LAS3007 - SEM1-AE Software Test Automation and Continuous Integration

Waylon Mifsud

University of Malta

Lecturer: David Camilleri

Date: January, 2016.

1. Introduction

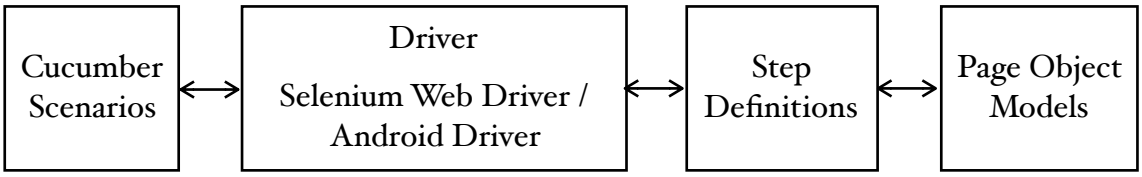
This assignment documents the development of a testing suite for the Evernote website on Mozilla Firefox and Google Chrome web browsers, along with Mobile Contact application testing using Android. The tool which is being used to interact with browser and mobile application elements from the test suite is called Selenium. For the mobile application, Appium is being used to interact between the test suite and the mobile application. Genymotion, which is an emulator for the mobile application, is run on a Virtual Box VM. During development, code was versioned to keep track of changes being done. Versioning was conducted using github and the testing suite can be found at: https://github.com/wmifsud/plas-assign. Maven configuration and how to configure the test suite can be found in the readme.md file which is versioned with the code on github along with this report.

Selenium configuration does not require any additional drivers when running tests on Firefox but an additional driver is required when running tests on Chrome. Such a file is called chromedriver and is located in the browserdriver/chrome directory of the test suite. Selenium driver is started by calling the startWebDriver method and the driver is terminated by calling the nullWebDriver method.

Selenium Grid is an extension of Selenium which provides a hub with a number of browser instances where each instance is referred to as a node. For example, one can launch Selenium Grid with 7 Firefox and Chrome instances. Hence, such a configuration would support parallel testing where multiple tests are executed at the same time using more than one browser instance.

2. Code Structure

The test suite uses a maven structure and the following diagram shows how its layers are interacting with one another:



2.1. Cucumber Scenarios feature files

Feature files are located under the test resource directory and serve as the starting point for running the tests. These files are coded using a language called Gherkin and Cucumber handles the interaction between the feature files and steps definition files. Each feature file is divided into scenarios which handle a specific test. Feature files are easy to

create since a scenario is written in simple English and is understandable as you read it hence, Gherkin is very easy to code with. An example is as follows:

```
Scenario: Create New Note
Given I create 1 note with title as test title a and body as test body a
Then the test title a is in the notes list
```

Feature files contain tags which are helpful if you want to run just one specific test, a whole feature file or all feature files. For example in the login.feature:

- If we want to run a specific scenario such as Login with no credentials, then the specified tags in the maven configuration has to contain the @1b tag.
- If the complete login.feature tests need to be run, then the @login.feature tag need to be passed on to the configuration.
- If all tests need to be run, the @all tag needs to be passed on to the configuration.

Such a configuration is helpful if you either want to run a specific set of test or if you simply want to run them all.

2.2. Driver classes (WebDriver and AndroidDriver)

Driver classes are required to communicate with the browsers and the mobile application. It links the Cucumber scenarios to the step definition classes.

For browser testing, Selenium's WebDriver class is being used. this class is being used in the Driver class. This class makes use of static methods only since the driver is being initialised at the start and terminated at the end of the tests. Methods consist of getWebDriver which will retrieve the Selenium web driver if initialised, startWebDriver which will initialise Selenium web driver with the arguments provided in the Maven configuration. A selenium web driver can be initialised in four ways:

- localFirefox: where Selenium will interact with a local instance of Firefox web browser.
- localChrome: where Selenium will interact with a local instance of Chrome web browser.
- gridFirefox: where selenium will interact with Selenium Grid which provides multiple instances of Firefox browsers.
- gridChrome: where selenium will interact with Selenium Grid which provides multiple instances of Chrome browsers.

The AndroidDriver is initialised in the MobileDriver class following the same structure as the Driver class except for the startAndroidDriver method. In this method, the

AndroidDriver is being initialised with the capabilities which show the driver with what it will interact. This driver will interact with Appium in order to simulate the tests and assertions on the mobile application.

2.3. Helper Classes

Such classes are required to carry out tasks before and after running the tests. The CucumberBeforeAfter class starts the required driver before running the tests using the Driver or MobileDriver classes. It closes the respective driver after the tests are run.

In addition to the above, this class is responsible for removing any created notes on the Evernote website or any contacts created on the mobile application. Such created notes and contacts are executed by the tests themselves and the test suite is configured to remove such generated information after tests are executed.

Note and contact deletion occur depending on the value that is assigned in the @After annotation. For example the removeNotes will trigger on the note, search, tag and 5b tests. In order to provide additional help when a test fails, this class contains methods which make use of the DriverScreenShotHelper class where upon a test failure, the test suite will take a screenshot of either the browser or mobile test which was being executed at the time.

Hence, a tester not only has the logs in hand to check what went wrong in the test but the tester will also have a screenshot when exactly the test failure occurred.

2.4. Page Object Model classes

These classes provide a layer where the web driver is being called so that the step definition classes will not need to go into extra complexity when requiring elements from the web browser.

The web driver calls are also contained in this layer in order to keep the step definitions as simple as possible. This layer also caters for certain complexities which were required to fetch certain elements and also to carry out complex assertions.

Classes were created according to the tests required. The following are the Page Object Model classes.

2.4.1. DriverPage

This is the parent class page for all other pages. It holds a reference to the WebDriver class so that all child pages do not have to call Driver.getWebDriver each time it is required but they simply call the webDriver directly. This class also contains methods which are used throughout the other child pages which are the following:

- To wait for text to be assigned to a specific WebElement.

- To wait for an element to be clickable.
- To wait for an element to be visible.
- To fetch a list of WebElements according to their class.
- To retrieve a list of delete buttons according if notes or notebooks are being deleted.
- To hover on a particular web element on the browser.
- To retrieve a confirmation button. (This was placed here instead of the ButtonPage since it is being used in multiple POM classes).
- A method to pause the execution for 2 seconds. This was required especially with Chrome since Chrome is faster than Firefox and was not giving enough time for elements to load even though the above mentioned methods are used.

2.4.2. ButtonPage

Even though there were no tests related to specific buttons, the ButtonPage class was required because buttons were being used throughout different POM classes.

The ButtonPage class contain methods which are the following:

- Retrieving the sign up button.
- Retrieving the notebook button.
- Retrieving the create note book button.
- Retrieving the confirm create notebook button.
- Retrieving the add to shortcut button.
- Retrieving the shortcuts button.
- Retrieving the table button.
- Retrieving the over flow button.
- Retrieving the options button.
- Retrieving the search button.
- Retrieving the trash button.
- Retrieving the trash header.
- Retrieving the empty trash button.
- Retrieving the notes button.
- Retrieving the tag button.

2.4.3. LoginPage

This class contains all elements required to test the login functionality of the Evernote web site which are described below:

- Retrieval of the sign in button.
- Retrieval of the sign in link.
- Retrieval of the header sign in link. This consisted of retrieving a list of elements first and then going through each one of them until the element which has the sign-in text present in the data-action is found.
- Retrieval of the header menu.
- Retrieval of the username text box to enter username.
- Retrieval of the password text box to enter the password upon login.
- Retrieval of the account menu link.
- Retrieval of the logout link.
- Retrieval of the password error message.
- Execution of logging out. This involves multiple calls to elements in the LoginPage class itself and since it is being used multiple times in the step definition layer, it made sense to create a method for it.

2.4.4. NotebookPage

This class contains all elements required to test the notebook functionality of the Evernote web site which are described below:

- Retrieving the notebook header.
- Retrieving the notebook box.
- Retrieving the notebook heading.
- Deletion of a notebook by notebook name. This required to first fetch an element list by qa-name since a notebook name is contained in elements with qa-name as class. Elements retrieved are looped and once the notebook is found, the hoverOn method from the DriverPage is called in order to make its delete button visible. Once visible, We loop through all delete buttons present on the screen and click the one which is visible which will be the one of the notebook that needs to be deleted.

2.4.5. NotePage

This class contains all elements required to test the note functionality of the Evernote web site which are described below:

- Retrieval of the number of notes.
- Retrieval of the done button which makes use of the DriverPage waitForTextInElement since the button button which acknowledges the note creation must have the text Done in order to save the note, otherwise the note will not be created.
- Retrieval of the new note button.
- Retrieval of the note title.
- Retrieval of the note message.
- Assertion that a particular note is in the list. Such a method requires to check beforehand that all notes are loaded correctly since at times, Evernote first loads the note titles as Untitled until the DOM is refreshed with the correct note titles. So this section of the code is making sure that all notes have no Untitled text before the test can continue. It then goes through all note elements and returns of it manages to find the note or not.
- Creation of a table in a note. This method is quite complex and it had to be made in such a way since at the time of development, the class names were changed by Evernote so test started to fail. Hence, the test is going through all elements until it manages to populate a list of elements which total to 36 being the max table that can be created, that of a 6x6. Once the list of elements are identified, the element is clicked according to the size of the table that is being tested.
- Retrieval of table rows and columns which are used to ascertain that the table with the correct number of rows and columns were created.
- Retrieval of sort option which goes through a list of elements by selector option.
- Assertion of sort method which makes sure that list is sorted according to the order parameter passed and also if the sort needs to be carried out using the date or the title name. Method first creates a sort list according to the TreeMap sort and to the parameters passed. The method then compares the list created to the actual list that exists in the DOM.
- Deletion of all notes, method is used by the CucumberBeforeAfter class to delete the notes after each test related to note creation.
- This class also stores an internal class called Table which is returned by the retrieveTable method so that it can be asserted in the Note class.

2.4.6. SearchPage

This class contains all elements required to test the search functionality of the Evernote web site.

The only method it has is retrieving the search box element.

Thanks to code reusability, such test methods required on the search functionality are already being used in other testing scenarios such as note.

2.4.7. TagPage

This class contains all elements required to test the tag functionality of the Evernote web site. Such methods are the following:

- Retrieval of the create tag button.
- Assigning a specific tag to a note. The method first identifies to which note it should assign the tag. Once identified, the note is selected, its empty tag clicked and the tag name is sent by using the Actions class which mimics keyboard entry to that particular element and a new tag is created and linked to that note.
- Selecting a particular tag. A list of tag elements is retrieved and once a tag name is identified having the same name as provided to the method, the tag is returned.
- Retrieval of the delete tag.

2.4.8. TrashPage

This class contains all elements required to test the trash functionality of the Evernote web site. Such methods are the following:

- Emptying the trash can. This is done by identifying trash button, hovering on it and clicking the web element.
- Retrieval of the trash header.
- Restoring a deleted note. Such a method is searching for a list of notes which were trashed. Once identified, the hoverOn method is called to hover on the element and the restore button becomes visible. Once it is visible buttons are retrieved according to the tag name where 2 buttons exist, delete or restore. The restore button is then clicked and the method waits for the notes count to become 0 since that was the only note in the trash.

2.4.9. MobileDriverPage

The class acts like the DriverPage but for the AndroidDriver. Such a class also contains methods to delay an operation and also to wait for an elements to become clickable.

2.4.10. ContactPage

This class contains all elements required to test the contact functionality of the Contact mobile application using Appium.

Such methods are the following:

- Retrieval of several icons such as contact, navigate up, delete, confirm, edit and favourites.
- Retrieval of the contact text. Method accepts two strings; text which needs to be input to a particular web element and attribute which is the attribute stored along the element. Once element is identified, it is returned to the Contact step definition class.
- Retrieval of the large contact title.
- Retrieval of the header from the info web element.
- Retrieval of the text from the info web element.
- Method to assert that specific text is under a particular attribute.
- Method to assert that a particular contact can be found in the contacts list.
- Retrieval of the contact list.
- Retrieval of the more options image.
- Retrieval of a web element by attribute text.
- Retrieval of a specific contact by contact name.
- Retrieval of the edit elements.
- Retrieval of a web elements that needs to be updated.
- Retrieval of the contact message.
- Retrieval of a web element under the favourites option.
- Deletion of contacts used by the CucumberBeforeAfter class to remove all contacts after each test scenario.
- Reset mobile application. In certain cases, it was required to terminate the android driver and start it up again in order to simulate the home button on the mobile. This was required to carry out other assertions such as checking contact creation.

2.5. Runner classes

The RunCukesTest class is the main executor of the Cucumber scenarios along with the java unit tests to run. The configuration makes use of a CucumberOptions annotation where the features location is specified, glue specifies where the java unit tests are located and additional plugins that are required to execute the tests.

2.6. Step definition classes

Step definition classes are the classes which interact with the feature in order to determine which particular java unit method requires execution. This interaction is being handled by Cucumber. Web elements are being fetched by the POM layer and this layer is interacting with these elements such as clicking them, sending keyboard entries to them, hovering over them, etc. The following annotations are used:

- @Given is used at the start of every scenario.
- @When is used to describe and execute a particular event.
- @And is a continuation of @When
- @Then makes sure that the end result of a particular test is what was expected.

The above annotations are very helpful especially when it comes to re-usability since they make use of regex expressions. The following methods are used multiple times to carry out or simulate actions on different elements during different scenarios specified by the feature files. The following points are the step definition classes and their methods.

2.6.1. Button

The Button class makes use of the ButtonPage class to fetch elements that it requires. The button class has only one method which is the button click and is always called when there exists a scenario which requires a button to be clicked.

Basically it has a switch statement for every button being used, it fetches the button required via the button page class and simulates the click.

2.6.2. Login

The Login class is used by the Login feature to carry out tests based on the Gherkin code.

It makes use of the following methods:

- signInClick: Since Evernote has three methods of signing in, this method is taking care of all scenarios in the Login.feature for logging in using a switch statement.
- inputEmailAddress: to send the email address to the username field.
- inputPassword: to send the password to the password field.
- loggedInCheck: checks that the account menu link web element is displayed.
- errorDisplayedCheck: makes sure the correct euro is displayed when sending incorrect login details.
- login: simulate login with static email and password fields.
- logout: simulates the logout via the LoginPage class.

2.6.3. Navigation

This class is only used to load the Evernote web site via the URL passed through the scenarios.

2.6.4. Note

This class is used by the Note.feature. Such methods include:

- createNewNote: this method will create new notes through the Evernote web site depending on the parameters passed and by calling elements via the NotePage class.
- assertNoteCreated: Method will assert that a note having the provided title exists in either the notes or shortcuts list.
- createTable: Simulates table creation in a note for which the size cannot be greater than a 6x6 table.
- assertTableCreation: asserts that the table was created and with the correct size.
- selectSortMethodOption: retrieves the element and clicks it.
- assertSortingIsCorrect: calls note page class's assertSortMethod to make sure list is sorted in correct order.

2.6.5. Notebook

This class is used by the Notebook.feature. Such methods include:

- inputSearchBox: used to create a notebook with the provided text.
- assertNoteLinkedToNotebook: assertion is carried out by clicking the note and making sure that the notebook heading of the note is showing the notebook provided.
- deleteNotebook: deletes a notebook having the name set to the provided notebook string.

2.6.6. Search

This class is used by the Search.feature. The only method is inputSearchBox which is used to enter search text into Evernote's search text box.

2.6.7. Tag

This class is used by the Tag.feature. Such methods include:

- assignTagToNote: makes use of TagPage's assignTagToNote method to assign the provided tagName to the provided note.
- selectTag: Simulates tag selection based on the provided tag name.
- deleteTag: deletes a tag with the provided tag name.

2.6.8. Trash

This class is used by the Trash.feature. Such methods include:

- deleteAllNotes: removes all notes by calling the TrashPage method.
- emptyTrashCan: call emptyTrashCan TrashPage method.
- assertTrashEmpty: checks that the trash can is actually empty.
- restoreNote: restores the provided note name by calling restoreNote TrashPage method.

2.6.9. Contact

This class is used by the Contact.feature. Such methods include:

- selectContactIcon: retrieves a particular web element from the contact page and simulates a click when returned depending on the regex expression. This method is similar the button method however, since this Contact.class relates to Mobile testing, it was kept separate.
- placeContactText: method retrieves a web element according to the attribute passed into the regex and sends text to that element according to the data passed in the regex expression.
- contactCreationAssertions: Method carries out assertions according to the attribute passed.
- updateContactText: method used to update text in a particular element.
- noFavourites: Method asserts no contacts exist under favourites.

3. Test Maintainability

Tests can easily be maintained at feature and at step definition level since the interaction between the two levels occurs thanks to the Annotations and regex at step definition level. Hence, knowledge required at this stage would be that of regex expressions and the understanding of interaction between both layers. The IntelliJ IDE also makes it easier to understand such interaction since regex expressions are highlighted in blue at feature level and if you click on a particular line in a scenario, IntelliJ will take you to the test method which matches that line.

Code gets slightly more complex at POM level where one needs to understand how elements are fetched from the DOM. Such elements are searched using tools like Firebug (Firefox) or Developer tools (Chrome) and the Selenium Web Driver needs to be called to fetch such elements.

It is important to plan on reusability wherever possible. There are many methods at step definition level which are called more than once in the same or even in different scenarios. This helps to maintain easier code and also simpler to track any bugs which might exist in the test suite especially when dealing with assertions.

4. Installation and Configuration

4.1. Jenkins

Jenkins installation is quite simple since all is required is to download the war file from the Internet and double click it once downloaded. Jenkins will auto deploy itself. Once loaded, Jenkins needs to know from where it will retrieve the code. This is done by the following steps:

- Click on new job.
- Name the job and select maven project.
- Click on the newly created job.
- Click on configure.

- We need to tell Jenkins where the project is located and this is shown in the following screenshot:

- We also need to inform Jenkins which configuration it needs to run as shown below:

- Once completed, click on apply and save at the bottom of the screen.
- Build the project by clicking on build from the menu on the left.

4.2. Selenium Grid

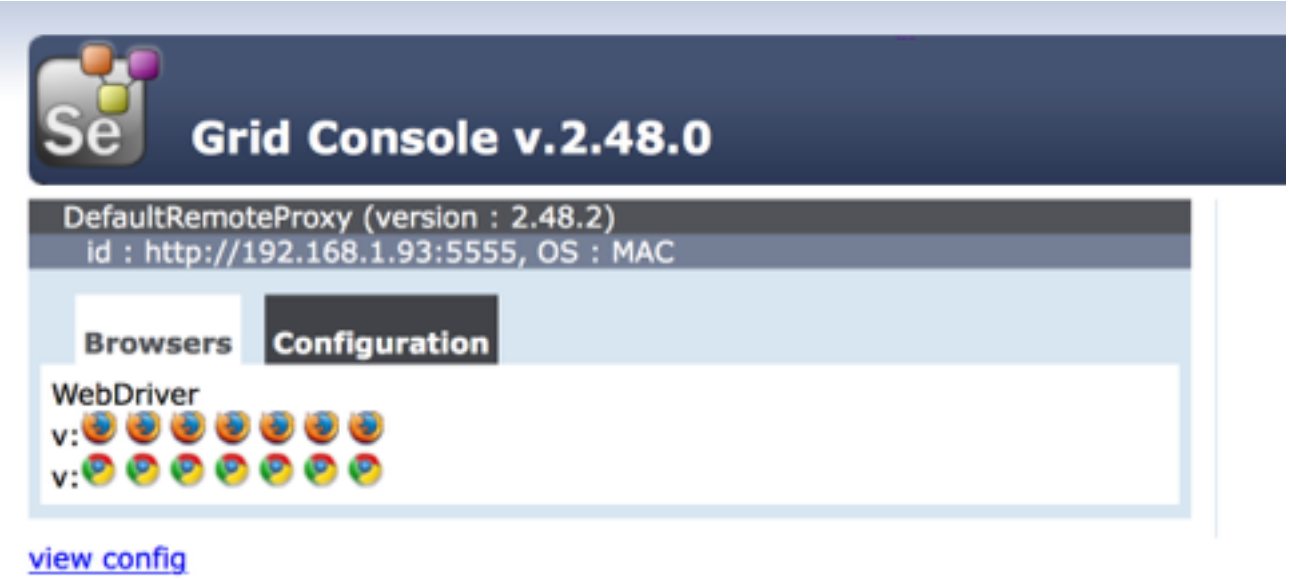
The Selenium Grid jar file needs to be downloaded. The version used for this assignment was 2.48.2. Once downloaded, Selenium Grid is started by starting the node with the following command: java -jar selenium-server-standalone-2.48.2.jar -role hub

Once the hub is started, go to http://localhost:4444/grid/console and the following should load up:

Nodes will then need to be started up by running the following command:

java -jar selenium-server-standalone-2.48.2.jar -role node -hub http://localhost:4444/grid/register -browser "browserName=firefox,maxInstances=7,platform=MAC" -browser "browserName=chrome,maxInstances=7,platform=MAC" -Dwebdriver.chrome.driver={chrome driver directory}

Once the nodes are started, they will auto hook to the hub and the hub should be updated as the following shows:



The above shows that Selenium Grid has 7 Firefox instances and 7 Chrome instance configured and ready to be consumed by our Evernote testing suite.

4.3. Mobile Application

The following applications need to be downloaded and installed to run the mobile application:

- GenyMotion.
- Oracle VirtualBox.
- Android Virtual Device: the OVA file having the Android Mobile application.
- Android SDK with the Android SDK Build, Platform and Tools plugins.
- Appium.

Once Genymotion is loaded onto the Oracle VirtualBox and the OVA file is fed to Genymotion, the VM can be started via Genymotion by double clicking the VM as shown below.

Once the VM loads, the mobile application is started and should look as follows:

Finally, Appium can be started by opening it and clicking the Launch button. Once Appium is launched, the mobile tests can be started since Appium translates the commands being sent by the test suite to be executed on the mobile application.

5. Problems encountered during installations

Since Selenium Grid commands are quite long to especially when starting the nodes, it was quite problematic until the correct syntax was identified in order to load them.

The mobile application setup required installation of a number of applications and this required understanding of how the applications are interacting with each other including the AndroidDriver running on the test suite application. Android SDK needed to be properly set including its environment variables ANDROID_HOME and PATH. Without Android SDK installation and correct configuration, Appium would be unable to send commands to the mobile application.

6. Problems Encountered During Element Locating

Major problems were definitely related to page rendering since even though elements load up on the web site, they might be not fully initialised since tools like Javascript would still need to carry out the final rendering of the website. This caused issues such as clicking a particular element where Javascript would still be loaded. Once the Javascript ends loading, Selenium would already have interacted with the element but the web site would require another interaction to that element since Javascript stopped the initial interaction.

Hence, Selenium tools such as to wait for element visibility or to wait for particular text to be present in an element would need to be called in order to ascertain that a particular element can now be interacted with. Such wait interactions are called Explicit Waits and their job is to wait for a particular element to be in a particular state or contain particular text.

Another problem was the browsers themselves. It was identified that Chrome is slightly faster than Firefox and hence, certain tests were working fine on Firefox but were failing on Chrome. Other checks had to be introduced in order to make sure that the tests developed worked on both browsers. It was also noted that when logging in, the sign in header option was behaving differently on Firefox and Chrome. This was later on fixed by Evernote themselves.

Another problem related to changes being done by the Evernote team which were causing tests to fail. Hence such tests needed to be identified and analysed in order to understand what changed in the website. For example the test related to table creation, the Evernote team changed classes for the table thus failing our test since originally, the test depended on the element classes.

7. Further improvements to the framework

The Evernote website was not fully tested and the following show additional tests that can be created in order to test as much as possible the site. Please note that a particular website cannot be 100% test automated mainly because the cost and maintainability required to keep such a test suite up to date. The Evernote website is continuously being updated with new features and functionality which still can cause current test scenarios to fail due to new features introduced. Additional testing features which can be considered are the following:

- Notes sharing with other Evernote clients.
- Notes font customisations and editing such as bullets and numbering, checkboxes, insertion of links, adding of attachments, indentation and alignment of text.
- Expansion of notes to make utmost use of the screen when editing a note.
- Additional information showing note creation date, note last updated date, etc.
- Reminder functionality.

As for the mobile application testing additional scenarios can be introduced for the contacts application which are the following:

- Import of a number of contacts from a storage or from a sim card.
- Signing in to an account.

There are also other mobile applications which can be tested out such as the following:

- Calculator application.
- Mobile search functionality.
- Calendar including appointments and reminders.

It is important to note that when new test scenarios are introduced, the current architecture needs to be followed in order to keep the test suite as clean and readable as possible.

8. Conclusion

Selenium Web Driver and Android Driver are two of the many ways that web sites and mobile applications can be tested with. Even though they are not the perfect tools for software automation, it is quite easy and straight forward to retrieve elements with Selenium. Unfortunately, things get complicated when testing with elements which are quite difficult to identify, a case in point is table creation in case of the Evernote tests however, you can still get along with them.

Thanks to Java and its interaction with Selenium, they have the ability to loop through elements and ascertain that specific elements are in a certain state.

Mobile testing also proved quite easy thanks to the Selenium Android driver and Appium. Elements were easy to identify with the UIAutomationViewer tool which takes a screenshot and outputs all elements on the mobile screen.