Up

The beauty of Calabash is that it really works well for both Android and iOS. Despite of minor differences in Ruby implementations and sometimes with Calabash versions, we’re here to help you to run your Calabash tests smoothly on our devices. When you log in to Testdroid Cloud and upload your application plus tests, you’ll see the project overview of your test runs:




If you run a Calabash test in Testdroid Cloud you get all the fine-grained details of Calabash steps under each test run. Just click the specific device run open (click the device name or anything on that line) and you'll get the very same steps shown in the device run view:



The UI shows you Calabash test steps and by clicking “Show screenshots” you’ll be able to see what happened with the test execution during each step with the app. Furthermore, you’ll also get the specific CPU and memory consumption details of the test run which provides an excellent glance on what should be focused in performance optimization as well as memory allocations/deallocations.

Finally, Calabash test runs done in Testdroid Cloud also provide you Calabash log which can be used to further investigate behavior of test script and what happened in execution.

```
2016-02-26 10:24:16 - JDK found on PATH.
2016-02-26 10:24:16 - Android SDK found at: /var/lib/jenkins/android-sdk-linux
No test server found for this combination of app and calabash version. Recreating test server.
2016-02-26 10:24:16 - Signature files:
2016-02-26 10:24:16 - /tmp/d20160226-3262-6wlivy/META-INF/CERT.RSA
2016-02-26 10:24:16 - "/usr/bin/keytool" -v -printcert -J"-Dfile.encoding=utf-8" -file "/tmp/d20160226-3262-6wlivy/META-INF/CERT.RSA"
2016-02-26 10:24:16 - MD5 fingerprint for signing cert (/var/lib/jenkins/jobs/Cloud.LGSpiritH440N_LGH440ne4398081.5bae6d9f8d5ae2a1/workspace/application.apk):
F8:59:5A:4E:C4:90:6F:2A:65:7F:4C:8E:CD:83:A0:E3
2016-02-26 10:24:16 - /var/lib/jenkins/jobs/Cloud.LGSpiritH440N_LGH440ne4398081.5bae6d9f8d5ae2a1/workspace/application.apk was signed with a certificate with
fingerprint F8:59:5A:4E:C4:90:6F:2A:65:7F:4C:8E:CD:83:A0:E3
2016-02-26 10:24:16 - Reading keystore data from keystore file '/var/lib/jenkins/.android/debug.keystore'
2016-02-26 10:24:16 - "/usr/bin/keytool" -list -v -alias androiddebugkey -keystore /var/lib/jenkins/.android/debug.keystore -storepass android "-J"-
Dfile.encoding=utf-8"" "-J"-Duser.language=en-US""
2016-02-26 10:24:16 - Key store data:
2016-02-26 10:24:16 - Alias name: androiddebugkey
Creation date: Oct 7, 2011
Entry type: PrivateKeyEntry
Certificate chain length: 1
Certificate[1]:
Owner: CN=Android Debug, O=Android, C=US
Issuer: CN=Android Debug, O=Android, C=US
Serial number: 4e8ea420
Valid from: Fri Oct 07 09:02:56 CEST 2011 until: Sun Sep 29 09:02:56 CEST 2041
Certificate fingerprints:
     MD5:  F8:59:5A:4E:C4:90:6F:2A:65:7F:4C:8E:CD:83:A0:E3
     SHA1: 8D:F5:93:C1:B6:EA:A6:EA:DA:DC:E3:68:31:FE:82:B0:8C:AC:8D:74
     Signature algorithm name: SHA1withRSA
     Version: 3
2016-02-26 10:24:16 - Fingerprint: F8:59:5A:4E:C4:90:6F:2A:65:7F:4C:8E:CD:83:A0:E3
2016-02-26 10:24:16 - Signature algorithm name: SHA1withRSA
2016-02-26 10:24:16 - Unlocked keystore at /var/lib/jenkins/.android/debug.keystore - fingerprint: F8:59:5A:4E:C4:90:6F:2A:65:7F:4C:8E:CD:83:A0:E3
2016-02-26 10:24:16 - Trying to read keystore from: /var/lib/jenkins/jobs/Cloud.LGSpiritH440N_LGH440ne4398081.5bae6d9f8d5ae2a1/workspace/debug.keystore - no such
file
2016-02-26 10:24:16 - Trying to read keystore from: /var/lib/jenkins/.local/share/Xamarin/Mono\ for\ Android/debug.keystore - no such file
2016-02-26 10:24:16 - Trying to read keystore from: /var/lib/jenkins/AppData/Local/Xamarin/Mono for Android/debug.keystore - no such file
2016-02-26 10:24:16 - Found tools directory at '/var/lib/jenkins/android-sdk-linux/build-tools/19.0.0'
2016-02-26 10:24:16 - Found tools directory at '/var/lib/jenkins/android-sdk-linux/build-tools/19.0.0'
2016-02-26 10:24:16 - Found tools directory at '/var/lib/jenkins/android-sdk-linux/build-tools/19.0.0'
```

# CHAPTER 5

# Combine Cucumber with Other Mobile Test Automation Frameworks



Share this eBook!

Behaviour-Driven Development (BDD) gathers supporters for many obvious reasons as it is readable (by everyone), understandable (across teams and functions) and accountable (metric-driven for success/failures). Cucumber is a great example of optimized tool/framework to create such tests. This allows basically anyone (developers, QA, testers etc.) to write specifications in any spoken language that then gets executed as automated tests against mobile apps. Furthermore, this sort of BDD approach incorporates the main standards of test-driven development (TDD) and provides a shared view and process for collaboration between all stakeholders.

## Cucumber for Test Automation

Cucumber is a neat tool that developers can use to test their stuff. In mobile context, Cucumber is also usable and for example is used with Calabash. Basically, Cucumber can run automated acceptance tests written in a behavior-driven development (BDD) style and is available with Ruby. In addition, there are other Cucumber projects available for some other platforms/programming languages, but we'll take a look at Ruby side of things here.

One of the best parts of Cucumber is that it also provides documentation and feature specification for mobile app testing. When the test script is written in an understandable language, it can also serve as a design, development and test specification – and that's what makes Cucumber highly usable across functions.

Cucumber can be also used to enable a large-scale test automation with real mobile devices. There are myriad of different test automation frameworks that can get you there instantly, but also combined together with Cucumber. Cucumber behavior-driven development together with one of the best test automation frameworks today – Appium – is a great combination and can help to you automate testing of your apps, write nicely documented specifications and make clarity for test execution.

## Prerequisites and Installations

First and foremost, check that your Ruby environment is up-to-date, by using the following terminal line:

```
$ ruby -v
```

Things may work pretty well on some older Ruby versions, but to get all the best features and the most stable experience, we recommend using the latest (e.g. 2.2.4 or newer).
RVM – Ruby Version Manager

In case you don't have or have old version of Ruby or you want to use multiple versions of Ruby, RVM (Ruby Version Manager) is a neat command-line tool which allows quick and easy installations, management of multiple versions and Ruby environments with interpreters and any set of gems. You can simply get it install with these instructions:

```
$ curl -sSL https://get.rvm.io | bash -s stable
$ rvm install ruby
```

List of all installed versions of Ruby and set one of those as default one:

```
$ rvm list
$ rvm --default use 2.1.1
```

The upgrade of RVM can be done with the following lines:

```
$ rvm get head
$ rvm autolibs homebrew
$ rvm install ruby
```

Note that this takes care of Homebrew installation as well. Another alternative to install Homebrew is done as follows:

```
$ ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

After setting up Homebrew you can (and you should) check every now and then if the environment is up to snuff:

```
$ brew doctor
```

This should give you Your system is ready to brew but Your Homebrew is outdated is also very common and then keeping Brew up-to-date can be simply done with an update:

```
$ brew update
```

Again, it's worth of checking that it's Ruby is properly installed by printing out the Ruby version:

```
$ ruby -v
```

Update RubyGems and Bundler:

```
$ gem update --system
$ gem install --no-rdoc --no-ri bundler
$ gem update
$ gem cleanup
Check that RubyGems is >= 2.1.5


$ gem --version
```

## Platform-Specific Tools

Depending on which platform – OS X, Linux or Windows – you're about to run your tests, you need to setup all required development tools (Xcode for iOS, Android Studio) and other stuff. For detailed instructions on how to setup Appium for any platform, with desired programming language and all dependencies, take a look at our Appium tutorial.

## Cucumber Installation

Setting up Cucumber is pretty straightforward.

Create a directory for test example:

```
$ mkdir cucumber-test
cd cucumber-test
```

Then create a file under this folder to include the following lines:

```
source "https://www.rubygems.org"
gem "cucumber"
gem "selenium-webdriver"
gem "rspec"
gem "rspec-expectations"
```

Now, run bundler at the Ruby example directory to install dependencies:

```
$ bundle install
```

Now, we have installed everything that it takes to start writing tests and running those. For detailed information on how to write Cucumber tests, you can check from our tutorial. This tutorial gives instructions to write Calabash tests but overall the syntax is the same and tests, steps and features are done the same way.

# Conclusion

Although Appium has been in tremendous growth, there is no reason to forget Calabash. Both of them provide great foundation for cross-platform testing strategy. Calabash can be used to write automated functional and acceptance tests using any Ruby-based test framework. And the good thing with Calabash is that it supports Cucumber. And Cucumber lets you express the behavior of your app using natural language that can be understood by business experts and non-technical QA staff.

At Testdroid, we provide Testdroid Cloud on which we host thousands of real devices with unique models to provide you the broadest device selection and it also enables to run unlimited concurrent Calabash tests on all of the devices.


If interested, you can contact us at sales@bitbar.com and we'll get back to your as soon as possible to tell you more about what Testdroid Cloud has to offer.