Appium Workshop

2016 Selenium Conference
Bangalore, India

Install Dependencies

- Windows: https://github.com/isonic1/appium-workshop/blob/master/Appium%20Windows%20Installation%20Instructions.md
- Mac: https://github.com/isonic1/appium-workshop/blob/master/Appium%20Mac%20Installation%20Instructions.md
- Validate everything is installed correctly by running the test example listed the instructions below.
    - https://github.com/isonic1/appium-workshop

What is Appium?

Appium is an open-source tool for automating native, mobile web, and hybrid applications on iOS and Android platforms. Native apps are those written using the iOS or Android SDKs. Mobile web apps are web apps accessed using a mobile browser (Appium supports Safari on iOS and Chrome or the built-in 'Browser' app on Android). Hybrid apps have a wrapper around a "webview" -- a native control that enables interaction with web content. Projects like Phonegap, make it easy to build apps using web technologies that are then bundled into a native wrapper, creating a hybrid app.

Importantly, Appium is "cross-platform": it allows you to write tests against multiple platforms (iOS, Android), using the same API. This enables code reuse between iOS and Android testsuites.

Appium Philosophy

- Appium was designed to meet mobile automation needs according to a philosophy outlined by the following four tenets:
- You shouldn't have to recompile your app or modify it in any way in order to automate it.
- You shouldn't be locked into a specific language or framework to write and run your tests.
- A mobile automation framework shouldn't reinvent the wheel when it comes to automation APIs.
- A mobile automation framework should be open source, in spirit and practice as well as in name!

Appium is at its heart a webserver that exposes a REST API. It receives connections from a client, listens for commands, executes those commands on a mobile device, and responds with an HTTP response representing the result of the command execution. The fact that we have a client/server architecture opens up a lot of possibilities: we can write our test code in any language that has a http client API, but it is easier to use one of the Appium client libraries. We can put the server on a different machine than our tests are running on. We can write test code and rely on a cloud service like Sauce Labs to receive and interpret the commands.

Source: http://appium.io/introduction.html

History of Appium

Dan Cuellar was the Test Manager at Zoosk in 2011, when he encountered a problem. The length of the test passes on the iOS product was getting out of hand. Less testing was an option, but would come with additional risk, especially with it taking several days to get patches through the iOS App Store Review process. He thought back to his days working on websites and realized automation was the answer.

Dan surveyed the existing landscape of tools, only to find that all of them hand major drawbacks. The tool supplied by Apple, UIAutomation, required tests to be written in JavaScript, and did not allow for real time debugging or interpretation. It also had to be executed inside the Xcode profiling tool, Instruments. Other 3rd-party tools used private APIs and required SDKs and HTTP Servers to be embedded into the application. This seemed highly undesirable.

Unsatisfied with the existing options, Dan asked his manager for some additional time to see if he could find a better way. He spent 2 weeks poking and prodding around to see if there was a way to use approved Apple technologies to automate an iOS application. The first implementation he tried used AppleScript to send messages to Mac UI elements using the OS X accessibility APIs. This worked to some degree, but would never work on real devices, not to mention other drawbacks.

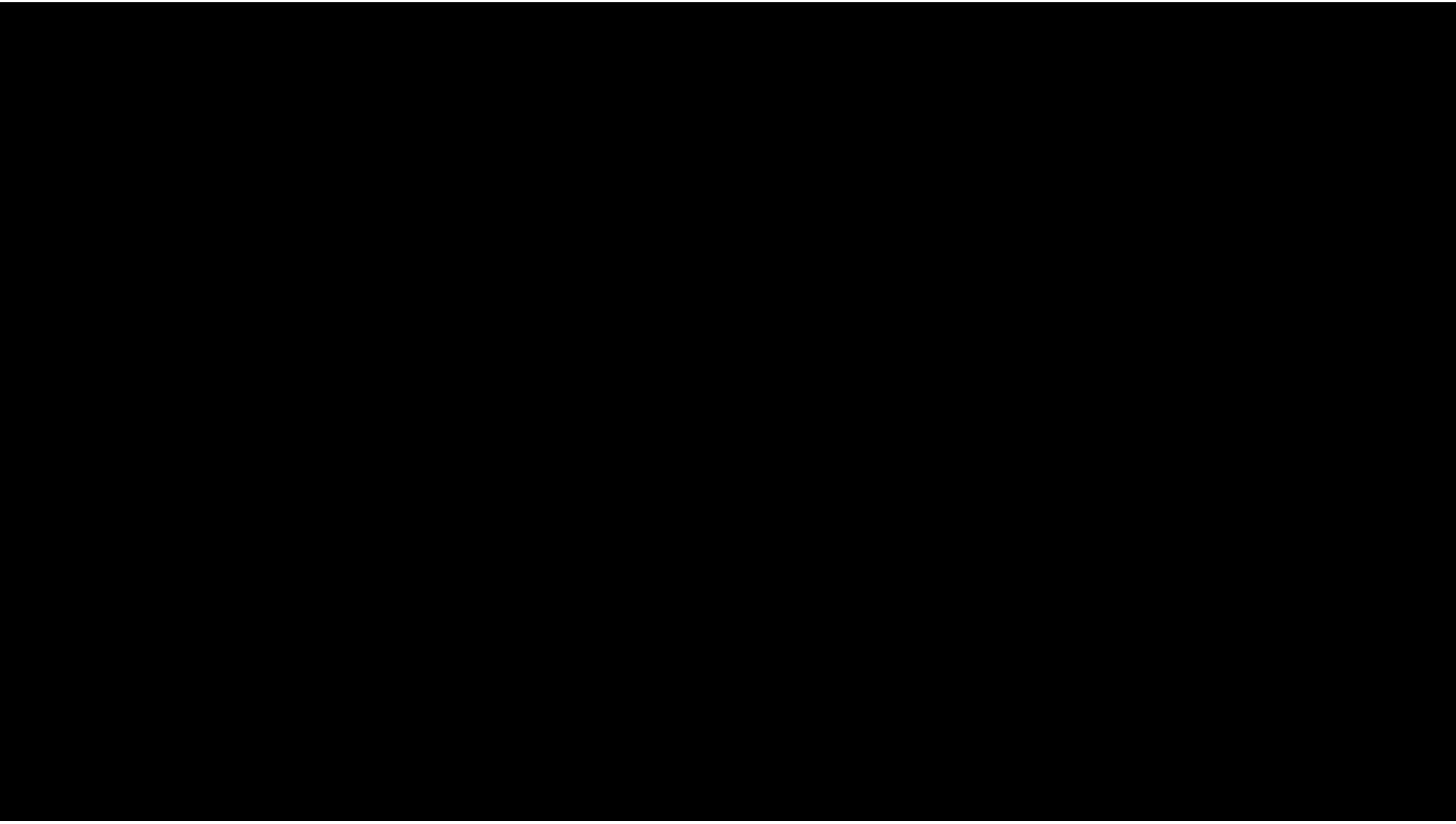
So he thought, what if I could get the UIAutomation framework to run in real time like an interpreter? He looked into it and he determined that all he would need to do is find a way to receive, execute, and reply to commands from within a UIAutomation javascript program. Using the utility Apple provided for executing shell commands he was able to cat sequentially ordered text files to receive commands, eval() the output to execute them, and write them back to disk with python. He then prepared code in C# that implemented the Selenium-style syntax to write the sequentially ordered javascript commands. iOSAuto is born.

Selenium Conference 2012: On the second day of the conference, Dan stepped up on stage to give the lightning talk. Jason Huggins, co-creator of Selenium, moderated the lightning talks. Dan experienced technical difficulties getting his presentation to load, and Jason nearly had to move on to the next lightning talk. At the last moment, the screen turned on and Dan jumped into his presentation. He explained the details of his implementation and how it worked, begged for contributors, and in five minutes it was over. The crowd applauded politely, and he left the stage.

Months Later: Four months after the Selenium Conference, Jason called Dan. Jason had been working on iOS testing support for a client at Sauce Labs. Jason remembered Dan's lightning talk and thought the project might be useful to Jason's work, but Dan's source code was not public. Jason asked Dan to meet up. Later that week, Dan met Jason in a bar in San Francisco and showed him the source code for iOS Auto. Jason encouraged Dan to change the language to make the project more appealing to potential contributors. Dan uploaded a new version in Python. In September, Jason added a web server and began to implement the WebDriver wire protocol over HTTP, making iOS Auto scriptable from any Selenium WebDriver client library in any language.

The Mobile Testing Summit: Jason decided that the project should be presented at the Mobile Testing Summit in November, but suggested that the project get a new name first. Many ideas were thrown out and they settled on AppleCart. A day later, while he was perusing some of Apple's guidance on copyright and trademarks, Jason noticed that under the section of examples for names Apple would defend its trademarks against, the first example was "AppleCart". He called Dan and informed him of the situation, and they brainstormed for a bit before Jason hit the jackpot. Appium... Selenium for Apps.

Appium Introduction Video



Credit: Dan Cuellar

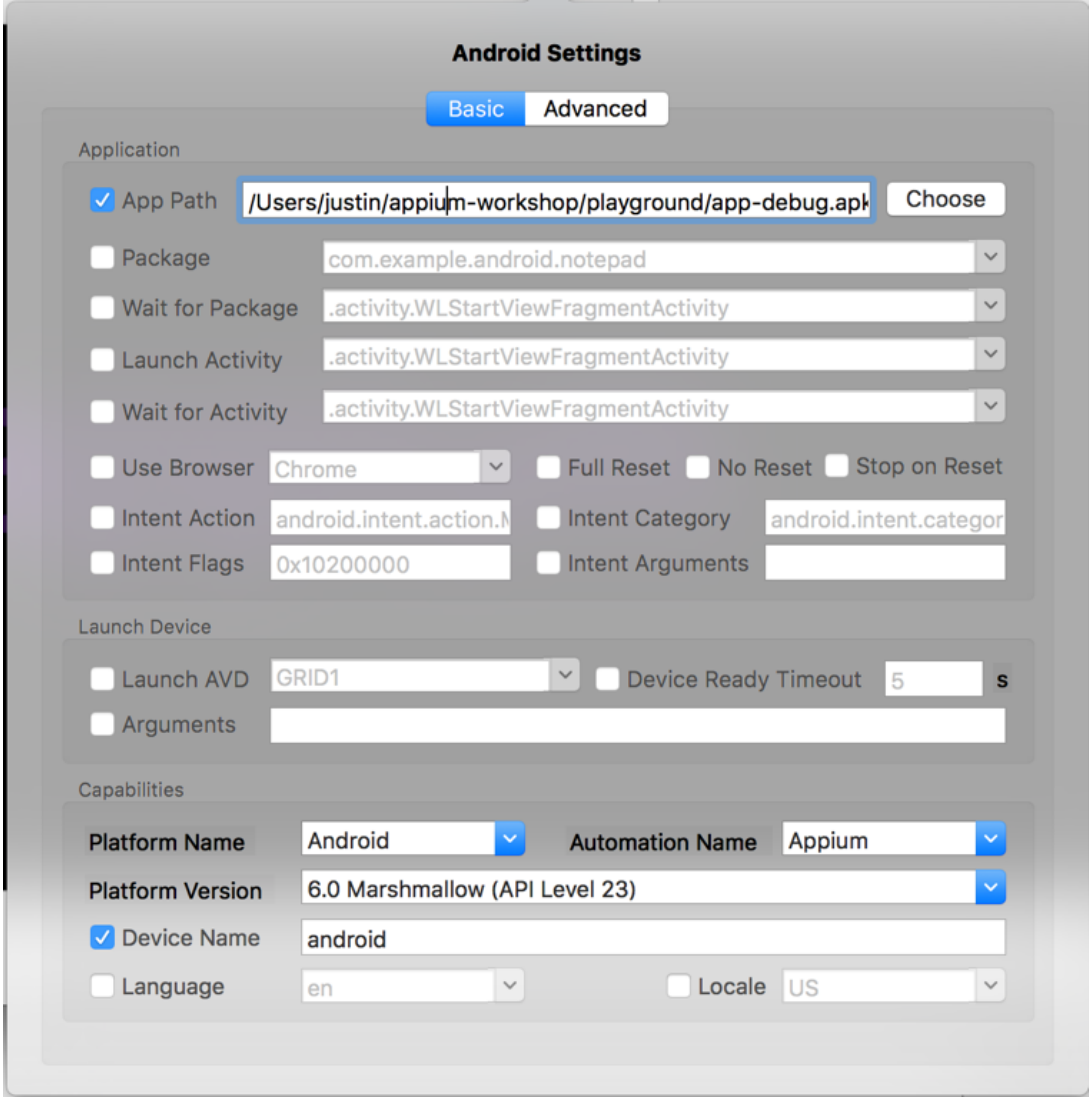
Before We Start

- Download, clone or pull on the appium-workshop repo to get the latest updates.
    - https://github.com/isonic1/appium-workshop
- I will try to answer all of your questions. However, if do not have an answer I will try my best to find it. I'll follow up with the class later with an email to all the questions.
- Please give me feedback!
    - Please take notes during the workshop.
    - I would like to know:
        - What I can improve upon.
        - Anything that didn't make sense to you.
        - What you thought I did well.
        - What you would like me to go in-depth more.
        - etc…
- You can send these to me an email or give them to me after the workshop. :)

Lets Get Started!

Appium IDE

- Start the Appium IDE.
- Click the Android button.
- Select the Application Path checkbox.
  - Windows: C:\Users\**your_user_name**\appium-workshop\playground\app-debug.apk
  - Mac: Users/**your_user_name**/appium-workshop/example/playground/app-debug.apk
- Select the Launch AVD checkbox.
  - Select EM1 in the dropdown menu.
- Select Platform Name: Android
- Select Automation Name: Appium
- Set Platform Version: 6.0 Marshmallow (API Level 23) *Mac this looks like a non-editable field but you can edit it.
- Select the Device Name checkbox.
  - Set to: android
- Close the Android Settings.



Android Settings

Basic | Advanced

Application

- [✓] App Path: /Users/justin/appium-workshop/playground/app-debug.apk — Choose
- [ ] Package: com.example.android.notepad
- [ ] Wait for Package: .activity.WLStartViewFragmentActivity
- [ ] Launch Activity: .activity.WLStartViewFragmentActivity
- [ ] Wait for Activity: .activity.WLStartViewFragmentActivity
- [ ] Use Browser: Chrome
- [ ] Full Reset
- [ ] No Reset
- [ ] Stop on Reset
- [ ] Intent Action: android.intent.action.M
- [ ] Intent Category: android.intent.categor
- [ ] Intent Flags: 0x10200000
- [ ] Intent Arguments

Launch Device

- [ ] Launch AVD: GRID1
- [ ] Device Ready Timeout: 5 s
- [ ] Arguments

Capabilities

- Platform Name: Android
- Automation Name: Appium
- Platform Version: 6.0 Marshmallow (API Level 23)
- [✓] Device Name: android
- [ ] Language: en
- [ ] Locale: US

Appium Inspector

- Click the Play(Windows) or Launch(mac) button.
- Click the search/magnify glass button.
    - This should launch the emulator and start the app.
    - Windows: Click the Refresh button to load the page.
- So what did the Appium IDE just do?
    - It started a new appium session with the following server capabilities.
    - --platform-name "Android"
    - --platform-version "6.0"
    - --app "/Users/justin/appium-workshop/playground/app-debug.apk"
    - --avd "GRID1" --device-name "android"
    - See full list of capabilities here: http://appium.io/slate/en/master/?ruby#appium-server-capabilities
- Now explore the application with the Inspector.
    - Take note of the UI layouts, views and elements.
    - The mac client you can interact with the image to click elements. ** The windows client unfortunately cannot and is lacking many features. :(

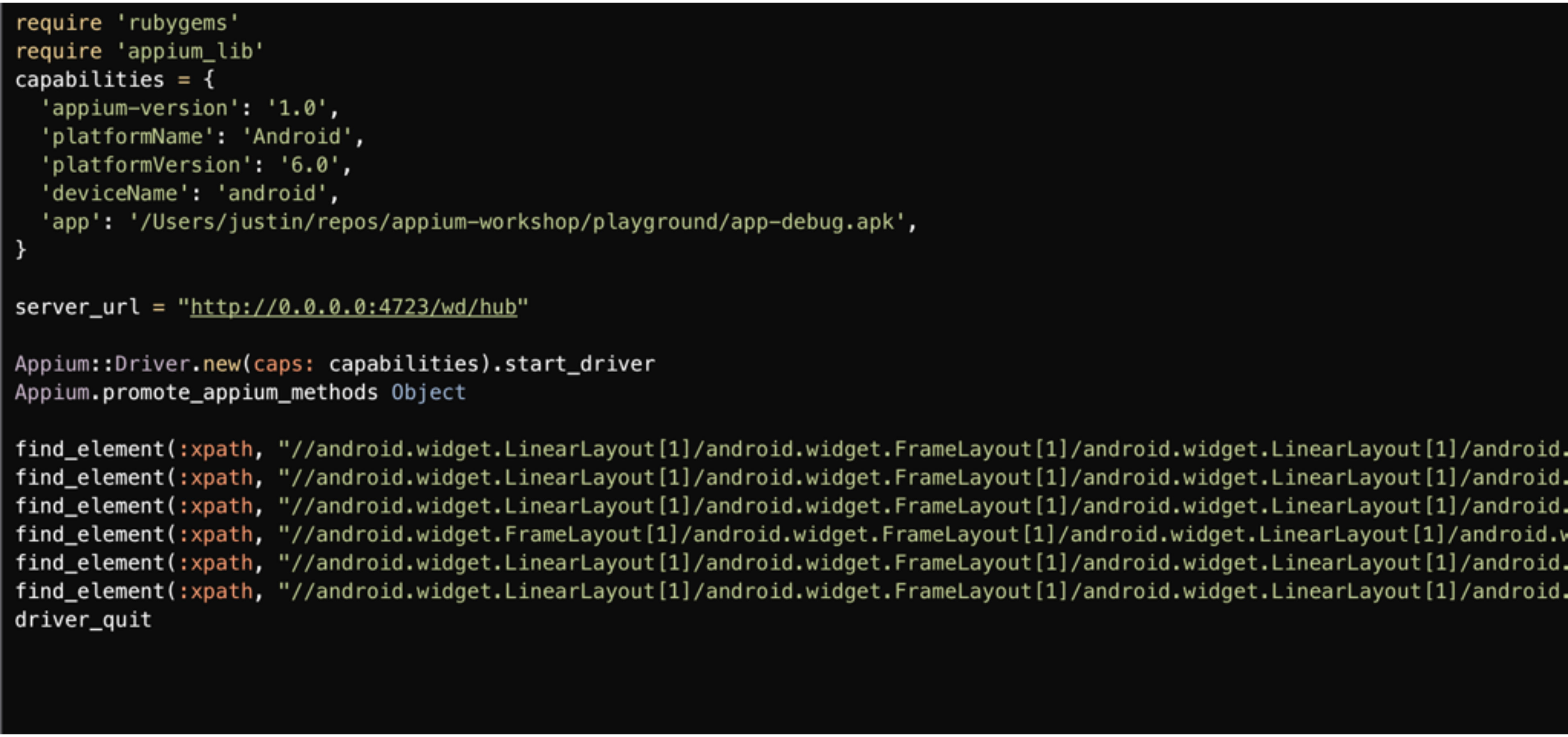
Recording a Test (Mac Only)

- Click the Record button. You should see a box slide from the bottom with the Appium boilerplate code.
- Switch the language to Ruby.
- Click the Refresh button to make sure it displays the current app page.
- Ok! Now lets create a test to trigger a Alert Popup.
- Using the inspector:
  - Tap the hamburger button.
  - Tap the Alerts menu option.
  - Tap the ALERT button.
  - Tap the OK button.
  - Tap the hamburger button.
  - Tap the Home button.
- Now click the Replay button in the recording section.
- Did the script work? probably not… Lets fix it!
- Close the Appium Inspector window and stop the Appium server.

Fixing the Test

- For Mac Users:
  - Click the Save button in the recording section.
  - Name the file “alert.rb” and save it to the playground directory in appium-workshop.
- For Windows:
  - Rename the alert_example.rb to alert.rb in the appium_workshop directory.
- Open alert.rb in Sublime editor if you haven’t already.

alert.rb



```ruby
require 'rubygems'
require 'appium_lib'
capabilities = {
  'appium-version': '1.0',
  'platformName': 'Android',
  'platformVersion': '6.0',
  'deviceName': 'android',
  'app': '/Users/justin/repos/appium-workshop/playground/app-debug.apk',
}

server_url = "http://0.0.0.0:4723/wd/hub"

Appium::Driver.new(caps: capabilities).start_driver
Appium.promote_appium_methods Object

find_element(:xpath, "//android.widget.LinearLayout[1]/android.widget.FrameLayout[1]/android.widget.LinearLayout[1]/android.
find_element(:xpath, "//android.widget.LinearLayout[1]/android.widget.FrameLayout[1]/android.widget.LinearLayout[1]/android.
find_element(:xpath, "//android.widget.LinearLayout[1]/android.widget.FrameLayout[1]/android.widget.LinearLayout[1]/android.
find_element(:xpath, "//android.widget.FrameLayout[1]/android.widget.FrameLayout[1]/android.widget.LinearLayout[1]/android.w
find_element(:xpath, "//android.widget.LinearLayout[1]/android.widget.FrameLayout[1]/android.widget.LinearLayout[1]/android.
find_element(:xpath, "//android.widget.LinearLayout[1]/android.widget.FrameLayout[1]/android.widget.LinearLayout[1]/android.
driver_quit
```

- Your test should look close to this. Note: Remove the appium-version and platformVersion caps. These are not needed.

UI Automator Viewer

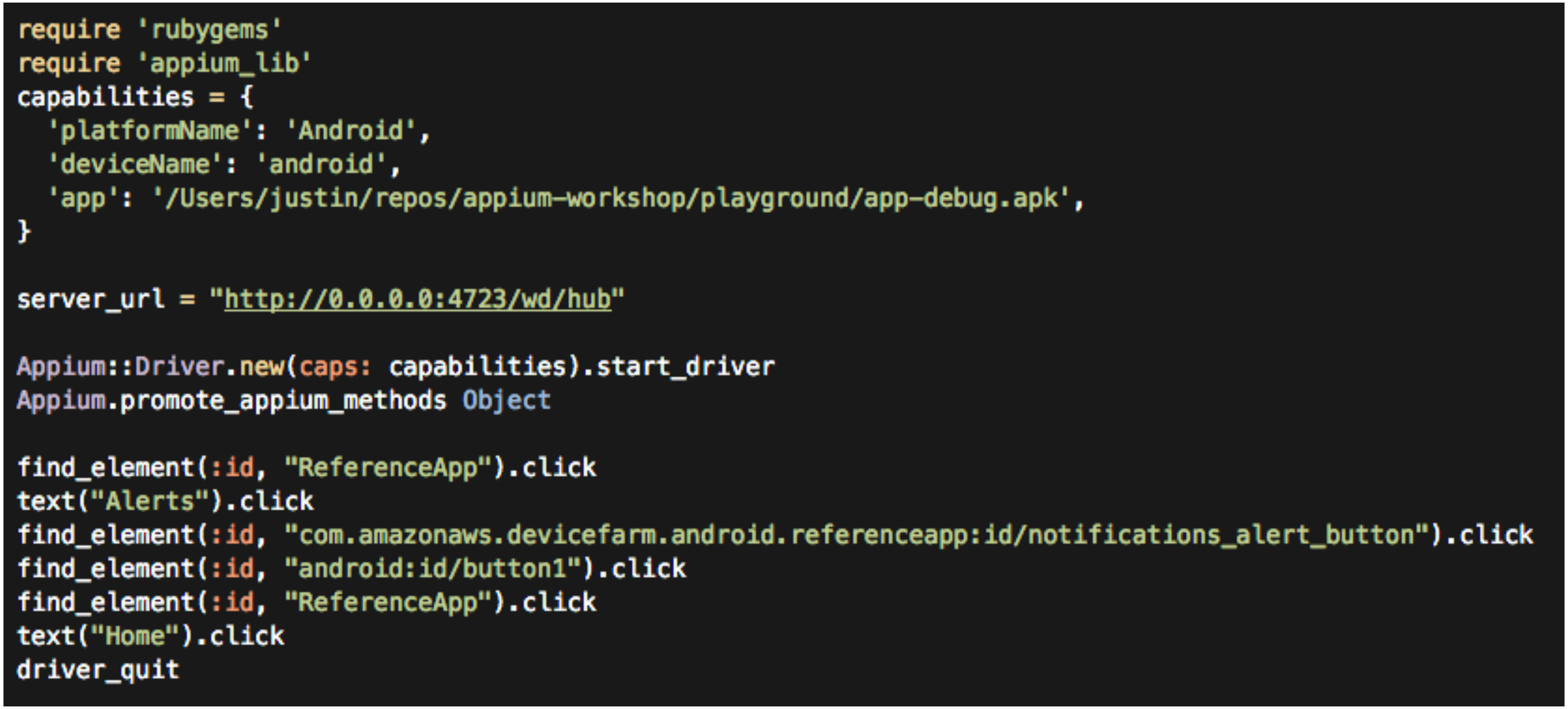
GUI tool to scan and analyze the UI components of an android application.

- Run “uiautomatorviewer” in a terminal or shell window.
- Open the test application on the emulator.
- Click the Compressed Hierarchy button. This is a bit cleaner view IMO.
- Now lets step through each action of the alert.rb test and get better locators.

Bad Xpath, Bad!

- Xpath is ok to get you started but try to at all costs avoid using it as a locator strategy. These are index based, if anything in your UI hierarchy changes the index is then shifted.
    - Xpath leads to flaky and unreliable tests.
- Use resource-id instead, which is :id in Selenium. These locators are less likely to change in the future!
- What do you do if you don't have a resource-id?
    - Ask the developer to add one for the object you wish to interact with.
    - You can use content-desc as an :id. This used to be the :name locator but is now deprecated.
        - Generally, it's also not a good idea to use content-desc if you have a multi-locale application. Values for each locale will be different and the text can change often.

- Lets change all our :xpath locators to :id’s. Using the uiautomatorviewer select each element and grab it’s resource-id, or content-desc if it does not have one.
    - Tap the Hamburger button. <- This doesn’t have a resource-id but does have a content-desc. Replace the xpath with **(:id, “ReferenceApp”)**.
    - Tap the Alerts menu option. <- So here is where I will contradict myself. I say to use a resource-id when one is available. However, in this case the resource-id’s are the same for for each menu option. Take a look for yourself.
        - We have options, though. Since the only uniqueness of each element is the title text, we can use that. Although, it’s not ideal.
        - appium_lib gem has a very handy method to search elements for text. Remove the entire xpath row and change it to “**text(“Alerts”).click**”. We will go into depth on this later.
    - Tap the ALERT button. <- Replace the xpath with **(:id, “com.amazonaws.devicefarm.android.referenceapp:id/notifications_alert_button”)**
    - Tap the OK button. <- Replace the xpath with **(:id, “android:id/button1”)**
    - Tap the Hamburger button. <- Replace the xpath with **(:id, “ReferenceApp”)**
    - Tap the Home button. <- Replace the xpath with **text(“Home”).click**.
- Save the test script.



```ruby
require 'rubygems'
require 'appium_lib'
capabilities = {
  'platformName': 'Android',
  'deviceName': 'android',
  'app': '/Users/justin/repos/appium-workshop/playground/app-debug.apk',
}

server_url = "http://0.0.0.0:4723/wd/hub"

Appium::Driver.new(caps: capabilities).start_driver
Appium.promote_appium_methods Object

find_element(:id, "ReferenceApp").click
text("Alerts").click
find_element(:id, "com.amazonaws.devicefarm.android.referenceapp:id/notifications_alert_button").click
find_element(:id, "android:id/button1").click
find_element(:id, "ReferenceApp").click
text("Home").click
driver_quit
```

- alert.rb should look like this now.

Lets try the test again!

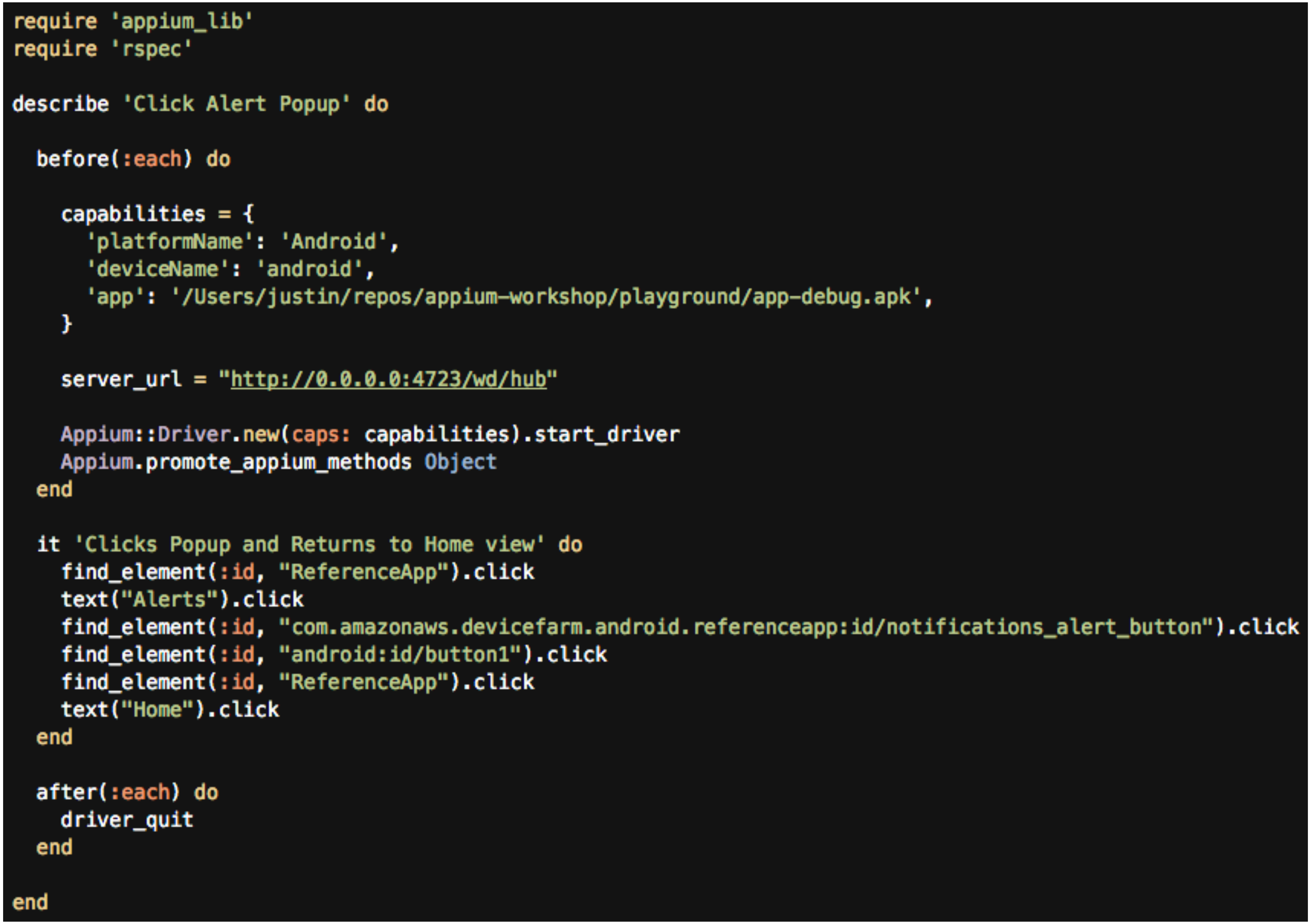
- Start the Appium server. Click the Play or Launch button.
- From the command line, run “ruby alert.rb”.
- Did the test run without errors?
- Raise your hand if it didn’t.

Assertions!

- Ok, Great! We have things automated now, but we want to know if the app does what it should.
- We need to add Rspec to do our assertions.
    - From the command line, run “gem install rspec”.
- Copy alert.rb to the spec folder in playground.
- Rename alert.rb to alert_spec.rb.
- Open alert_spec.rb in sublime.

Let add the Rpec framework!

- First we need to require rspec.
- Next we need to define our Describe / Scenario.
- We need to add our setup methods. We will place these in our before(:each) block.
- Next we need to add test steps and our assertion. This will be placed in the “it” block.
- Finally, we need to add our cleanup methods. We will place these in our after(:each) block.



```ruby
require 'appium_lib'
require 'rspec'

describe 'Click Alert Popup' do

  before(:each) do

    capabilities = {
      'platformName': 'Android',
      'deviceName': 'android',
      'app': '/Users/justin/repos/appium-workshop/playground/app-debug.apk',
    }

    server_url = "http://0.0.0.0:4723/wd/hub"

    Appium::Driver.new(caps: capabilities).start_driver
    Appium.promote_appium_methods Object
  end

  it 'Clicks Popup and Returns to Home view' do
    find_element(:id, "ReferenceApp").click
    text("Alerts").click
    find_element(:id, "com.amazonaws.devicefarm.android.referenceapp:id/notifications_alert_button").click
    find_element(:id, "android:id/button1").click
    find_element(:id, "ReferenceApp").click
    text("Home").click
  end

  after(:each) do
    driver_quit
  end

end
```

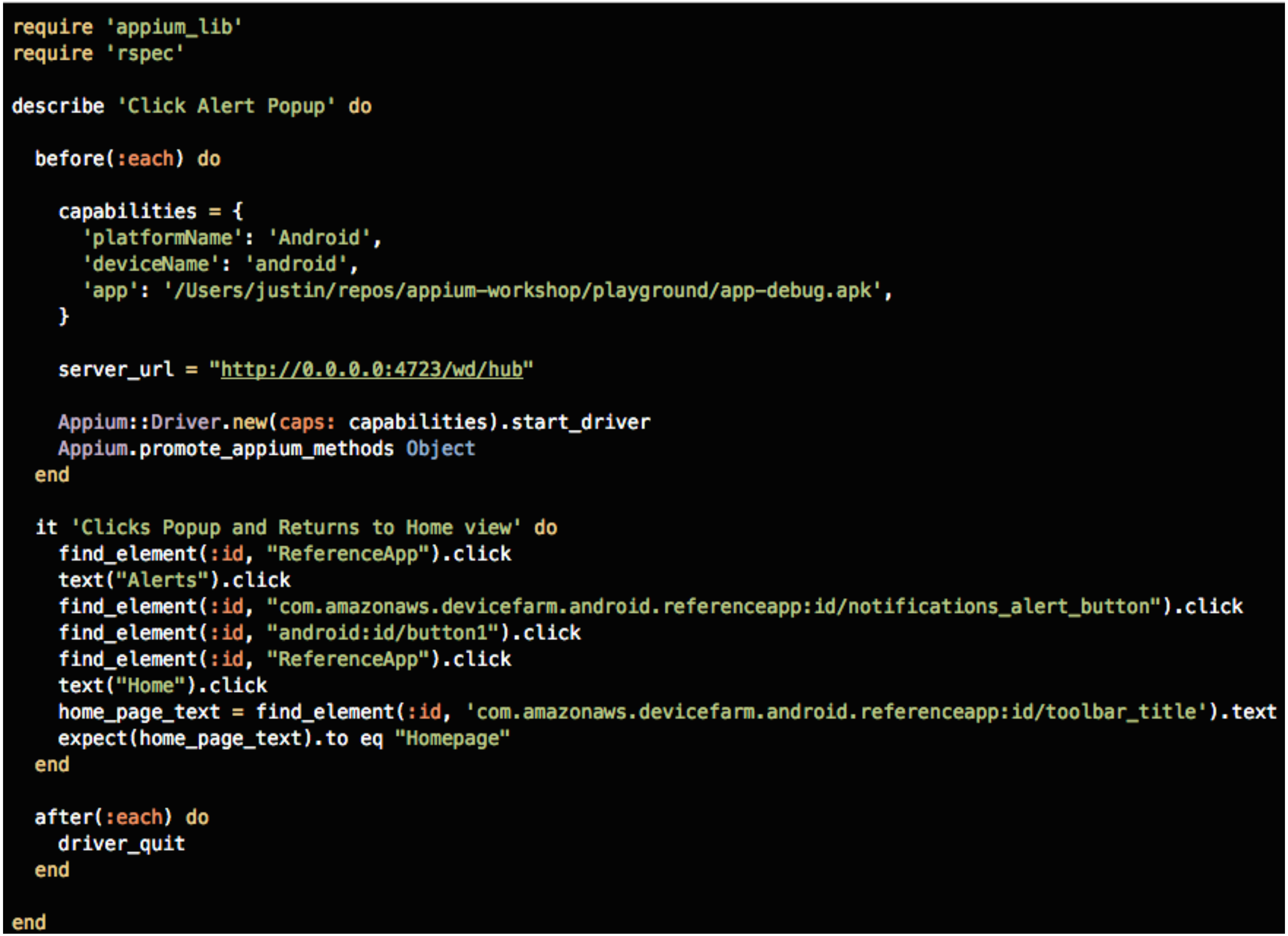
- Update alert_spec.rb to look like this.

Lets run again…

- Start the Appium server.
- Goto the spec directory in playground.
- From the command line, run “rspec alert_spec.rb”
- You should see “1 example, 0 failures” returned.
- Raise your hand if you don't see this.

That’s Great! but whats wrong with this?

- For one, we need to add an assertion. Basically, rspec ran and completed because it didn’t encounter a assertion failure or an error.
    - An error can be anything that occurs between steps which prevents another step.
- Lets add the assertion. We want to validate the app returns to the home screen and displays text “Homepage”. We will use the “.text” method to accomplish this.
- Open uiautomatorviewer again. Lets get the resource-id for the Homepage title.
    - The :id is 'com.amazonaws.devicefarm.android.referenceapp:id/toolbar_title'
- Add the following below “text(“Home”).click”
    - home_page_text = find_element(:id, 'com.amazonaws.devicefarm.android.referenceapp:id/toolbar_title').text
    - expect(home_page_text).to eq “Homepage"



```ruby
require 'appium_lib'
require 'rspec'

describe 'Click Alert Popup' do

  before(:each) do

    capabilities = {
      'platformName': 'Android',
      'deviceName': 'android',
      'app': '/Users/justin/repos/appium-workshop/playground/app-debug.apk',
    }

    server_url = "http://0.0.0.0:4723/wd/hub"

    Appium::Driver.new(caps: capabilities).start_driver
    Appium.promote_appium_methods Object
  end

  it 'Clicks Popup and Returns to Home view' do
    find_element(:id, "ReferenceApp").click
    text("Alerts").click
    find_element(:id, "com.amazonaws.devicefarm.android.referenceapp:id/notifications_alert_button").click
    find_element(:id, "android:id/button1").click
    find_element(:id, "ReferenceApp").click
    text("Home").click
    home_page_text = find_element(:id, 'com.amazonaws.devicefarm.android.referenceapp:id/toolbar_title').text
    expect(home_page_text).to eq "Homepage"
  end

  after(:each) do
    driver_quit
  end

end
```

- alert_spec.rb should look like this now.

Lets run again…

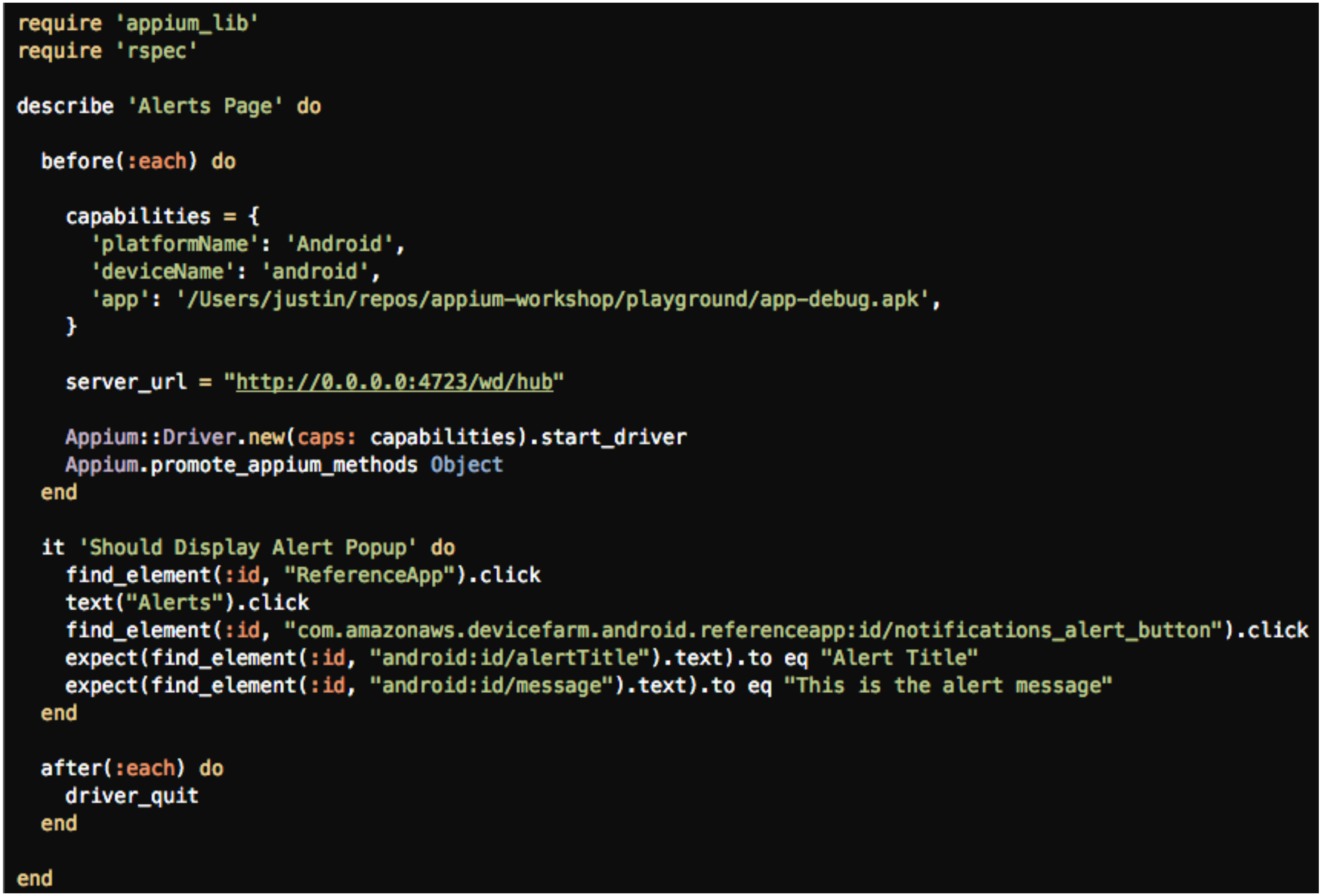
- Start the Appium server.
- Goto the spec directory in playground.
- From the command line, run “rspec alert_spec.rb”
- You should see “1 example, 0 failures” returned.
- Ok, same as before but this time it actually validated something. But… we still don't really know if it did.
- In the alert_spec.rb, change the expected text from “Homepage” to “OMG WHERE AM I”.
- From the command line, run “rspec alert_spec.rb”
- This time we should get:
    - “Failure/Error: expect(home_page_text).to eq "OMG WHERE AM I” got: "Homepage"
    - “1 example, 1 failure”.
- Raise your hand if you did not receive this.

Awesome! but what else is wrong with this test?

- We are doing actions outside of the scope of the test.
- The scenario is to “Click Alert Popup” so we really shouldn’t do any more steps after we click the popup.
- Also, we should change the describe to be more broader of the Alert and Dialogs page.
- We should split these into two tests.

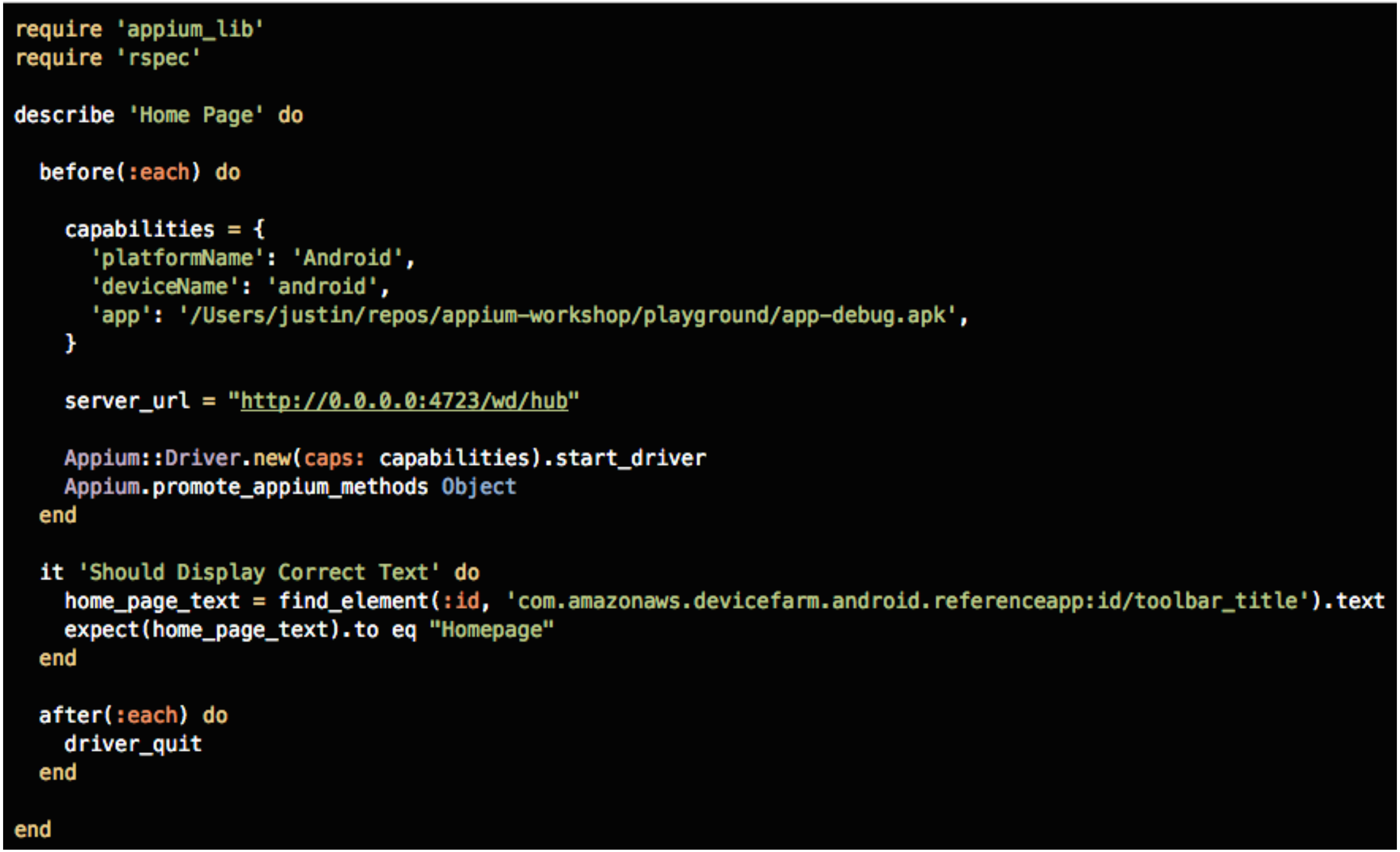
Test Refactor

- Create a new file named home_spec.rb in the spec folder.
- Copy and paste the contents from alert_spec.rb into home_spec.rb.
- Lets change the describe to: “Home Page”.
- Change the “it” block to: “Should Display Correct Text”.
- Remove all test steps up to where we get the home title text. e.g. leave the last two steps inside the “it” block.
- Change the expect text back to “Homepage”
- In the alert_spec.rb.
- Change the describe to: “Alerts Page”.
- Change the “it” block to: “Should Display Alert Popup”.
- Remove everything after: find_element(:id, "com.amazonaws.devicefarm.android.referenceapp:id/notifications_alert_button").click
- Now we need to add an assertion for the alerts_spec.
- Open uiautomatorviewer again. Lets get the resource-id’s for the Alert popup title and message.
  - The title :id is “android:id/alertTitle”
  - The message :id is “android:id/message”
- Lets add the expects for these:
  - expect(find_element(:id, “android:id/alertTitle”).text).to eq “Alert Title”
  - expect(find_element(:id, “android:id/message”).text).to eq “This is the alert message”



```ruby
require 'appium_lib'
require 'rspec'

describe 'Alerts Page' do

  before(:each) do

    capabilities = {
      'platformName': 'Android',
      'deviceName': 'android',
      'app': '/Users/justin/repos/appium-workshop/playground/app-debug.apk',
    }

    server_url = "http://0.0.0.0:4723/wd/hub"

    Appium::Driver.new(caps: capabilities).start_driver
    Appium.promote_appium_methods Object
  end

  it 'Should Display Alert Popup' do
    find_element(:id, "ReferenceApp").click
    text("Alerts").click
    find_element(:id, "com.amazonaws.devicefarm.android.referenceapp:id/notifications_alert_button").click
    expect(find_element(:id, "android:id/alertTitle").text).to eq "Alert Title"
    expect(find_element(:id, "android:id/message").text).to eq "This is the alert message"
  end

  after(:each) do
    driver_quit
  end

end
```

- alert_spec.rb should look like this now.



```ruby
require 'appium_lib'
require 'rspec'

describe 'Home Page' do

  before(:each) do

    capabilities = {
      'platformName': 'Android',
      'deviceName': 'android',
      'app': '/Users/justin/repos/appium-workshop/playground/app-debug.apk',
    }

    server_url = "http://0.0.0.0:4723/wd/hub"

    Appium::Driver.new(caps: capabilities).start_driver
    Appium.promote_appium_methods Object
  end

  it 'Should Display Correct Text' do
    home_page_text = find_element(:id, 'com.amazonaws.devicefarm.android.referenceapp:id/toolbar_title').text
    expect(home_page_text).to eq "Homepage"
  end

  after(:each) do
    driver_quit
  end

end
```

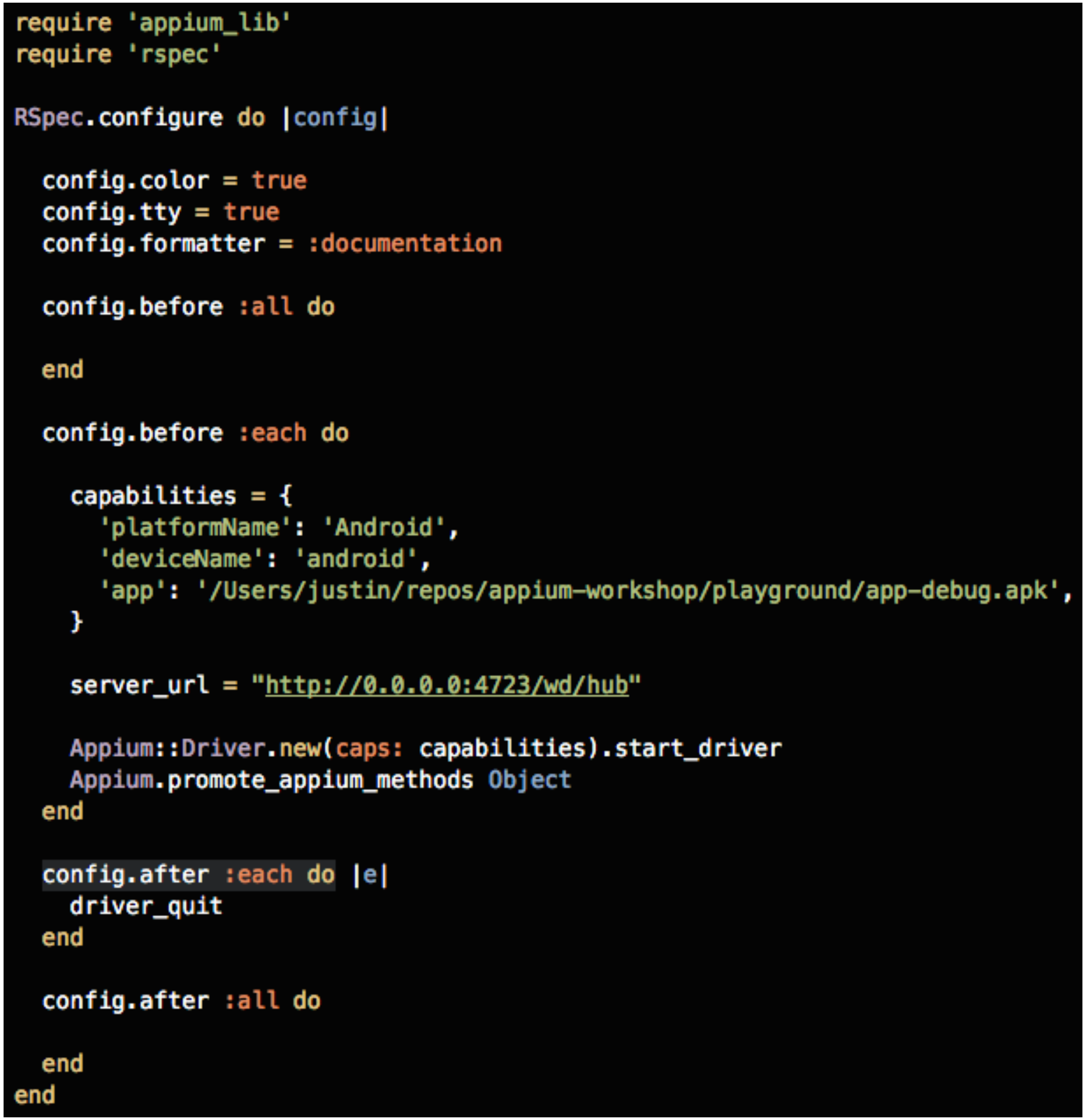
- home_spec.rb should look like this.

We have to specs now!

- Lets run them together!
- Start your Appium server. (if it’s not already)
- On the command line, go back/up one directory to the playground dir.
- Run “rspec spec”
- This should have ran both specs. You should see “2 examples, 0 failures”.
- Raise your hand if you didn’t get this.

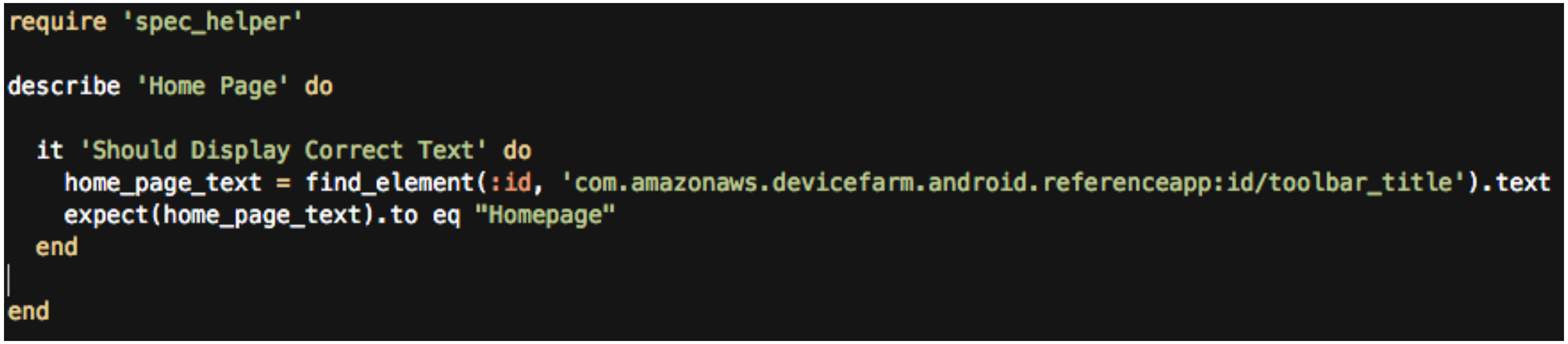
What else is wrong with this?

- We have code duplication!
- Thankfully, rspec has something for this called spec_helper.
- Lets take our before(:each) from each spec and put these in the “config.before :each do“ block of the spec_helper.rb.
- Lets put the driver.quit in the “config.after :each do” block on the spec_helper.
- Then remove these before and after blocks from each spec.
- Add require “spec_helper” to the top of each spec. This should be the only require for each one.



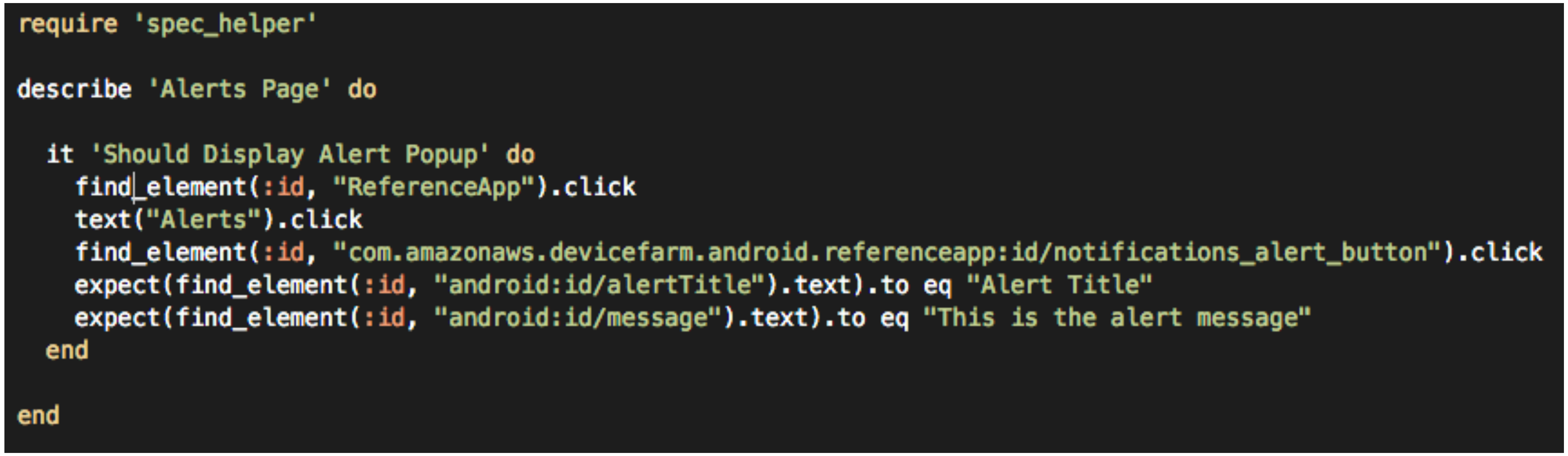
```ruby
require 'appium_lib'
require 'rspec'

RSpec.configure do |config|

  config.color = true
  config.tty = true
  config.formatter = :documentation

  config.before :all do

  end

  config.before :each do

    capabilities = {
      'platformName': 'Android',
      'deviceName': 'android',
      'app': '/Users/justin/repos/appium-workshop/playground/app-debug.apk',
    }

    server_url = "http://0.0.0.0:4723/wd/hub"

    Appium::Driver.new(caps: capabilities).start_driver
    Appium.promote_appium_methods Object
  end

  config.after :each do |e|
    driver_quit
  end

  config.after :all do

  end
end
```

- spec_helper.rb should look like this.



```ruby
require 'spec_helper'

describe 'Home Page' do

  it 'Should Display Correct Text' do
    home_page_text = find_element(:id, 'com.amazonaws.devicefarm.android.referenceapp:id/toolbar_title').text
    expect(home_page_text).to eq "Homepage"
  end

end
```

- home_spec.rb should look like this.

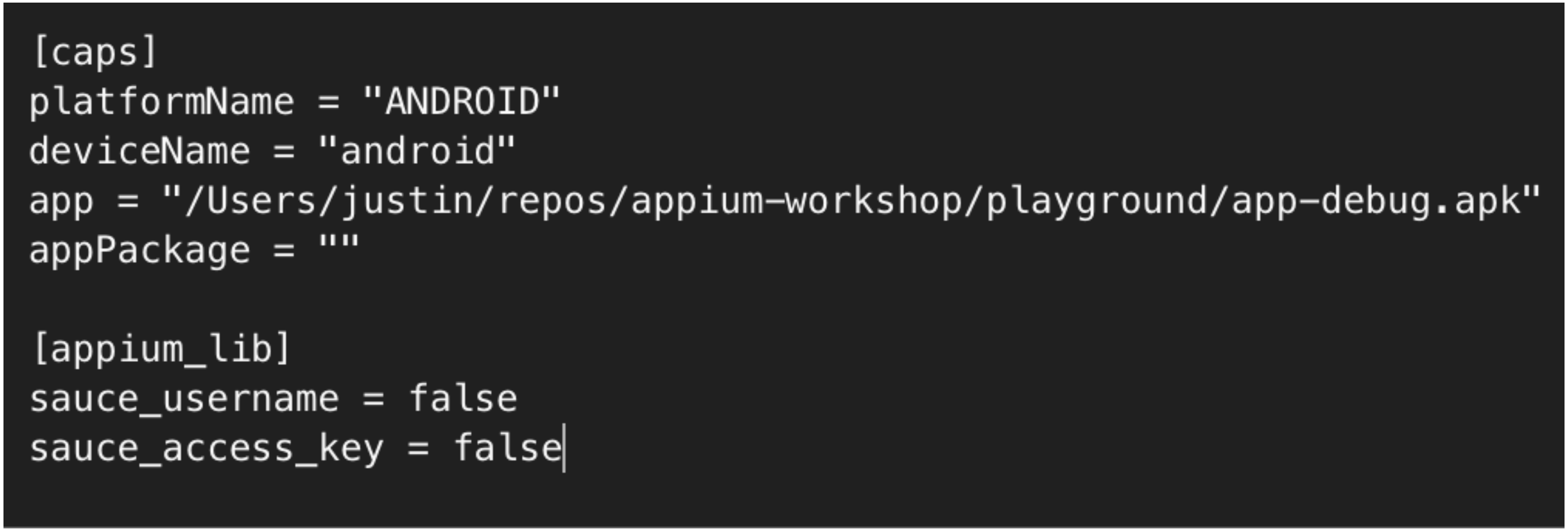


```ruby
require 'spec_helper'

describe 'Alerts Page' do

  it 'Should Display Alert Popup' do
    find_element(:id, "ReferenceApp").click
    text("Alerts").click
    find_element(:id, "com.amazonaws.devicefarm.android.referenceapp:id/notifications_alert_button").click
    expect(find_element(:id, "android:id/alertTitle").text).to eq "Alert Title"
    expect(find_element(:id, "android:id/message").text).to eq "This is the alert message"
  end

end
```

- alert_spec.rb should look like this.

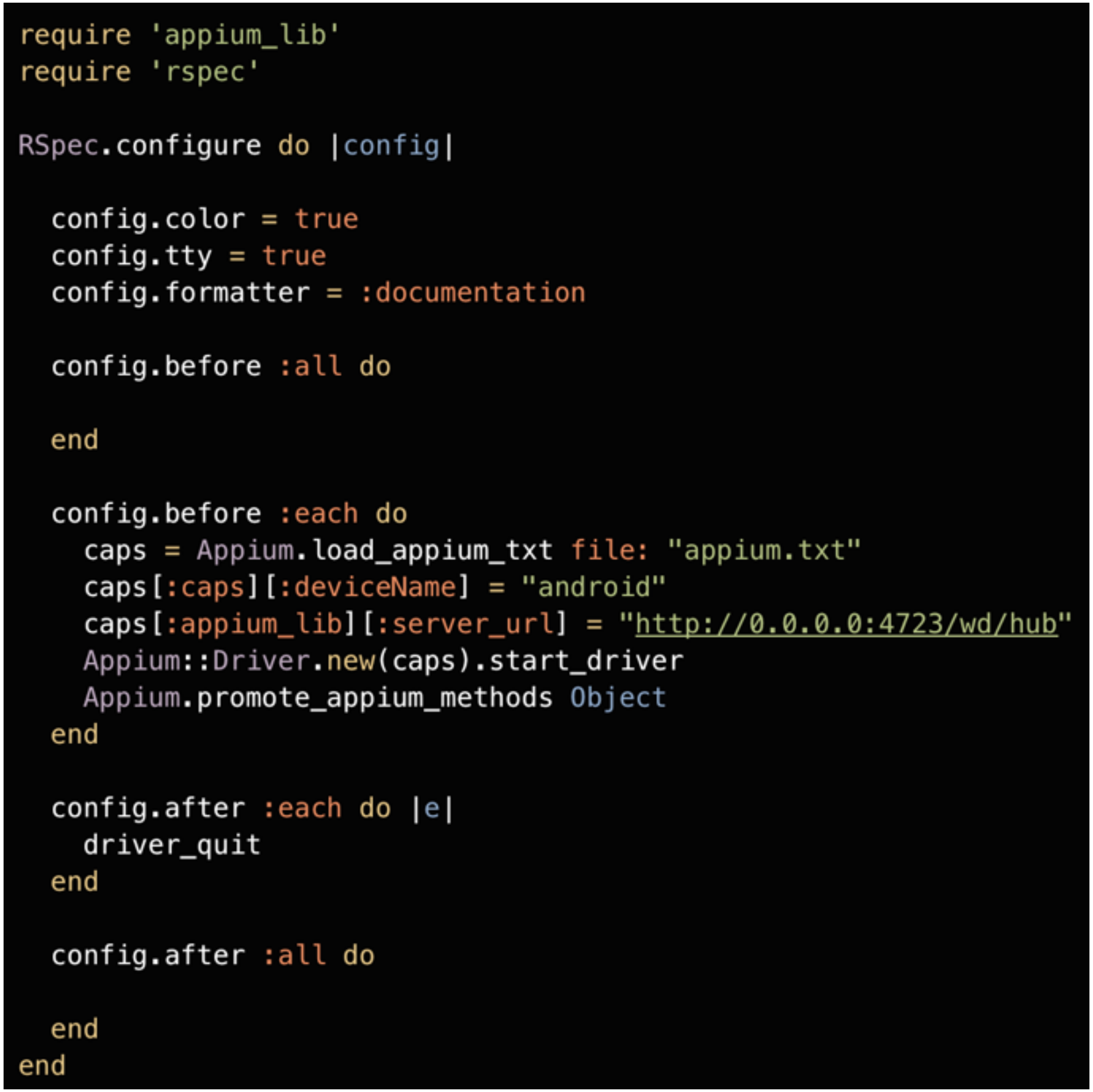
appium.txt

- There is one more thing we can do to clean things up a bit. We can create a appium.txt file to store our capabilities.
- Navigate to the spec folder in playground. From the command line, run “arc setup android”. This will create a appium.txt file.
- Open appium.txt in sublime.
- Set the capabilities for our test app like below under the [caps]
    - platformName = “Android"
    - deviceName = "android"
    - app = "/Users/justin/repos/appium-workshop/playground/app-debug.apk"
    - appPackage = “"
    - Set sauce_username and sauce_access_key = false
- In the spec_helper.rb remove the capabilities hash.
- Replace the capabilities hash with the below in the before(:each) block:
    - caps = caps = Appium.load_appium_txt file: "appium.txt"
    - caps[:caps][:deviceName] = "android"
    - caps[:appium_lib][:server_url] = "http://0.0.0.0:4723/wd/hub"



```
[caps]
platformName = "ANDROID"
deviceName = "android"
app = "/Users/justin/repos/appium-workshop/playground/app-debug.apk"
appPackage = ""

[appium_lib]
sauce_username = false
sauce_access_key = false
```

- appium.txt should look like this.



```ruby
require 'appium_lib'
require 'rspec'

RSpec.configure do |config|

  config.color = true
  config.tty = true
  config.formatter = :documentation

  config.before :all do

  end

  config.before :each do
    caps = Appium.load_appium_txt file: "appium.txt"
    caps[:caps][:deviceName] = "android"
    caps[:appium_lib][:server_url] = "http://0.0.0.0:4723/wd/hub"
    Appium::Driver.new(caps).start_driver
    Appium.promote_appium_methods Object
  end

  config.after :each do |e|
    driver_quit
  end

  config.after :all do

  end
end
```

- spec_helper.rb should now look like this.

Lets Run The Specs

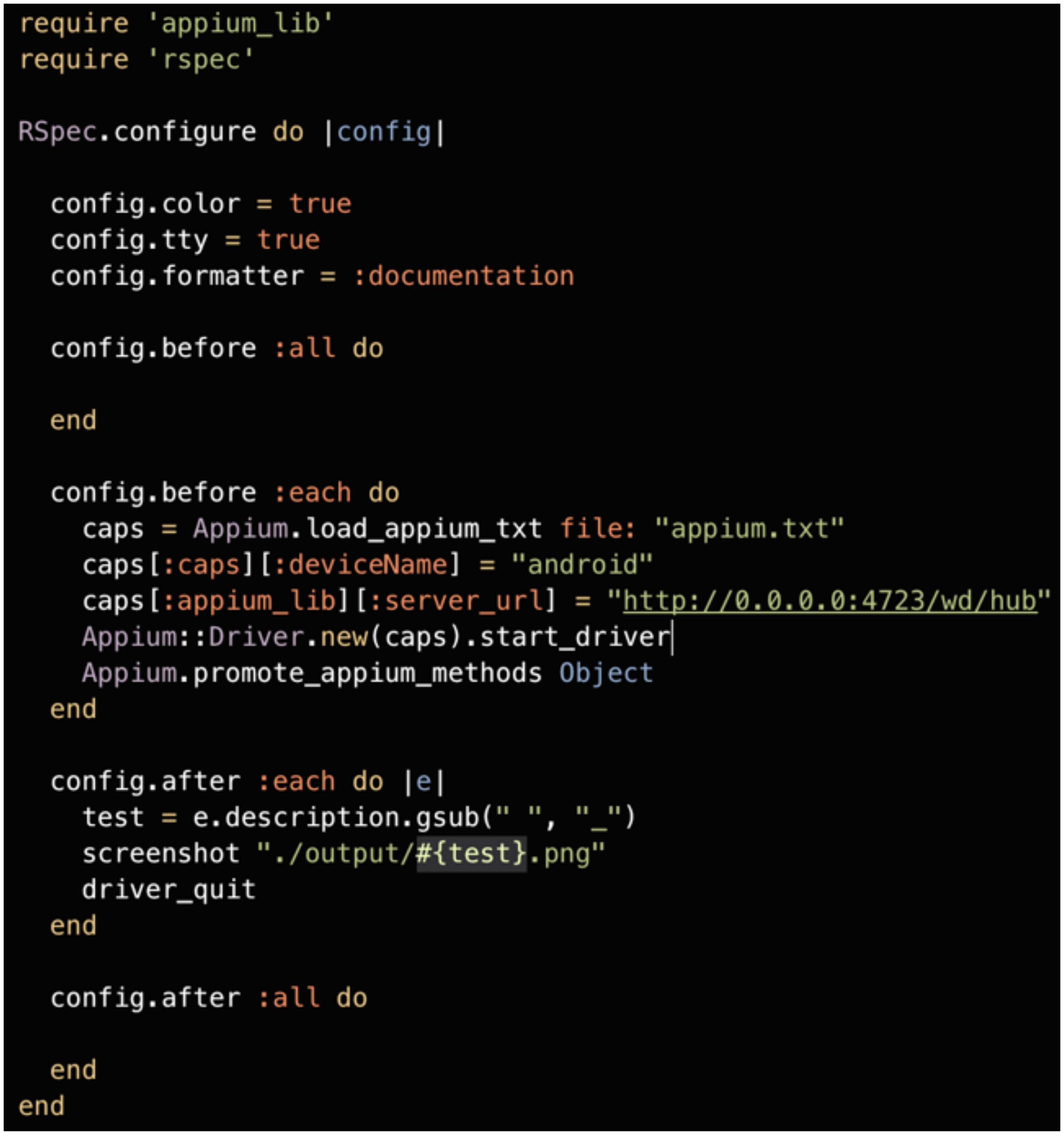
- From the playground directory, run “rspec spec”.
- You should now see a much nicer output now.
  - This is set from the config.color and config.formatter in spec_helper.rb.
- Raise your hand if you didn’t get this.

Test Data!

- Lets capture some test data for our tests.
    - Screenshots
    - Logs

Screenshots

- Lets add the selenium command to capture a screenshot at the end of each test.
- The best place to put this is in the spec_helper.rb and inside the after(:each) block.
- First we need to figure the name of the screenshot. The test name makes the most sense!
    - Add the following to the after(:each) block before the driver.quit.
    - test = e.description.gsub(" ", “_”) <- The .gsub is to remove spaces and replace with a underscore.
    - Now we need to take the screenshot. Add this after the test variable.
    - screenshot “./output/#{test}.png"
- This will take a screenshot after every test completes regardless of pass or fail.
- If you only want a screenshot on failure you can use a “unless”.
    - e.g. screenshot “./output/#{test}.png” unless e.exception.nil?



```ruby
require 'appium_lib'
require 'rspec'

RSpec.configure do |config|

  config.color = true
  config.tty = true
  config.formatter = :documentation

  config.before :all do

  end

  config.before :each do
    caps = Appium.load_appium_txt file: "appium.txt"
    caps[:caps][:deviceName] = "android"
    caps[:appium_lib][:server_url] = "http://0.0.0.0:4723/wd/hub"
    Appium::Driver.new(caps).start_driver
    Appium.promote_appium_methods Object
  end

  config.after :each do |e|
    test = e.description.gsub(" ", "_")
    screenshot "./output/#{test}.png"
    driver_quit
  end

  config.after :all do

  end
end
```

- spec_helper.rb should look like this now.

Lets Run The Specs

- From the playground directory, run “rspec spec”.
- After the tests are finished you should see two screenshots named for each test in the output folder.
- Raise you hand if you don't see this.

Logs!

- On the command line, run “adb logcat -v long”.
    - That will print out the log output of your android devices. the “-v long” makes it a bit easier to read.
- This is good information to capture, especially if errors occur in your app (or device) which isn’t displayed on the UI.
- Now we need to create a method to start capturing the log output and then stopping it.
- Create a method like below. Put this in the spec_helper.rb at the bottom for the end for the rspec block. *It’s not recommended to put methods inside the spec_helper file but to create another file and then require it. We will touch on this later.

```
def start_logcat test

   pid = spawn("adb logcat -v long", :out=>"./output/logcat-#{test}.log")

   ENV[“LOGCAT”] = pid.to_s

end
```

- What the start_logcat method does is it starts a background process (“spawn”) capturing the log output and placing it in the output directory.
- We need to now pipe the example/test info into the before(:each) block. Replace “config.before :each do” with “config.before :each do |e|”.
- Create a variable to capture the test name similar to what we did for the screenshot after(:each). It should be placed anywhere after the Appium::Driver.new(caps).start_driver.
    - test = e.description.gsub(" ", “_”)
- Put the start_logcat method after the test variable.

Logs Continued…

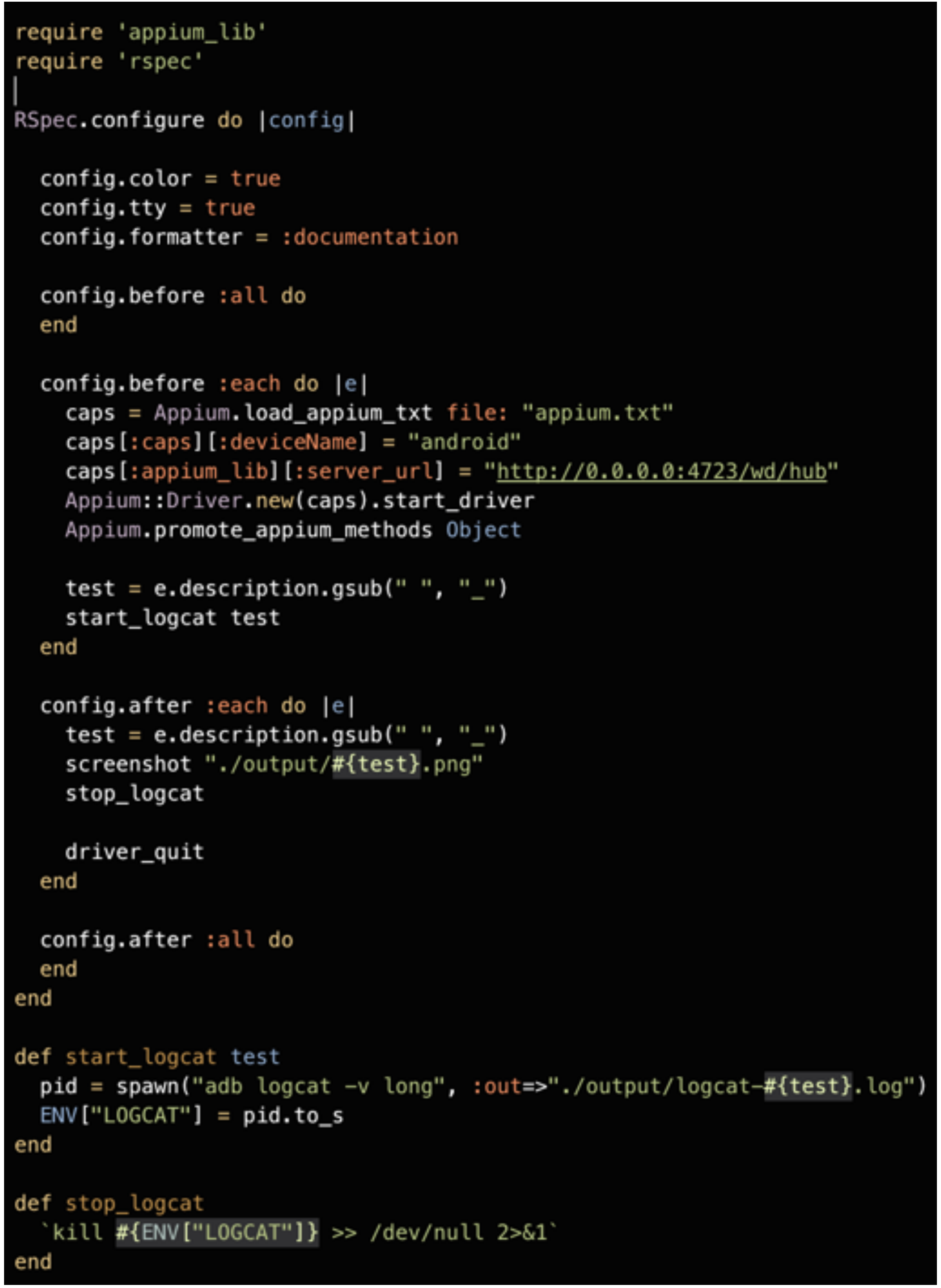
- Now we need a method to stop capturing the output. The ENV[“LOGCAT”] environment variable is the running process id. We can use this to kill it.
  - For mac:

```
def stop_logcat pid

   `kill #{ENV[“LOGCAT”]} >> /dev/null 2>&1`

end
```

  - For Windows:

```
def stop_logcat pid

   system("taskkill /f /pid #{ENV[“LOGCAT”]} > NUL”)

end
```

- Put this method in the after(:each) block in spec_helper.rb. It should be placed anywhere before the driver.quit.



```ruby
require 'appium_lib'
require 'rspec'

RSpec.configure do |config|

  config.color = true
  config.tty = true
  config.formatter = :documentation

  config.before :all do
  end

  config.before :each do |e|
    caps = Appium.load_appium_txt file: "appium.txt"
    caps[:caps][:deviceName] = "android"
    caps[:appium_lib][:server_url] = "http://0.0.0.0:4723/wd/hub"
    Appium::Driver.new(caps).start_driver
    Appium.promote_appium_methods Object

    test = e.description.gsub(" ", "_")
    start_logcat test
  end

  config.after :each do |e|
    test = e.description.gsub(" ", "_")
    screenshot "./output/#{test}.png"
    stop_logcat

    driver_quit
  end

  config.after :all do
  end
end

def start_logcat test
  pid = spawn("adb logcat -v long", :out=>"./output/logcat-#{test}.log")
  ENV["LOGCAT"] = pid.to_s
end

def stop_logcat
  `kill #{ENV["LOGCAT"]} >> /dev/null 2>&1`
end
```

- spec_helper.rb should look like this now.

Lets Run The Specs

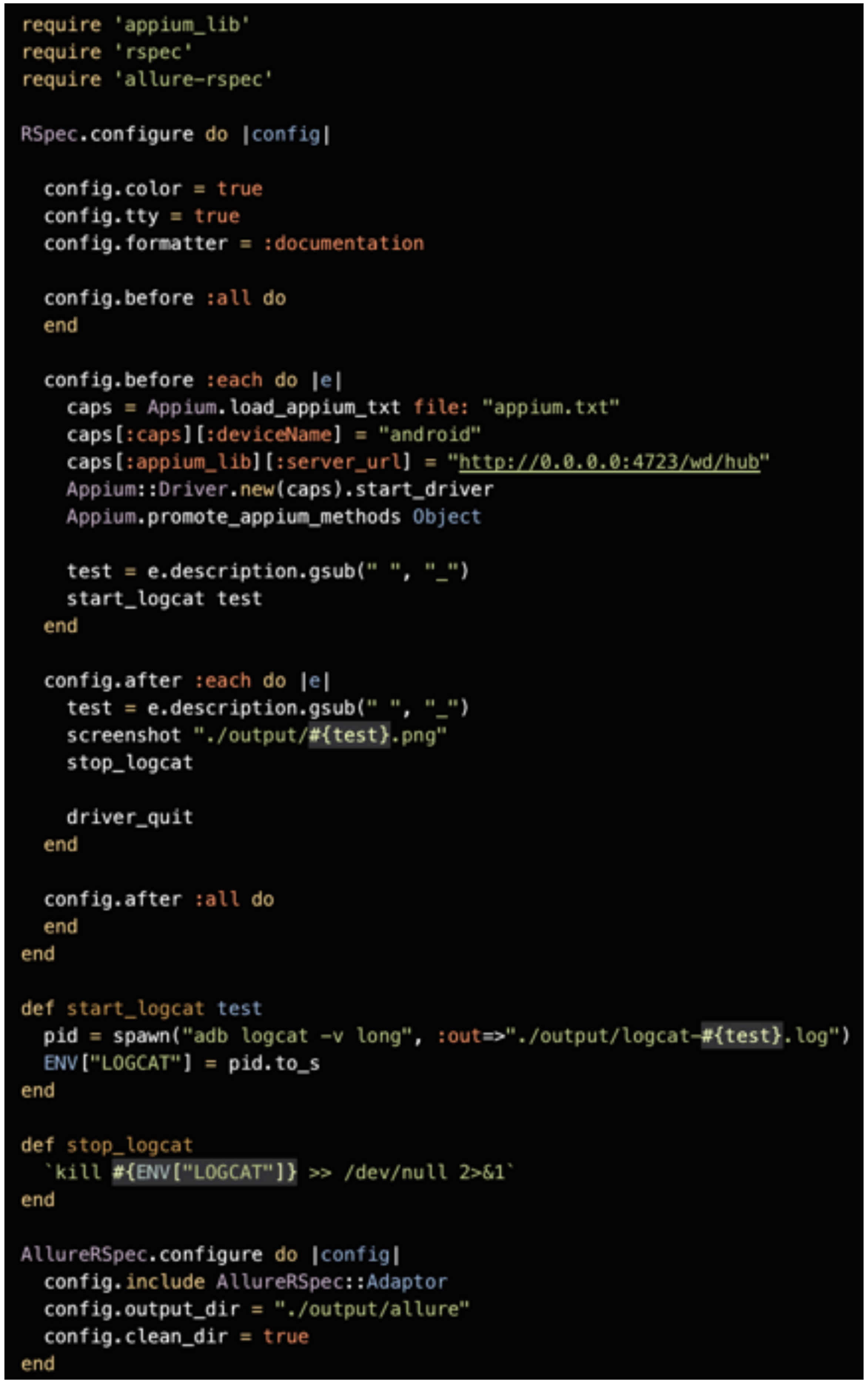
- From the playground directory, run “rspec spec”.
- After the tests are finished you should see two logs named for each test in the output folder.
- Raise you hand if you don't see this.

Reporting!

- Test are worthless without good reporting! That is why we are going to use Allure.
- Allure is a cross-platform reporting framework that lets you attach files (metadata).
- First we are going to have to install the rspec adapter for the allure.
    - On the command line, run “gem install allure-rspec”.
    - Then require it at the top of the spec_helper.rb.
        - require “allure-rspec”
- Next we need to add the config block at the bottom of our spec_helper.

```
AllureRSpec.configure do |config|

  config.include AllureRSpec::Adaptor

  config.output_dir = "./output/allure"

  config.clean_dir = true

end
```

- Place this beneath the stop_logcat method.



```ruby
require 'appium_lib'
require 'rspec'
require 'allure-rspec'

RSpec.configure do |config|

  config.color = true
  config.tty = true
  config.formatter = :documentation

  config.before :all do
  end

  config.before :each do |e|
    caps = Appium.load_appium_txt file: "appium.txt"
    caps[:caps][:deviceName] = "android"
    caps[:appium_lib][:server_url] = "http://0.0.0.0:4723/wd/hub"
    Appium::Driver.new(caps).start_driver
    Appium.promote_appium_methods Object

    test = e.description.gsub(" ", "_")
    start_logcat test
  end

  config.after :each do |e|
    test = e.description.gsub(" ", "_")
    screenshot "./output/#{test}.png"
    stop_logcat

    driver_quit
  end

  config.after :all do
  end
end

def start_logcat test
  pid = spawn("adb logcat -v long", :out=>"./output/logcat-#{test}.log")
  ENV["LOGCAT"] = pid.to_s
end

def stop_logcat
  `kill #{ENV["LOGCAT"]} >> /dev/null 2>&1`
end

AllureRSpec.configure do |config|
  config.include AllureRSpec::Adaptor
  config.output_dir = "./output/allure"
  config.clean_dir = true
end
```

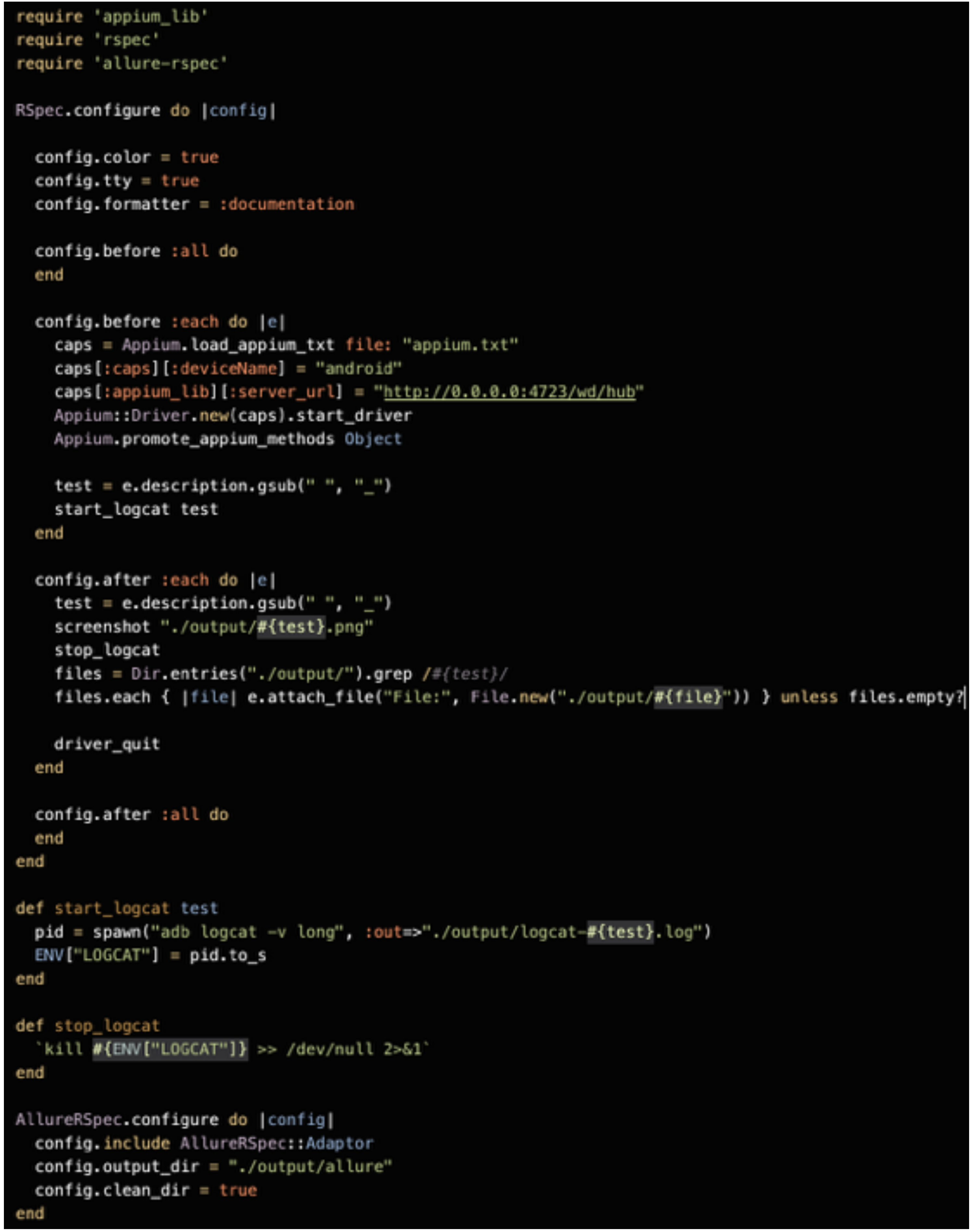
- spec_helper.rb should look like this now.

Lets Run The Specs

- From the playground directory, run “rspec spec”.
- After the tests are finished you should see a new folder in the output directory named allure with some xml files in it.
- Raise you hand if you don't see this.
- Lets generate the report!
    - From the command line, run “allure report generate output/allure”
    - Then run “allure report open”. This should then open a browser window with the report.
- Raise you hand if you don't see this.
- Go ahead and click the menu buttons on the left side and see the report details.
- On the XUnit screen you can see your tests here. By default it shows only the failed tests. You can , however, change this by clicking the Passed tab.
- Something is missing, though. The test data we generated!

Attaching Files to Allure

- Now we need to attach the files to the report.
- First we need to get the files to attach. Thankfully we can use the test name variable we've already created.
    - files = Dir.entries("./output/").grep /#{test}/
    - This creates an array of the files that have the test name.
    - files.each { |file| e.attach_file("File:", File.new("./output/#{file}")) } unless files.empty?
    - This then loops through each file in the array and attaches it to the report. If the array was empty it would not attach anything.
- Place these underneath the stop_logcat method in the after(:each) block in spec_helper.
- As you can see the spec_helper.rb is becoming large and messy. We'll fix this part later.



```ruby
require 'appium_lib'
require 'rspec'
require 'allure-rspec'

RSpec.configure do |config|

  config.color = true
  config.tty = true
  config.formatter = :documentation

  config.before :all do
  end

  config.before :each do |e|
    caps = Appium.load_appium_txt file: "appium.txt"
    caps[:caps][:deviceName] = "android"
    caps[:appium_lib][:server_url] = "http://0.0.0.0:4723/wd/hub"
    Appium::Driver.new(caps).start_driver
    Appium.promote_appium_methods Object

    test = e.description.gsub(" ", "_")
    start_logcat test
  end

  config.after :each do |e|
    test = e.description.gsub(" ", "_")
    screenshot "./output/#{test}.png"
    stop_logcat
    files = Dir.entries("./output/").grep /#{test}/
    files.each { |file| e.attach_file("File:", File.new("./output/#{file}")) } unless files.empty?

    driver_quit
  end

  config.after :all do
  end
end

def start_logcat test
  pid = spawn("adb logcat -v long", :out=>"./output/logcat-#{test}.log")
  ENV["LOGCAT"] = pid.to_s
end

def stop_logcat
  `kill #{ENV["LOGCAT"]} >> /dev/null 2>&1`
end

AllureRSpec.configure do |config|
  config.include AllureRSpec::Adaptor
  config.output_dir = "./output/allure"
  config.clean_dir = true
end
```

- spec_helper.rb should look like this now.

Lets Run The Specs

- From the playground directory, run “rspec spec”.
- Lets generate the report!
  - From the command line, run “allure report generate output/allure”
  - Then run “allure report open”. This should then open a browser window with the report.
- Click the XUnit button (left side) and then the Passed tab (right side).
- Then click any of the tests to view the results. Notice the screenshot and log files that are now attached to the report.
- Raise you hand if you don't see this.

Interactive Ruby (IRB)

- We are going to see a new way of finding elements without having to use the Appium Inspector or the uiautomatorviewer.

- From the command line in playground directory, run “arc”.

  - This should put you into the appium console and start the app on the emulator/device.

- Raise your hand if you don't see this.

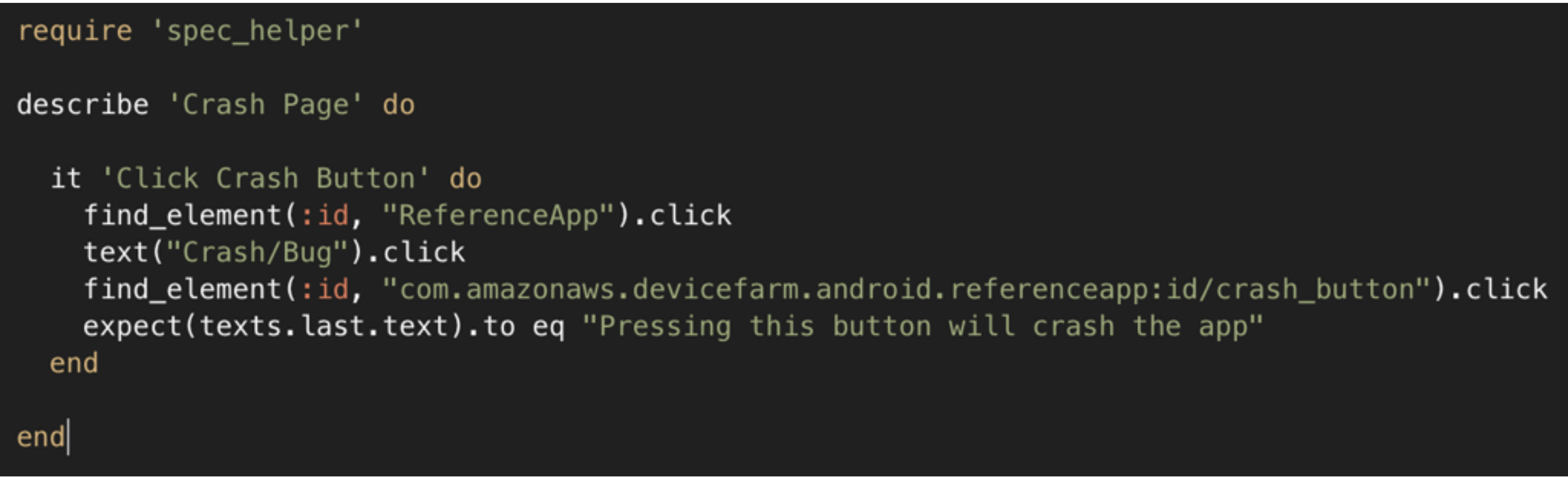
Finding Elements

- Thankfully, appium_lib has a lot of really nice helper methods baked into it.
- To name a few:
  - get_source <- returns you an xml of the page. It's impossible to read, though.
  - page <- parses the get_source xml and returns a nice printout of the elements.
  - buttons <- returns an array of all button elements with a "android.widget.ImageButton" class.
  - texts <- returns an array of all text elements with a "android.widget.TextView" class.
  - textfields <- returns an array of text field elements with a "android.widget.EditText" class.
  - back <- This will tap the back button. Be careful when using this method. It can close your app depending where your tests are.
  - find_element <- At this point you've used this. It's a method to get a single element.
  - find_elements <- This is like find_element except it will gather all the elements with the same locators into an array.

- There are many more helper methods that can be found here: **place link…..**
- Go ahead and play around with them and interact with the app in realtime.
  - Get the name of a element by doing “texts.first.text” on a page. Or buttons.first.text. If neither of those return a value you can loop through each element.
  - texts.each { |e| puts e.text }
  - buttons.each { |e| puts e.text }
  - From the page output. Get the :id of an element, then do a find_element(:id, “the locator id”).click or .text. You can also do a .methods to see what you can retrieve from the selenium object.
  - If you are getting a method by the :id locator. You can also use a method named “id”. e.g. id(“the locator id”).click instead of find_element(:id, “the locator id”).click

Create Another Test

- Create a new file in the spec folder and name it “crash_spec.rb”.
- Copy the contents of alert_spec.rb file and put it into the crash_spec.rb. Save the file.
- Rename the describe to “Crash Page”
- Rename the “it” block to “
- Using arc (appium_console) lets get the elements.
- Use the “page” method to see the elements on each page.
    - You need the hamburger button :id. You already have this so you can reuse it from the alert_spec.rb.
    - Click the “Crash/Bug” button.
    - Click the CRASH button.
    - Now set an assertion. e.g. expect(texts.last.text).to eq "Pressing this button will crash the app”.
- The app of course is going to crash but this good! We’ll then have the logcat output we can review to see the crash. This can be sent to the developer or added to a bug report!
- Exit out of appium_console by typing “driver.quit” and then “exit”.



```ruby
require 'spec_helper'

describe 'Crash Page' do

  it 'Click Crash Button' do
    find_element(:id, "ReferenceApp").click
    text("Crash/Bug").click
    find_element(:id, "com.amazonaws.devicefarm.android.referenceapp:id/crash_button").click
    expect(texts.last.text).to eq "Pressing this button will crash the app"
  end

end
```

- crash_spec.rb should look close to this.

Lets Run the “Spec”

- From the playground directory, run “rspec spec/crash_spec.rb”.
- Only the crash_spec test should have run and thus having the app crash.
- Lets generate the report!
    - From the command line, run “allure report generate output/allure”
    - Then run “allure report open”. This should then open a browser window with the report.
- Click the XUnit button (left side) and then the Passed tab (right side).
- Then click the test to view the results. Notice the screenshot and log files that are attached.
- Raise your hand if you don't see this.

Gestures

- A lot of apps use gestures to enable or create actions.
- Navigate the app to the Input Controls page. Then scroll to the right until your reach the Gesture tab.
    - Extra credit if you can do this using Appium!
- There are a ton of different gestures we can do but we'll only go through a few.

Common Gestures

- Lets get the element :id for the Swipe Here box.
  - type “page”
  - Looks like we can use :id “Gesture Action Pad”.
- A Tap: (A basic tap/click)
  - id(“Gesture Action Pad”).click
  - Notice the output on the screen tells you what actions were just made. You should see Down, Show Press, Single Tap Up, Single tap confirm.
  - This is the same as:

```
button = id("Gesture Action Pad").location.to_h

button.merge!(fingers: 1)

Appium::TouchAction.new.long_press(button).release.perform
```

- Long Press: (Tap and hold an element for 2 seconds)

```
button = id("Gesture Action Pad").location.to_h

button!(fingers: 1, duration: 2000)

Appium::TouchAction.new.long_press(button).release.perform
```

- Swipe Up:

  start = id("Gesture Action Pad").location.to_h

  finish = text(“Gestures").location.to_h

  Appium::TouchAction.new.press(start).wait(200).move_to(finish).release.perform

- Swipe Left:

  start = text("Submit Button").location.to_h

  finish = text("Gestures").location.to_h

  Appium::TouchAction.new.press(start).wait(200).move_to(finish).release.perform

- Swipe Down: (Pull to Refresh)

  - Navigate to the Pull to Refresh tab.

  start = text("Pull To Refresh").location.to_h

  start[:y] = start[:y] + 100

  finish = id("com.amazonaws.devicefarm.android.referenceapp:id/input_refresh_display").location.to_h

  Appium::TouchAction.new.press(start).wait(2000).move_to(finish).release.perform

- Scroll To:

  - Tap the Hamburger/menu button.

  - Make sure the menu is at the top. e.g. You see the Home menu option.

  - scroll_to “Fixtures” <- Notice the app will then scroll down until it finds the word Fixture.

Common Controls

- Lets play around with some common controls you will most likely have to use.
- Text Input:
  - Move app to the Input Controls > Text Field view.
  - type “page” to get the page elements.
  - Look for a class name of “android.widget.EditText”.
    - Get the :id for this field. e.g. “com.amazonaws.devicefarm.android.referenceapp:id/input_edit_text”
  - Now lets type in the text field.
    - find_element(:id, “com.amazonaws.devicefarm.android.referenceapp:id/input_edit_text”).send_keys “HELLO\n” <- The “\n” will create a new line. e.g. press enter/submit.
  - Go ahead and play around with this control.
  - Raise your hand if you have any questions or issues.

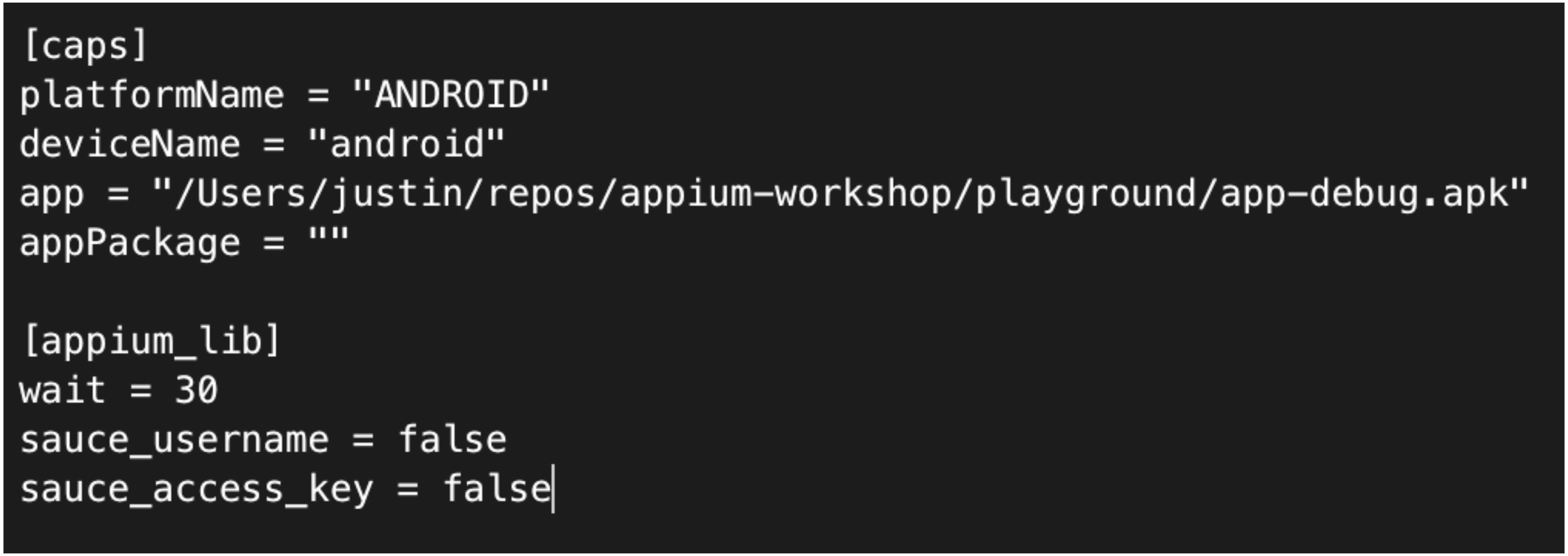
- Checkboxes:
    - type “page” to get the page elements.
    - Look for the class name of “android.widget.CheckBox”.
    - Get the :id for the checkbox. e.g. “com.amazonaws.devicefarm.android.referenceapp:id/input_checkbox”.
    - Now lets check it!
        - find_element(:id, “com.amazonaws.devicefarm.android.referenceapp:id/input_checkbox”).click
    - Lets verify it’s checked!
        - find_element(:id, “com.amazonaws.devicefarm.android.referenceapp:id/input_checkbox”).attribute(“checked”) <- this should return true.
- Radio Buttons:
    - type “page” to get the page elements.
    - Look for the class name of “android.widget.RadioButton”.
    - Get the :id for one of the Web Radio button. e.g. “com.amazonaws.devicefarm.android.referenceapp:id/radio_button_2”
    - Now lets select it!
    - find_element(:id, “com.amazonaws.devicefarm.android.referenceapp:id/radio_button_2”).click
- Lets verify it’s selected!
    - find_element(:id, “com.amazonaws.devicefarm.android.referenceapp:id/radio_button_2”).attribute(“checked”) <- this should return true

Implicit vs Explicit Waits

- Implicit waits are ideal for mobile apps. A lot of apps now rely on web services to return a payload(json/xml) before a page can load. Web does this too but connection speeds and system performance aren’t the same.
- Use explicit waits when the action you’re testing is part of the assertion or if you want to set a hard timeout.
- appium_lib has a nice helper method to wrap a implicit wait on locator finders.
- Lets test this out!
- In appium_console/arc terminal: Type “id(“DOES NOT EXIST”)”. Notice it returns an error right away saying it cannot find the element. This would be pretty annoying if it did exist but just hadn’t loaded yet.
- The “wait” method.
    - wait(5) { id(“DOES NOT EXIST”) } <- Notice this now waits 5 seconds until it stops looking.
- Lets try this with an element on another page.
    - Start on the Homepage of the app.
    - Type wait(30) { id(“com.amazonaws.devicefarm.android.referenceapp:id/notifications_alert_button”) } <- this is the alert popup button locator on the Alerts and Dialog page.
    - The Appium session is now waiting until it can find that element. Navigate the app the Alerts and Dialog page. Notice Appium returns the element once you navigate on to the page.
- Raise your hand if you don't see this.

Set a Global Implicit Wait

- Open appium.txt file in sublime.
- Under the [appium_lib] section. Set the below:
    - wait = 30
- Now when Appium starts it gets this value and sets a global implicit wait on finding locators. You don't have to use the wait(sec) method unless you are overriding the set value.



```
[caps]
platformName = "ANDROID"
deviceName = "android"
app = "/Users/justin/repos/appium-workshop/playground/app-debug.apk"
appPackage = ""

[appium_lib]
wait = 30
sauce_username = false
sauce_access_key = false
```

- appium.txt should now look like this.

Explicit Wait

- So there are times when you need to use them.
- You can easily do this by overriding the global implicit wait.
- There are a couple methods we can use:
  - wait(sec) <- You used this already for implicit but you can override the global wait time by passing a lower or higher time.
  - wait_true <- You can use this to wait until something is true.
    - wait_true(60) { id(“id of my element”).displayed? }

Clean Up Time!

- Ok, so now our test suite is starting to grow a bit. We should start cleaning things up.
- Move your specs (alert and crash) from the spec folder in the playground directory to the spec folder in the android directory. I’ve created an example for the home_spec.rb already so no need to move that.
- I’ve also already setup the whole framework of our new test suite to run in a single process, parallel and distributed.
- Lets go over everything… It’s a lot!

Eliminating Duplication

- How is this done?
  - We already did it using the spec_helper.
  - Lets now create common methods and functions.
  - Lets also create page objects.

Locator Methods

- Using Sublime, open the locators.rb in the android > spec > pages directory.
- This is where we can create our own custom methods or wrap more logic around the global appium_lib and selenium ones.
- I’ve shortened the find_element to fe and find_elements to fa(find all).
- This eliminates duplication because we now have one source containing this logic. So in the future, for example, if the appium project changes the .click function to .tap we would only need to change it in one place.
- The is_displayed? Will return true or false if the element is displayed.

Page Objects

- Open the home.rb in the android > spec > pages directory.
- home.rb is where we will put the page objects for the home/landing page.
- We create a Home class which will contain our objects for this page.
- We assign each object/id a constant.
- The page_displayed? now shows an example of where we override the default implicit wait we set in the appium.txt. Some apps rely on a web service and if that service is slow to respond we're giving Appium an extra 30 seconds (60 total) to wait.
- A good practice to do is then include the page_displayed? in methods of your page, so if you're performing an action on that page, it will first check (and wait if necessary) for the page to exist before it does anything else.
- Also notice how we now get the text for the the home page title. Instead of doing find_element(:id, 'com.amazonaws.devicefarm.android.referenceapp:id/toolbar_title').text, we just do "text HOME_PAGE_TITLE_ID". So if if .text later becomes something else we just change it on the locator.rb page.

Common Objects

- So some apps (including this one) share a lot of the same elements. Especially, when you have a common menu bar that is displayed across every page.
- Instead of duplicating the page objects for every page, we can instead create a common page class, and each page then inherits that class.
- The Hamburger button! This is displayed on every page so it's a good one to add! So is the page title object.
- Open the common.rb in the android > spec > pages directory.
- I've gone ahead and created the click_hamburger and page_title_text methods for you as an example. See how it inherits the Locators class. Now from here on, every new page inherits the Common class.

Lets Add Pages

- Go ahead and create new files named alert.rb and crash.rb in the pages directory.
- Use the home.rb as an example of how the the boiler plate code should look.
  - Name the class for each page the same as the page filename.
  - Add the objects each spec uses to execute the validations. Note: you should not add objects you don’t use. This would add clutter and make things messy.
- If you’re stuck. Take a look at each example I’ve create in the pages directory. e.g. alert.rb.example and crash.rb.example.

Update the Specs

- So now we have our page objects. We need to update the specs with our next object methods.
- Take a look at the home_spec.rb for an example.
  - We initialize the Home class. Now any methods in the Home, Common and Locators are available to us, in addition to the global appium_lib ones.
- Now go and update the alert and crash specs with the new page objects you created.
- If you're stuck. Take a look at the examples I've created in the spec directory. e.g. alert_spec.rb.example and crash_spec.rb.example.

Getting Device Data

- With Sublime, open the server_launcher.rb in the android directory.
- The methods at the top of the file, the first two get_android_devices and get_android_device_data will capture the information we want.
- Why is this important information to have? Well, you need to know everything about the platform your tests are running on. A test might run on OS: 6.0 but will fail on 4.4. Also the manufacture make a difference. I've had Samsung and LG phone just act completely different from other manufactures with the same OS version.
- Notice we put the device data into an environment variable "DEVICES". We also assign the device a thread value. We will touch on this a bit later.
- You can test this yourself by running the following on the command line:
    - OS: adb shell getprop ro.build.version.release
    - Manufacturer: adb shell getprop ro.product.manufacturer
    - Model: adb shell getprop ro.product.model
    - SDK: adb shell getprop ro.build.version.sdk
- Lastly, the save_device_data method. We use this to save the data to a file which we will later attach to Allure report so we know all about the device the test ran on.
- If you want to, you can open IRB by typing irb on the command line from the appium-workshop > android directory. Then copy and paste these three methods into the IRB terminal.
    - require 'json'
    - Then run "get_android_devices". You should get a array with a printout of your device or devices information.
    - You can save this to a file by doing "save_device_data(get_android_devices)". This will save a file for each connected device in the output directory.

Start Appium Programmatically

- There comes a time when you cannot use the Appium IDE anymore. That time is now!

- But why? If you ever want to scale your tests you have to launch the server pragmatically.

- In sublime, open the server_launcher.rb file located in appium-workshop > android directory. You will see a method name appium_server_start.

  - Basically, what it’s does is launch a appium server in the background with the given commands.

Generating Node Configs

- What is a node? A node is what connects to a selenium hub (grid). The hub then distributes tests to it. So in our case, it’s a device (emulator or physical phone).
- So we need to create node configs to tell the node server (Appium) what information to pass to the selenium hub.
- We will do this programmatically using the generate_node_config method in the server_launcher.rb file.
- However, I separated this into it’s own file to demo in the playground directory.
- On the command line, navigate to the nappium-workshop > playground directory.
- Open another shell/terminal window and navigate to the same directory.
- In one shell/terminal run: java -jar selenium-server-standalone-2.53.0.jar -role hub
    - This will start a selenium server with the role as a hub.
    - Open a browser and goto: http://localhost:4444/grid/console <- you should have a appium grid hub running on your machine now.
- In the other shell/terminal run: adb devices <- displays a list of connected devices.
    - Get a UDID of a connected device. eg. emulator-5554
- Run: ruby gen_node_config.rb emulator-5554 4755 <- The first argument is the filename, the second the port number. This will then place a file names emulator-5554.json in the node_configs directory.
- Now lets start a Appium session using this config!
    - From the playground directory run: appium --nodeconfig ./node_configs/emulator-5554.json <- your appium server should start.
    - Open a browser and goto: http://localhost:4444/grid/console <- you will now see your device displayed there listed as one of the nodes.
    - You would create one of these for every device you have connected to your machine.
- Go ahead and kill the selenium hub and Appium node servers.

Everything Together!

- In sublime, open the server_launcher.rb file located in appium-workshop > android directory. You will see a method name launch_hub_and_nodes.
- This method does what it says.
  - First it starts the selenium hub in the background.
  - Captures all the connected device data
    - Saves it to a file.
  - Then in parallel, it creates the nodes and launches the Appium servers and writes the log data to the output dir.

Helpers!

- In sublime, open the helpers.rb file located in appium-workshop > android > spec directory.
- So as part of our code cleanup we want to split out our code/methods that are helpers functions.
  - In our case, these will be the video and logcat.
  - Note: we didn’t cover video earlier because we are using emulators. However, this will work for those of you using a real device.
- Here we are creating a module named Helpers. This will be promoted after we start the appium driver and included as methods we can use globally.

Setup

- In sublime, open the setup.rb file located in appium-workshop > android > spec directory.
- This is more cleanup. We've moved some functions from the spec_helper.rb file to keep it clean and easier to read. Some of these you should recognize from earlier.
- Now you will see where the "DEVICE" environment variable and thread value come into play.
- When we run in parallel library splits the tests into separate processes we need a way to connect a device to each one.
    - Thankfully the library we use creates a "TEST_ENV_NUMBER" environment variable we then match to the thread value of the "DEVICE" variable.
    - We then get the UDID from the get_device_data that matches it in the "DEVICE" variable. I know, it's a bit crazy.
- Why is the UDID important? There are couple reasons.
    - Most importantly, we pass it in as a capability so when the appium driver is initialize it knows which device to run the test on.
    - When you have multiple devices connected to one machine you need the UDID to tell adb which device to do the command against. e.g. adb -s emulator-5554 shell
    - We use them to add to filenames and test descriptions. e.g. "describe "Home Page #{ENV["UDID"]}" do"
- Take a look at the remaining file and raise your hand if you have any questions.

spec_helper

- In sublime, open the spec_helper.rb file located in appium-workshop > android > spec directory.
- Notice how much cleaner it is now. You can see the helper and setup methods.
- This is basically everything we did before in the playground, but now it is more organized and easier to read.

Rake

- Ruby’s version of Make is Rake!
- In sublime, open the Rakefile file located in appium-workshop > android directory.
- This file contains the tasks to run our entire automaton suite easily. So instead of running several individual commands or scripts, we can run one command to start everything.
- Run rake -T from the command line to see all available tasks. There is only one for this demo but you could add one later for iOS. See https://github.com/isonic1/appium-mobile-grid for an example.
- Run single process:
  - rake android[single]
  - Starts a Appium server.
  - Runs the specs in a single process.
  - Executes: “rspec spec” <- what we did earlier.

- Run in parallel:
    - rake android[parallel]
        - Creates a Appium node and Appium session for each device.
        - Splits the “test suite” into separate processes based on how many devices are connected to your machine, which is defined by the THREADS variable.
        - Parallel in the case means it will run all tests on every device.
        - This runs “parallel_test #{threads} -e 'rspec spec’”
- Run distributed:
    - rake android[dist]
        - Creates a Appium node and Appium session for each device.
        - Splits the “specs” into separate processes based on how many devices are connected to your machine, which is defined by the THREADS variable.
    - Distributed means it will distributes the specs/tests to each process reducing the total run time.

Lets Run The Specs!

- But… before we do, lets create another emulator.
    - On the command line, run “android avd”.
    - Create another emulator like the EM1 you created before. This time name it EM2.
    - The directions for mac: https://github.com/isonic1/appium-workshop/blob/master/Appium%20Mac%20Installation%20Instructions.md
    - The directions for windows: https://github.com/isonic1/appium-workshop/blob/master/Appium%20Windows%20Installation%20Instructions.md
- Make sure both the EM1 and EM2 emulators are started.
- On the command line navigate to the appium-workshop > android directory.
- Run “bundle install” from the command line. This will install all the required libraries we need for the tests suite.
- Run rake android[single]
    - Lets generate the Allure report.
        - Run “allure report generate output/allure/*” from the command line.
        - Run “allure report open” to open the report.
- Run rake android[parallel]
    - After it completes. Run “allure report generate output/allure/*” from the command line.
    - Run “allure report open” to open the report.
- Run rake android[dist]
    - After it completes. Run “allure report generate output/allure/*” from the command line.
    - Run “allure report open” to open the report.

Thank You!

- Email: justin.ison@gmai.com
- Twitter: isonic1
- Workshop: https://github.com/isonic1/appium-workshop
- Appium-Mobile-Grid: https://github.com/isonic1/appium-mobile-grid
- Flick: https://github.com/isonic1/flick