Manoj Kumar- From Call back's hell to using Async Await: Automated testing with JavaScript

ManojKumar: Welcome everyone again. We are here to see about a primer on Selenium WebDriver JavaScript and Protractor titled From Callback's Hell to Using Async Await: How Automated Testing in JavaScript to Understand. For those of you who is already using JavaScript for automation, the will know what the pain of using the callbacks and promises. For those of you starting new, there's something exciting for you to get started. The timing is right, as JavaScript is evolving at a right pace for you to get started.

As we discussed, first we looked through overview the JavaScript for testing. Then, we will move onto BBD in JavaScript. Then, [inaudible 00:00:48] Selenium WebDriver in JavaScript and then Protractor.

Now, using JavaScript for end to end testing, really? How many of you got into situations like this, discussions with colleague or during a chat, when saying about, "Okay, I'm using JavaScript for automated testing"? Because people know JavaScript was [inaudible 00:01:13], like say for example like myself when I started writing automate test in JavaScript like three years ago when there wasn't enough methodologies to write automated tests in a more breathable way. It was all back then callbacks and that's exactly the situation as showed in this meme.

Now comes again the burning question, "Why JavaScript for testing?" There are a lot of reasons now to say, "Okay, it makes sense to use JavaScript for testing". Why? Because as we all know that most of the front ends, the UI developers with shiny new UIs are being developed using the JavaScript frameworks like AngularJS, ReactJS and Ember and VieJs, to name a few. That helps a better Dev and QA pairing, where developers could help you writing automated tests or make some enhancements to the framework and whatnot.

To put forth a different a view, the UIs the face of customers, digital is being the world now, are the clients, which is made on JavaScript. There are also projects or people are trying to move towards using Node as a backend. Given that, having a client and also the server in JavaScript, why not tests?

We may have been situations like the UI has been developed in JavaScript and the back end could be also in JavaScript and the automated test framework could be in Java. Why? Because automated testers feel comfortable with Java.

That's not the case anymore. It's in order to have a better dev QA pairing in a nodule project. Say, for example in a BDD enrollment. It all makes sense to have JavaScript for automation testing, thus making an end to end full stack JavaScript shop.

With that, JavaScript is evolving. Here are numbers, which I downloaded that starts for about one year. You can see the JavaScript downloads for the Selenium WebDriver tool starting from December last year to 19th of this month. I think it's dipping down a bit. That's due to, maybe due to the location time. If you see the downloads, the numbers are about 27 million downloads, averaging about like 2 or 2 and a half million downloads, amount that's huge, which quite eventually let us know that JavaScript is evolving and JavaScript is awesome, but it also kind of sucks.

Why? Because of the asynchronous nature of the language. One of its core strengths is also its Achilles heel. What is Achilles heel? Achilles heel is a weakness in spite of overall strength, which can lead to the downfall. That's exactly the asynchronous nature. We will get onto more in detail about why this is leading to a downfall and specifically how it affects the test automation world.

It's simple to get started. Before we get started with the Selenium WebDriver or [inaudible 00:05:01] about it or to see about BDD and JavaScript, let's step back and revisit an important topic.

That's the UI element locators. This is how we usually, I start or train people who are new to automation and tensor ring. At least 40% of the people hearing this talk will be beginners with automation and specifically in JavaScript world. This is something, important aspect, given that you have a basic understanding of a language, mean any language like Java or JavaScript, the next thing for you to get started about automation is knowing about how to uniquely capture element locators because we as human testers know functionally which button to click on and which link to click on and what to type and where to type and et cetera. Whereas in automation, it is a robotic process. We need to instruct the bot to click on a specific button, say 'Foo', a specific link, say 'Bar'. That's done by element locators, which is nothing but the document object, which is on the HTML page.

Say, let's take an example of a login page where there is a user name field, a password field and then a sign in button. The UI developers would develop a text to text box and then a button. If you inspect the web page, say a Chrome Dev Tools or a Firebug in Firefox, you might see those front end code, which will have different tags like 'Do' tags, 'Pan' tag.

You have a lot of options to locate an element uniquely on a page. If you're taking WebDriver, we can select an element through ID, CSS, Xpath, tag name or link text.

There are also a few options for Protector like binding, model, repeater and options. We'll get into that little later, but these are some of the options that might help you to get started.

Also, I'll bring in some analogy between how a manual testers works and also how automated test works. If you see this table, on the left I have the manual and then on the right I have the automated infraction. This is how it looks like and this is how I would recommend users to visualize and start writing automated tests. That's easy.

Say the first step would be to, a tester opens a browser and loads a URL of the application under test. The corresponding automated test's line would be, as you see, we are talking about WebDriver here, it's like this. WebDriver driver, new Chrome driver. Then, the next line would be 'Driver.get' and then any website URL where the application under test is hosted.

Now, the first line, 'WebDriver driver new Chrome driver' will exactly open the browser that we are asking for. Here, we are asking for Chrome browser. Hence, it is Chrome [inaudible 00:08:17]. Similarly, if it is Firefox driver, it will be Firefox. Sorry, the other way. Say, if it's a Firefox browser that we want, then we would be giving a Firefox driver.

Now, there will be different operations that we would be doing on the webpage, different interactions like clicking on a button, typing in and everything. That's exactly what we see here. The tester manually clicks on a button and then the corresponding automated code will be 'Driver.findElementby.cssSelector', and then click.

To explain a bit more, what happens, this driver is something about the instance of the WebDriver. We are asking Selenium WebDriver, "Hey driver, this is the element that I want you to find". After finding, do this operation. That could be 'Click' or that could be 'Type keys'.

This is the driver instance that caught from the Chrome driver. We're using the 'FindElement' call, which takes an argument of the 'By' selector. It's just that 'By.selector' or it could be 'By.id' or it could be 'By.xpath' as we saw in the previous slide. Then we use any APIs like 'Click' or 'Type keys'. Similarly, one common operation would be to select options from the drop down. For that, we have something like this that is little bit different. 'New Select driver.findElementBy.id', and then 'Select by VisibleText or select by options'. There are a couple of options for you to try, but in general, this is how it looks and this is how we should visualize when we start writing automated tests.

When you're starting new and you this have sort of visualization, it will be simple to get started. For those of you who think, "Okay, automation is a big thing and it's complex to learn. We need more programming and that knowledge", a programming knowledge is required, but what more than that is required would be all about how do perform that interactions very well. When I say "Very well", I'm trying to put a point that writing and automated test that is not flakey. Enough of introduction about how JavaScript testing world is done and a little bit of a revisit on how the element interactions can be done. Let's move on to BDD in JavaScript.

BDD in JavaScript. As we all know, in general BDD is a behavior driven development, which is all about collaboration and emphasis more on the behavioral aspect of the end to end test. Basically, if you take BDD and other languages like Ruby or Java ... We had a BDD framework like Cucumber where we'd had a feature file where we defined scenarios in a given when-then format, but if you really look at in JavaScript world, we really didn't had any that sort of framework where it allowed us to write given when-thens, which we called scenarios from the product owners or business owners, but all we had was the Jasmine and Mocha.

It's quite debatable whether Jasmine and Mocha is really a BDD framework, but let's not get into that, but that's all we had in JavaScript. It started as a unit testing framework, like say for example, for Protractor, we had Jasmine and for Selenium WebDriver, we had Mocha. That is how it gets started.

Jasmine and Mocha is really a simple syntax and is really all about 'Describe' and 'It' blocks, where 'Describe' is a top level test suite description and the 'It' blocks are nothing but the test step level description. Both these blocks carries two main parameters, one being a string. It could be a test step for 'It' block and a test suite name for the 'Describe' block. Being unit testing framework, it also had support for mocks, which could be quite useful if you're having a, say, [inaudible 00:13:01] is front end where you can have the unit tests return on Jasmine and we could mock those services and get data and write the automated tests in full stack manner, but then came the Cucumber for JavaScript. There we go.

I tried my hands on with the CucumberJs on Protractor and it really worked well. I'm quite happy that the Cucumber JavaScript came in to rescue writing the BDD style in JavaScript.

With that, enough of BDD in JavaScript in brief. Now, Selenium WebDriver with JavaScript. I just remembered I made small mistake here. If you see, it was actually 'Type keys', but it's actually 'Send keys'. There was a type-o, but then just recollected that.

Now moving forward, Selenium WebDriver with JavaScript. What NodeJs WebDriver bindings different from other WebDriver language bindings? It's all about asynchronous. It's nothing about Selenium WebDriver binding. It's coming from the language that we use. We know almost all [inaudible 00:14:32] parts of the JavaScript or asynchronous. When we mean asynchronous, we know that two or more things not happening at the same time. It might be a little bit difficult for test automation. For test automation or any testing process, it should be a synchronous sequential flow.

Say, for example, given a piece of a code to a Google search or logging in, the action should be sequential. Say, for example, opening a browser, typing in the URL, entering a search text, clicking on the enter button. These sequenced actions should be performed sequentially in order to have a proper output that's expected to validate the functionality is working fine or not, but in an asynchronous nature it becomes a little difficult. That's how JavaScript was weird back years. The asynchronous nature made all that difficulties, but also they had different callbacks and promises mechanisms to handle that.

Let's not deep dive into it. On a brief level, having a Selenium WebDriver JavaScript code, which callbacks will exactly look like this. Imagine, just imagine the script just for the Google search is as big as like 10 lines of code and what would happen if you really had a big end to end test, like say, getting an insurance code or performing a transaction or adding items to the cart and whatnot. Callbacks is nothing but passing in function as an argument to the next function.

As you see here, 'Driver.get www.google.com' and then function and then [inaudible 00:16:36] argument. That's the next step. This will help perform the actions sequentially, but as I said, this will not help in long run when we have more lines of code or more number of interactions to be performed on the webpage, so this isn't recommended. Now, in general, this is where people writing code using callbacks.

Then, came the promises. As the title states, "From callbacks hell". That's exactly what we saw in the previous slide. It's really a hell and it will really hard to debug. Now, the next thing is the promises. Promises is little bit like a classic. It's a classic solution to solve the asynchronous problem in the JavaScript, where it had a [inaudible 00:17:39] statement, where if you see we have our driver for browser and then we have 'Driver.get' the URL and then '.then' function and then an arrow key and then 'driver.findElementBy.name' and then the locator.

This will allow JavaScript to perform the actions sequentially. Say, for example, the first step will be 'Driver.get', loading of the URL. Then, as the same says, then follows the next actions. Then we have different 'Then' statements to follow what, the next step sequentially. Again, this will be a little bit verbose. It'll also become a little bit hard for us to debug.

Now, what happened with the Selenium code team is that they came up with something like Promise Manager. With that Selenium WebDriver having a Promise Manager, we were able to write a code something like this. This is exactly how you do in Java too.

The Promise Manager is a complex engineering. What happens is, the instructions that you saw here, starting with a piece of code, so the 'Driver.get', we'll actually store it into a scheduler. Then, the next step, 'Driver.findElement' and then the 'SendKeys' will also be stored as the next command in the scheduler. When the first command executed, then it will start the next one. It acted as a event loop, first in, first come serve.

That's how it was there, but now, it's deprecated. Why? Because, in favor of the native language constructs. As we all know, JavaScript is evolving and now it is without even the callbacks and Promise Manager. We will be able to write a better code in JavaScript. That's exactly from callbacks to promises to Async Await, the most awaited topic, Async Await.

These are small gifs that I would like show you, which was stolen from a web and couldn't find the proper name to give credit, but I hope everyone can find that if you search by the title 'Callbacks to Promises to Async Await'.

This how it looks. The callbacks as you see, the parentheses, then the arrow keys and then you can see the [inaudible 00:20:29] statements for using the promises. Then, comes the Async Await. That's a simple five second gif file. As we all know, the picture is about 1,000 words and that's exactly this.

Async Await. Here we go. Async functions, they allow to write promise based code as if it were synchronous, but without blocking the main threat. The make our asynchronous code less clever and more readable. It will generally grab from C#. With Async Await, no callbacks, no promises, no control. It's an ES2016 language specification. These ES2016 articles, something that you come across is the spec that's maintained by the PC39 group. They are the main guys who develop or write the spec for the JavaScript functions. Our hats off to those guys.

The main features of the Async Await that will help the automated testing world or beat any such developments happening using JavaScript, that is very concise. We don't have to write the '.then' statements, create an anonymous function to handle the response or give a name data to a variable that we don't need to use or a way to nesting of our code and a better error handling. Async Await makes it finally possible to handle both synchronous and asynchronous errors with the same construct and intermediate values, readable conditionals.

When I mean 'Intermediate values', you probably found yourself in a situation where you call a promise one and then use what it returns to call promise two and then use the results of both promises to call a promise three. This exactly could have happened in the test automation world. Say, for example, for probably, for those of use who already test automation using Selenium WebDriver with JavaScript, the 'Get pics' function exactly fits in there.

[inaudible 00:23:00] when you use 'Driver.findElement' with some value and then a 'Get text' ... In order to resolve the value, you need to resolve ... In order to get the value, you need to resolve the promise. That's easier now with the Async Await. The main thing, the debugging, you can set break points and error functions [inaudible 00:23:21] expressions. With Async Await, you don't need error functions as much and you can step through await calls exactly as if they were normal synchronous calls. That sounds fantastic.

This is how the code will look like using Selenium WebDriver JavaScript with Async Await. All you need to do is declare a function as 'Async' and then start writing the code with 'Await'. The 'Await', as we discussed earlier, will resolve the promise for. Have 'Await driver.get' and then 'Await driver.findElement'. Then, sequential actions as you want to perform where your function scenarios. Then, you call the function.

Let's look into Demo Time and we'll see how we get started with the Selenium.

How did we do?

☆☆☆☆☆