Appium Bootcamp 7: Automating Your Test Runs

Appium Bootcamp is a series of articles prepared by Selenium guru Dave Haeffner, and leading Appium contributor Matthew Edwards, for Sauce Labs. Dave also authors the Elemental Selenium website, which includes tips for using Selenium, and where you can sign up for his weekly email on the topic of Selenium testing. He is also the author of the Selenium Guidebook.

To make our tests as useful as possible, we'll want to automate when they get run. To do that, we'll use a Continuous Integration (CI) Server.

A Continuous Integration Server Primer

A Continous Integration server (a.k.a. CI) is responsible for merging code that is actively being developed into a central place (e.g., "trunk" or "master") frequently (e.g., several times a day, or on every code commit) to find issues early so they can be addressed quickly -- all for the sake of releasing working software in a timely fashion.

With CI, we can automate our test runs so they can happen as part of the development workflow. The lion's share of tests that are typically run on a CI Server are unit (and potentially integration) tests. But we can very easily add in our automated mobile tests.

There are numerous CI Servers available for use today. Let's pick one and step through an example.

A CI Example

Jenkins is a fully functional, widely adopted, and open-source CI server. It's a great candidate for us to step through.

Let's start by setting it up on the same machine as our Appium Server. Keep in mind that this isn't the "proper" way to go about this -- it's merely beneficial for this example. To do it right, the Jenkins server (e.g., master node) would live on a machine of its own.

Quick Setup

A simple way to get started is to grab the latest Jenkins war file. You can grab it from the Jenkins homepage, or from this direct download link.

Once downloaded, launch it from your terminal.

```
sh java -jar /path/to/jenkins.war
```

You will now be able to use Jenkins by visiting `http://localhost:8080/` in your browser.

Running Tests Locally

After loading Jenkins in the browser, we'll create a Job and configure it to run our Appium tests. Let's start with the Android tests first.

1. Click `New Item` in the top-left corner
2. Type a name into the `Item name` input field (e.g., `Appium Android`)
3. Select `Build a free-style software project`
4. Click `OK`

This will load a configuration screen for the Jenkins Job.

1. Scroll down until you reach the `Build` section (near the bottom of the page)
2. Click `Add build step`
3. Select `Execute shell`
4. Input the following into the `Command` input box

```
cd /path/to/your/appium/test/code bundle update rake android
```

In this set of commands we are telling Jenkins to change directories to our test code, make sure we have the necessary libraries, and then launch the Android tests.

Click `Save` at the bottom of the page, make sure your Appium Server is running (if not, load up the Appium GUI and click `Launch`), and click `Build Now` on the left-hand side of the Jenkins Job screen.

Once it's running, you can click on the job under `Build History`, and then click `Console Output`(from the left-hand panel). In it, you should see something similar to this:

```
Started by user anonymous
Building in workspace /Users/tourdedave/.jenkins/jobs/Appium Android/workspace
[workspace] $ /bin/sh -xe /var/folders/yt/h7v9k6px7jl68q81c9sqrd9h0000gn/T/hudson6140596697737249507.sh
+ cd /Users/tourdedave/Dropbox/_dev/appium/appium-getting-started/code-examples/7/1
+ bundle update
Fetching gem metadata from https://rubygems.org/...........
Fetching additional metadata from https://rubygems.org/..
Resolving dependencies...
Using rake 10.3.2
Using awesome_print 1.2.0
Using json 1.8.1
Using mini_portile 0.6.0
Using nokogiri 1.6.3.1
Using ffi 1.9.3
Using childprocess 0.5.3
Using multi_json 1.10.1
Using rubyzip 1.1.6
Using websocket 1.0.7
Using selenium-webdriver 2.42.0
Using blankslate 2.1.2.4
Using parslet 1.5.0
Using toml 0.1.1
Using appium_lib 4.0.0
Using bond 0.5.1
Using coderay 1.1.0
Using method_source 0.8.2
Using slop 3.6.0
Using pry 0.9.12.6
Using numerizer 0.1.1
Using chronic_duration 0.10.5
Using spec 5.3.4
Using appium_console 1.0.1
Using diff-lcs 1.2.5
Using mime-types 1.25.1
Using rdoc 4.1.1
Using rest-client 1.6.8
Using rspec-support 3.0.3
Using rspec-core 3.0.3
Using rspec-expectations 3.0.3
Using rspec-mocks 3.0.3
Using rspec 3.0.0
Using sauce_whisk 0.0.13
Using bundler 1.6.2
Your bundle is updated!
+ rake android
.
Finished in 38.39 seconds (files took 1.52 seconds to load)
1 example, 0 failures
Finished: SUCCESS
```

Making Sure We Have A Clean Finish

We now have a working job in Jenkins. But we're not there yet. While the job was runnning you should have seen the Android Emulator open, load the test app, and perform the test actions. Unfortunately, after the job completed, the emulator didn't close.

Closing the Android Emulator is something that Appium doesn't handle, so we'll need to account for this in our Jenkins build configuration. Otherwise, we won't leave things in a clean state for future test runs.

The simplest way to close the emulator is by issuing a `kill` command against the name of the process (ensuring that the command always returns `true`). That way we cover our bases in case there is more than one emulator process running or if we try to kill a process that doesn't exist. So let's go ahead and add the `kill` command to our existing commands under the `Build` section of our job. For good measure, let's add it before and after our test execution commands.

To get back to the job configuration screen, click `Configure` from the main job screen.

```
killall -9 emulator64-x86 || true
cd /path/to/your/appium/test/code
bundle update
rake android
killall -9 emulator64-x86 || true
```

Now let's save the job and build it again. The job will run just like before, but now the emulator will close after the test run completes.

Creating Another Job

Now let's create a second job to run our tests against iOS.

To save a step, let's create a copy of our existing job and modify the build commands as needed.

1. Click the Jenkins logo at the top of the screen (it will take you to the main page)
2. Click `New Item` in the top-left corner
3. Type a name into the `Item name` input field (e.g., `Appium iOS`)
4. Select `Copy existing Item`
5. Start to type in the name of the other job in the `Copy from` input field (e.g., `Appium Android`)
6. Select the job from the drop-down as it appears
7. Click `OK`

This will take us to a configuration screen for the new (copied) job. Let's scroll down to the `Build`section and modify the `Command` input field under `Execute Shell`.

```
killall -9 "iPhone Simulator" &> /dev/null || true
killall -9 instruments &> /dev/null || true
cd /path/to/your/appium/test/code
bundle update
rake ios
killall -9 "iPhone Simulator" &> /dev/null || true
killall -9 instruments &> /dev/null || true
```

Similar to the Android job, we're using `kill` to end a process (in this case two processes) and making sure the command returns `true` if it doesn't exist. This protects us in the event that the test suite doesn't complete as planned (leaving a simulator around) or if the simulator doesn't close instruments cleanly (which can happen).

If we save this and build it, then we will see the iPhone Simulator load, launch the app, run the tests, and then close the simulator.

Running Tests On Sauce

We've covered running things locally on the CI server, now let's create a job to run our tests on Sauce.

Let's create another copy of the `Appium Android` job and modify the build commands.

Since we're not going to be running locally, we can remove the `kill` line. We'll then specify our Sauce credentials (through environment variables) and update the `rake` command to specify `'sauce'` as a location. When we're done, our `Command` window should look like this:

```
export SAUCE_USERNAME=your-username
export SAUCE_ACCESS_KEY=your-access-key
cd /path/to/your/appium/test/code
bundle update
rake android['sauce']
```

If we save this and build it, our tests will now run on Sauce Labs. And you can view them as they happen on your Sauce Labs Account Page.

An iOS job would be identical to this, except for the job name (e.g., `Appium iOS Sauce`) and the `rake` incantation (which would be `rake ios ['sauce']`.

Outro

Now that we have our Appium tests wired up for automatic execution, we're now able to configure them to run based on various triggers (e.g., other CI jobs, a schedule, etc.). Find what works for you and your development team's workflow, and make it happen.