Selenium Web Application Automation

By

Pritish Kumar Roy
Summer Internship
Guide - Samin Yasar
Senior Software Engineer
ZS Solution Ltd

WHY AUTOMATE TESTING?

In today´s fast moving world, it is a challenge for any company to continuously maintain and improve the quality and efficiency of software systems development. In many projects, testing is neglected because of time or cost constraints. This leads to a lack of product quality, followed by customer dissatisfaction and ultimately to increased overall quality costs.

WHAT IS SELENIUM?

Selenium automates browsers. Primarily, it is for automating web applications for testing purposes, but is certainly not limited to just that.

It is also the core technology in countless other browser automation tools, APIs and frameworks.

SELENIUM WEBDRIVER

- The primary new feature in Selenium 2.0 is the integration of the WebDriver API. WebDriver is designed to provide a simpler, more concise programming interface in addition to addressing some limitations in the Selenium-RC API. Selenium-WebDriver was developed to better support dynamic web pages where elements of a page may change without the page itself being reloaded. WebDriver’s goal is to supply a well-designed object-oriented API that provides improved support for modern advanced web-app testing problems.

Junit Framework

- Junit is a framework for unit level testing and Selenium are functional level test, because Unit and Function level tests are completely different. Having said that, it doesn’t mean that we cannot use Junit for writing our Selenium tests.

Sample Code

```
package unitTests;

import java.util.concurrent.TimeUnit;

import org.junit.AfterClass;
import org.junit.Assert;
import org.junit.BeforeClass;
import org.junit.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;

public class SeleniumTest {
                    private static FirefoxDriver driver;
                    WebElement element;
```

```
            @BeforeClass
      public static void openBrowser(){
          driver = new FirefoxDriver();
          driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
                          }

            @Test
      public void valid_UserCredential(){

                          System.out.println("Starting test " + new
Object(){}.getClass().getEnclosingMethod().getName());
                  driver.get("http://www.store.demoqa.com");
                  driver.findElement(By.xpath(".//*[@id='account']/a")).click();
                  driver.findElement(By.id("log")).sendKeys("testuser_3");
                  driver.findElement(By.id("pwd")).sendKeys("Test@123");
                  driver.findElement(By.id("login")).click();
                  try{
                                       element = driver.findElement
(By.xpath(".//*[@id='account_logout']/a"));
                          }catch (Exception e){
                                      }
                  Assert.assertNotNull(element);
                  System.out.println("Ending test " + new
Object(){}.getClass().getEnclosingMethod().getName());
      }

            @Test
      public void inValid_UserCredential()
      {
                          System.out.println("Starting test " + new
Object(){}.getClass().getEnclosingMethod().getName());
                  driver.get("http://www.store.demoqa.com");
                  driver.findElement(By.xpath(".//*[@id='account']/a")).click();
                  driver.findElement(By.id("log")).sendKeys("testuser");
                  driver.findElement(By.id("pwd")).sendKeys("Test@123");
                  driver.findElement(By.id("login")).click();
                  try{
                                      element = driver.findElement
(By.xpath(".//*[@id='account_logout']/a"));
                  }catch (Exception e){
                                      }
                  Assert.assertNotNull(element);
                  System.out.println("Ending test " + new
Object(){}.getClass().getEnclosingMethod().getName());
      }

            @AfterClass
            public static void closeBrowser(){
                          driver.quit();
            }
}
```

Annotation before class will tell Junit to run this piece of code before starting any test. As you can see we did not start any browser in the test method '*valid_UserCredential()*' & '*Invalid_UserCredential()'.* So our test need a browser to execute steps. With the help of the *BeforeClass* we would be able to open the browser before running the test.

This is a simple test of Log In functionality of our Demo application. But what you can notice in it is try/catch block & assert statement. This try block is to test the presence of the element, if it does not find it the element will remain as null. With that assert statement will verify that the element is not null. The basic rule of assert statement is that they act only on failure. We would be soon writing a complete chapter on Junit Asserts.

This is to tell Junit engine that execute this piece of code once all the test has been executed. This will finally close the browser down, as after this no test would be needed any browser to execute any steps.
There are other annotations as well like ***@BeforeTest*** & ***@AfterTes***t. This is your exercise to use this annotation in your selenium examples.

SELENIUM REPORTS

In the Selenium manual it is written that most testers will sooner or later end up developing their own reports instead of or in addition to those reports which are presented by frameworks.
Who will use the reports? Developers, testers, manager, client.

A variety of people, with a variety of skills will have to work with reports, and it would be advantageous to have a separate type of report to match each person. Somewhere it only has to be a small message mentioning an error, somewhere there is information on how to reproduce this error.

One more wish on the list– a comprehensive description of the actions taken to reach an error, and their order. What actions is it? For example, to check login, we use these simple steps:
- open page
- fill in fields
- press the button
- check if 'logout' exists

To reproduce an error, it is enough to repeat those steps you took in the first place, they should be in your test report. This list of steps can contain a stack of method calls.

Let's assume that your test must login several times during the testing process. Obviously, all these simple steps will be executed every time, but is it necessary for them to be listed in the log every time? Majority of the time, you only need the list once – when you actually test login functionality. For all other instances, it would make sense to combine these steps into a higher order step – 'login' and show this in the reports instead of the whole chain of events.

Selenium has a function for receiving screenshots. We think reports should contain screenshots. There are many articles which describe how to take a screenshot immediately after an error occurs. But ideally we should be able to view a screenshot not only after an error but also before it. But it is almost impossible to predict when an error is about to occur, and the only solution I see, is to constantly take screenshots.

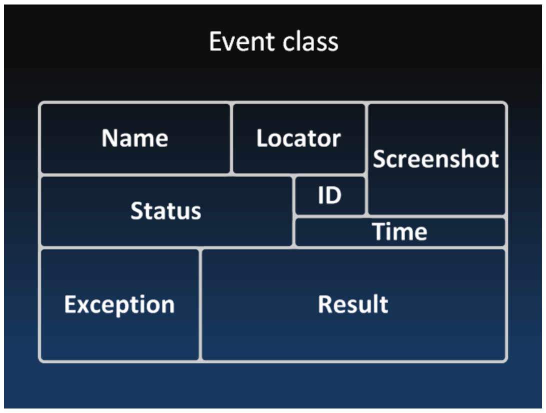
Junit, The deciding factor was the fact that it could run tests in parallel, and that's was exactly what we were looking for. The regular way for JUnit to inform you of errors, it to use a variety of assertXXX. If the requirements of an assert are not met, an exception is fired out and the test is considered failed. So any test from JUnit could have two statuses: fatal or ok.

REPORT

- Generate different types of reports for different uses/users
- A report should be able to group simple steps into more complex ones
- It should take screenshots before errors
- It should allow alerts to be filtered by severity
- It should be as detailed as possible

DIFFERENT VIEWS

This criteria means that the report shouldn't be written as the test is running, like a log. The test should remember everything. When the test is completed we have all the information and can decide what to do. First, the test can generate a report straight away. It can also stack all the data into intermediate format. From this format we can view the report with the help of a viewer or generate a report with a special generator when we need it.



A list of actions, each with some attributes and results. Logically, we should store actions, or ‘events’ in the form of a list (ArrayList) of Event type objects. These objects will store all the information of an action; name, start time, end time, target (for example, a locator), a value applied to a target (for example, a value entered into input field), expected action result, actual result, status (OK, ERROR, FATAL), error details, error from triggered exception (if an action triggered one), names of screenshot before and after action.

DETAILED REPORT

The report should be aware of the action’s hierarchy, that these actions are a part of a higher level action. Unfortunately, the report cannot find out about these dependencies without our help. This means that we need to somehow indicate which actions contain other actions during logging. We use this approach: at the beginning of each action we open an event. What does this mean? Basically a new object of Event class is created. It is added to our list of events, and some fields are filled out, at least the name and time. Let’s say we opened an event Event1. We are ready, an action has begun, the event is open.
At the end of an action, which is matched with an event, we close the event. What does this mean? We write down the results of the action: status, value, etc.

Separate Class are written wrappers for Selenium functions, which includes wrappers of Selenium methods with identical to Selenium names. This class is called CommonActions. Tests never directly call the driver methods, but interact with CommonActions methods.
Wrapper’s besides working with events, it also has several other functions.

BEFORE AND AFTER SCREENSHOT

In the first versions of our tests, we took screenshots immediately after an error. But errors can come up on different levels, like an exception generated by the driver command due to the missing element – an obvious error on this level, but on the next it may be considered insignificant. Or significant? Before we ended up taking screenshots of an error on every level, ending up with doubles.

Back to screenshots BEFORE errors. As I have said before, as we can’t predict an error, we need to constantly take screenshots. What do we mean by constantly? Take a screenshot after every line of code? That would be too complicated. The most obvious route is to take screenshots after every Selenium command, which can somehow affect the look of the page. What are these commands? The most obvious are; typeText(), click(), …andWait() methods.

DIFFERENT TYPES OF REPORT

Because we collect information while testing, we can freely create a few types of reports, or dump it in an xml file and use it later with a generator or a viewer. We decided to make HTML reports straight away, so we could either look at the most global issues, like the amount of errors, or, if need be, dig deeper. HTML + JavaScript easily solve the last problem on the list. The report should allow for filtration of alerts based on their importance (you can see a drop down list on the report).

SUMMARY

Tests have to be repeated often during development cycles to ensure quality. Every time source code is modified tests should be repeated. It may be tested on all supported operating systems and hardware configurations. Manually repeating these tests is costly and time consuming. Once created, automated tests can be run over and over again at no additional cost and they are much faster than manual tests. Automated software testing can reduce the time to run repetitive tests from days to hours. **A time savings that translates directly into cost savings.**

Shared automated tests can be used by developers to catch problems quickly before sending to QA. Tests can run automatically whenever source code changes are checked in and notify the team or the developer if they fail. Features like these save developers time.

THANK YOU