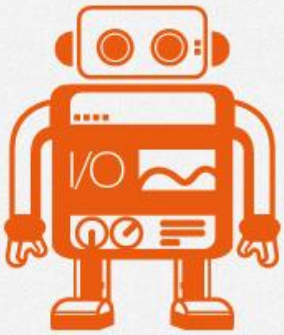


WEBDRIVER I/O

Webdriver IO

Student Handbook

Purpose: This guide is intended to be used alongside the lecture videos to remind you what we have covered. It is highly recommended you follow the course videos first and use this document as a quick reference guide to refamiliarize yourself on what has been covered to encourage your learning. **The videos will contain more detailed explanations.**

Salvatore Bruno & Gianni Bruno

Version 1.0 - Approved

QA Uni

Contents

MODULE 1 – INTRODUCTION 4

LECTURE 1 - COURSE INTRODUCTION 4
LECTURE 2 – EXAMPLE TEST EXECUTION USING WEBDRIVERIO & WEBDRIVERUNIVERSITY.COM 5
LECTURE 3 – HOW COURSE ATTACHMENT CODE EXAMPLES ARE STRUCTURED 7
LECTURE 4 – FREE 200+ PAGE E-BOOK DOWNLOAD - TO HELP YOU REVISE 7

MODULE 2 – SETUP & RESOURCES 8

LECTURE 5 - GITBASH SETUP WINDOWS 8
LECTURE 6 – ITERM2 SETUP MAC (MAC USERS ONLY) 9
LECTURE 7 – WEBDRIVER IO API 10
LECTURE 8 – NODEJS NPM PACKAGE JSON SETUP 11
LECTURE 9 – HANDLING DEPENDENCIES 14
LECTURE 10 – SELENIUM STANDALONE 17
LECTURE 11 – SUBLIME IDE SETUP 19

MODULE 3 – CREATING OUR FIRST TESTS 20

LECTURE 12 - CREATING OUR FIRST AUTOMATION TEST 20
LECTURE 13 - CREATING OUR SECOND AUTOMATION TEST 22

MODULE 4 – WDIO 24

LECTURE 14 - WDIO - CREATION & REVIEW 24
LECTURE 15 - WDIO - TRIGGERING TESTS 26
LECTURE 16 - WDIO - CONTACT US & LOGIN PORTAL TEST 28
LECTURE 17 - WDIO - EXECUTING OUR NEW & IMPROVED TESTS 30
LECTURE 18 - WDIO - REVIEWING THE WDIO FILE 31

MODULE 5 – MOCHA 33

LECTURE 19 - MOCHA – INTRODUCTION 33
LECTURE 20 - MOCHA – STRUCTURING & COMBINING TESTS PART 1 35
LECTURE 21 - MOCHA – STRUCTURING & COMBINING TESTS PART 2 37
LECTURE 22 - MOCHA – REVIEWING & EXECUTING OUR NEW & IMPROVED TESTS 38

MODULE 6 – WDIO SYNC MODE 40

LECTURE 23 - WDIO – SIMPLIFY TESTS AND SYNC MODE 40
LECTURE 24 - WDIO – CONFIGURING OUR TESTS TO USE SYNC MODE 42

MODULE 7 – SELENIUM STANDALONE & NPM SCRIPTS 44

LECTURE 25 - SELENIUM STANDALONE & NPM SCRIPTS 44

MODULE 8 – ENVIRONMENTS & BASE URL 46

LECTURE 26 - BASE URL SETUP 46
LECTURE 27 - HANDLING MULTIPLE ENVIRONMENTS DURING RUNTIME 48

MODULE 9 – LOGGING 50

LECTURE 28 - LOGGING DURING RUNTIME 50

MODULE 10 – NODE ASSERTIONS 52

LECTURE 29 - IMPLEMENTING NODE ASSERTIONS 52

MODULE 11 – CHAI 55

LECTURE 30 – CHAI – ADVANCED ASSERTIONS & CODE EXAMPLE – PART 1 55

Lecture 31 – Chai - Advanced Assertions & Code Example – Part 2 57
Lecture 32 – Chai - Adding Assertions to our Tests 59
Lecture 33 – Chai - Centralizing Assertions Using WDIO File 61

MODULE 12 – PAUSE, DEBUG MODE & SELECTORS 63

Lecture 34 - Pause Command Part 1 63
Lecture 35 - Pause Command Part 2 65
Lecture 36 - Debug Mode 66
Lecture 37 - Creating Selectors using Ranorex 69

MODULE 13 – TARGETING & SKIPPING SPECIFIC TESTS 71

Lecture 38 - Targeting Specific Tests 71
Lecture 39 - Skipping Specific Tests 73

MODULE 14 – CSS EXTRACTION 75

Lecture 40 - getCssProperty Command 75

MODULE 15 – MOCHA HOOKS 77

Lecture 41 - Mocha Hooks Part 1 77
Lecture 42 - Mocha Hooks Part 2 80

MODULE 16 – HANDLING BROWSER WINDOW TABS 83

Lecture 43 - Tabs Part 1 83
Lecture 44 - Tabs Part 2 87

MODULE 17 – VERIFY ELEMENTS 90

Lecture 45 - isExisting Part 1 90
Lecture 46 - isExisting Part 2 92
Lecture 47 - isVisible 94
Lecture 48 – hasFocus – Part 1 96
Lecture 49 – hasFocus – Part 2 99
Lecture 50 – isEnabled – Part 1 101
Lecture 51 – isEnabled – Part 2 103
Lecture 52 – isSelected – Part 1 105
Lecture 53 – isSelected – Part 2 107
Lecture 54 – isVisibleWithinViewport - Part 1 109
Lecture 55 – isVisibleWithinViewport - Part 2 113
Lecture 56 – getText, isVisible, isExisting - Part 1 116
Lecture 57 – getText, isVisible, isExisting - Part 2 121
Lecture 58 – getText, isVisible, isExisting - Part 3 125
Lecture 59 – waitForText Part 1 127
Lecture 60 – waitForText Part 2 131
Lecture 61 – waitForExist, waitForVisible Part 1 133
Lecture 62 – waitForExist, waitForVisible Part 2 136
Lecture 63 – waitUntil 137
Lecture 64 – waitForValue 139

MODULE 18 – USING EXTERNAL DATA (SYNC DATA MODE) 141

Lecture 65 - Using External Data (Sync Data Mode) - Part 1 141
Lecture 66 - Using External Data (Sync Data Mode) - Part 2 144

MODULE 19 – CUSTOM COMMANDS (ADDCOMMAND) 146

Lecture 67 - Custom Commands (addCommand) - Part 1 146

Lecture 68 - Custom Commands (addCommand) - Part 2 148

MODULE 20 – INJECTING JAVASCRIPT CODE (EXECUTE COMMAND) 150

Lecture 69 - Injecting JavaScript Code (Execute Command) - Part 1 150
Lecture 70 - Injecting JavaScript Code (Execute Command) - Part 2 153
Lecture 71 - Injecting JavaScript Code (Execute Command) - Part 3 156

MODULE 21 – PAGE OBJECT MODEL (POM) 157

Lecture 72 - Page Object Model (POM) – Intro 157
Lecture 73 - Page Object Model (POM) - Phase 1 - Part 1 160
Lecture 74 - Page Object Model (POM) - Phase 1 - Part 2 163
Lecture 75 - Page Object Model (POM) - Phase 1 - Part 3 166
Lecture 76 - Page Object Model (POM) - Phase 1 - Part 4 169
Lecture 77 - Page Object Model (POM) - Phase 1 - Part 5 171
Lecture 78 - Page Object Model (POM) - Phase 2 - Part 1 173
Lecture 79 - Page Object Model (POM) - Phase 2 - Part 2 174
Lecture 80 - Page Object Model (POM) - Phase 2 - Part 3 176
Lecture 81 - Page Object Model (POM) - Phase 2 - Part 4 178
Lecture 82 - Page Object Model (POM) - Phase 2 - Part 5 180
Lecture 83 - Page Object Model (POM) - Phase 2 - Part 6 183
Lecture 84 - Page Object Model (POM) - Phase 3 - Part 1 184
Lecture 85 - Page Object Model (POM) - Phase 3 - Part 2 185
Lecture 86 - Page Object Model (POM) - Phase 3 - Part 3 187
Lecture 87 - Page Object Model (POM) - Phase 3 - Part 4 189
Lecture 88 - Page Object Model (POM) - Phase 3 - Part 5 192
Lecture 89 - Page Object Model (POM) - Phase 3 - Part 6 194

MODULE 22 – ADVANCED REPORTING 195

Lecture 90 - Advanced Reporting – Intro 195
Lecture 91 - Advanced Reporting - Junit Reports 199
Lecture 92 - Advanced Reporting - JSON Reports 201
Lecture 93 - Advanced Reporting - Allure Reports 203
Lecture 94 - Advanced Reporting - Allure Reports - Attaching Images - Part 1 206
Lecture 95 - Advanced Reporting - Allure Reports - Attaching Images - Part 2 210
Lecture 96 - Free Up Space - Deleting Files and Folders 211
Lecture 97 - Jenkins Introduction and Explanation 214
Lecture 98 - Jenkins Installation and Setup 215
Lecture 99 - Jenkins Installing Plugins and Setting Up NodeJS 219
Lecture 100 - Trigger our Tests via Jenkins 222
Lecture 101 - Jenkins Adding Parameters 228
Lecture 102 - Jenkins Adding Additional Logging Information 234
Lecture 103 - Jenkins Execute Our Tests when Ever We Want 236
Lecture 104 - Generating Allure Reports within Jenkins – Part 1 237
Lecture 105 - Generating Allure Reports within Jenkins – Part 2 237
Lecture 106 - Generating Allure Reports within Jenkins – Part 3 239
Lecture 107 - Generating Allure Reports within Jenkins – Part 4 242
Lecture 108 - Generating Allure Reports within Jenkins – Part 5 244

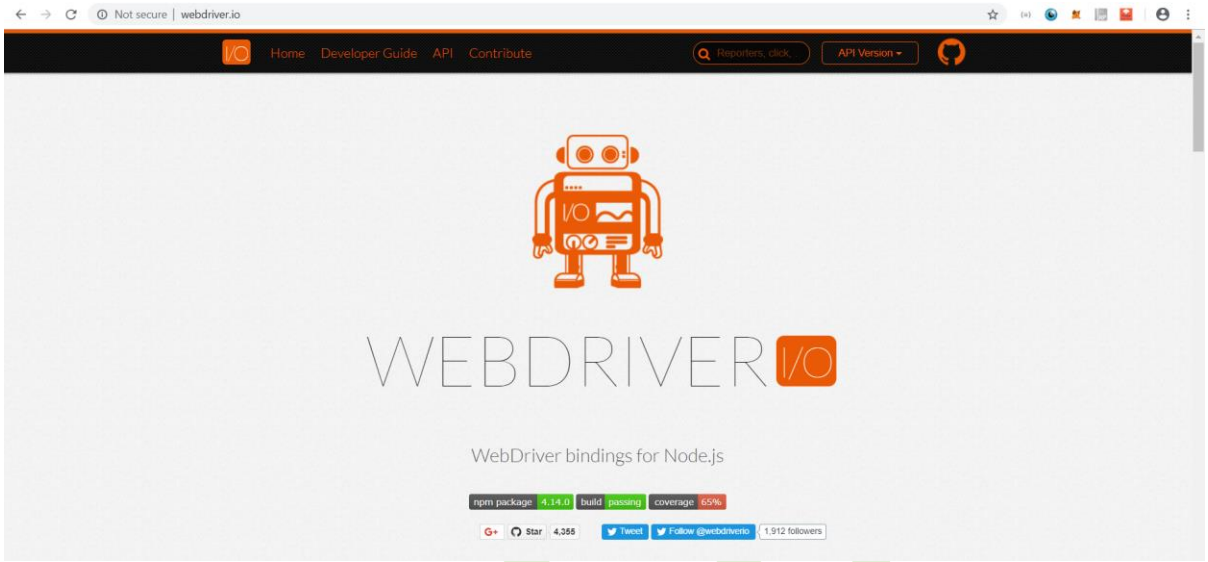
Module 1 – Introduction

Module 1 introduces you to Webdriver IO.

Lecture 1 - Course Introduction

Lecture Resources on Udemy:

- **None**



What is WebdriverIO?

WebdriverIO lets you control a browser or a mobile application with just a few lines of code. Your test code will look simple, concise and easy to read. The integrated test runner let you write asynchronous commands in a synchronous way so that you don't need to care about how to handle a Promise to avoid racing conditions. Additionally, it takes away all the cumbersome setup work and manages the Selenium session for you.

Working with elements on a page has never been easier due to its synchronous nature. When fetching or looping over elements you can use just native JavaScript functions. With the $ and $$functions.

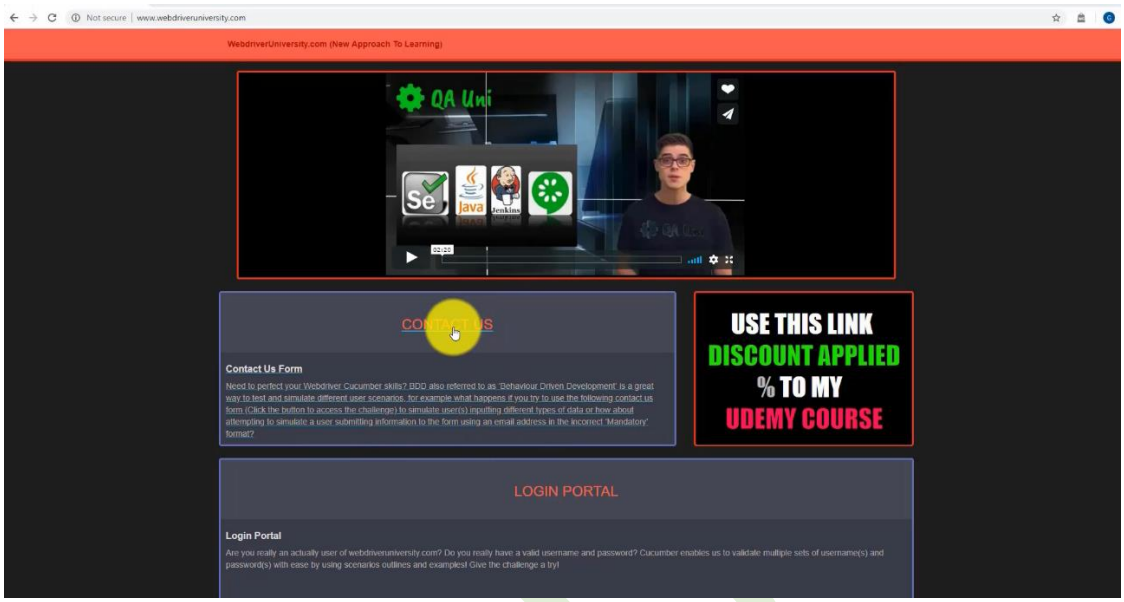
WebdriverIO provides useful shortcuts which can also be chained in order to move deeper in the DOM tree without using complex xPath selectors.

The test runner also comes with a variety of hooks that allow you to interrupt the test process in order to e.g. take screenshots if an error occurs or modify the test procedure according to a previous test result. This is used by WebdriverIOs services to integrate your tests with 3rd party tools like Appium.

Lecture 2 – Example Test Execution using webdriverIO & webdriveruniversity.com

Lecture Resources on Udemy:

- www.webdriveruniversity.com



In this lecture, you are introduced to my website www.webdriveruniversity.com that acts as a testing arena that allows you to run tests against a number of scenarios often found on websites. Some examples of these scenarios are:

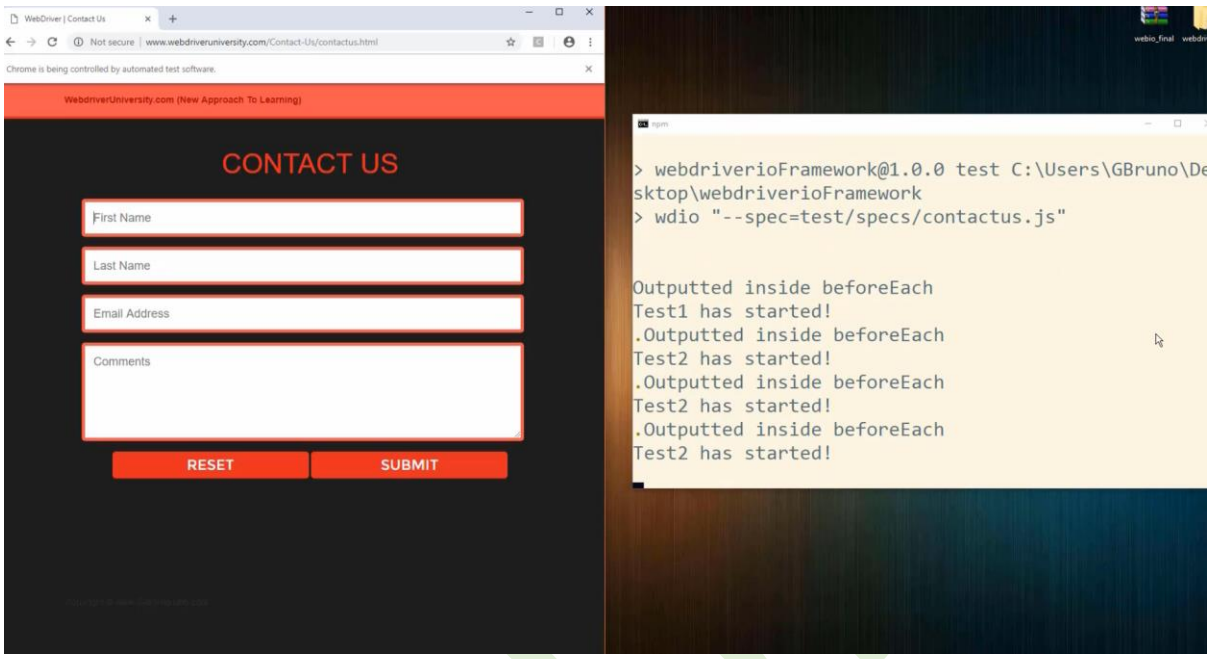
- Contact Us Forms
- Drop down lists
- Ajax Loader
- Button Clicks
- Etc.

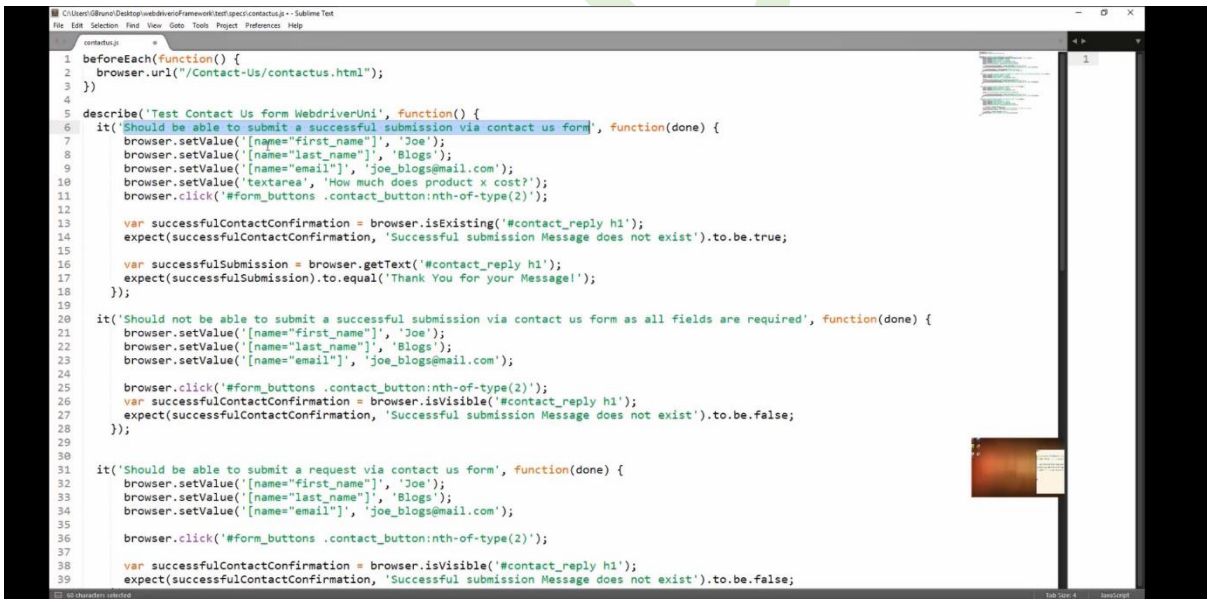
Some of the technologies you'll be shown during this course are:

- Selenium
- Webdriver IO
- Mocha
- Chai
- HTML5
- NodeJS

Continued Next Page

Finally, you are given a demonstration of a test that uses WebdriverIO that goes to a website (www.webdriveruniversity.com in this case) and opens a particular webpage (http://www.webdriveruniversity.com/Contact-Us/contactus.html) using the Chrome browser.





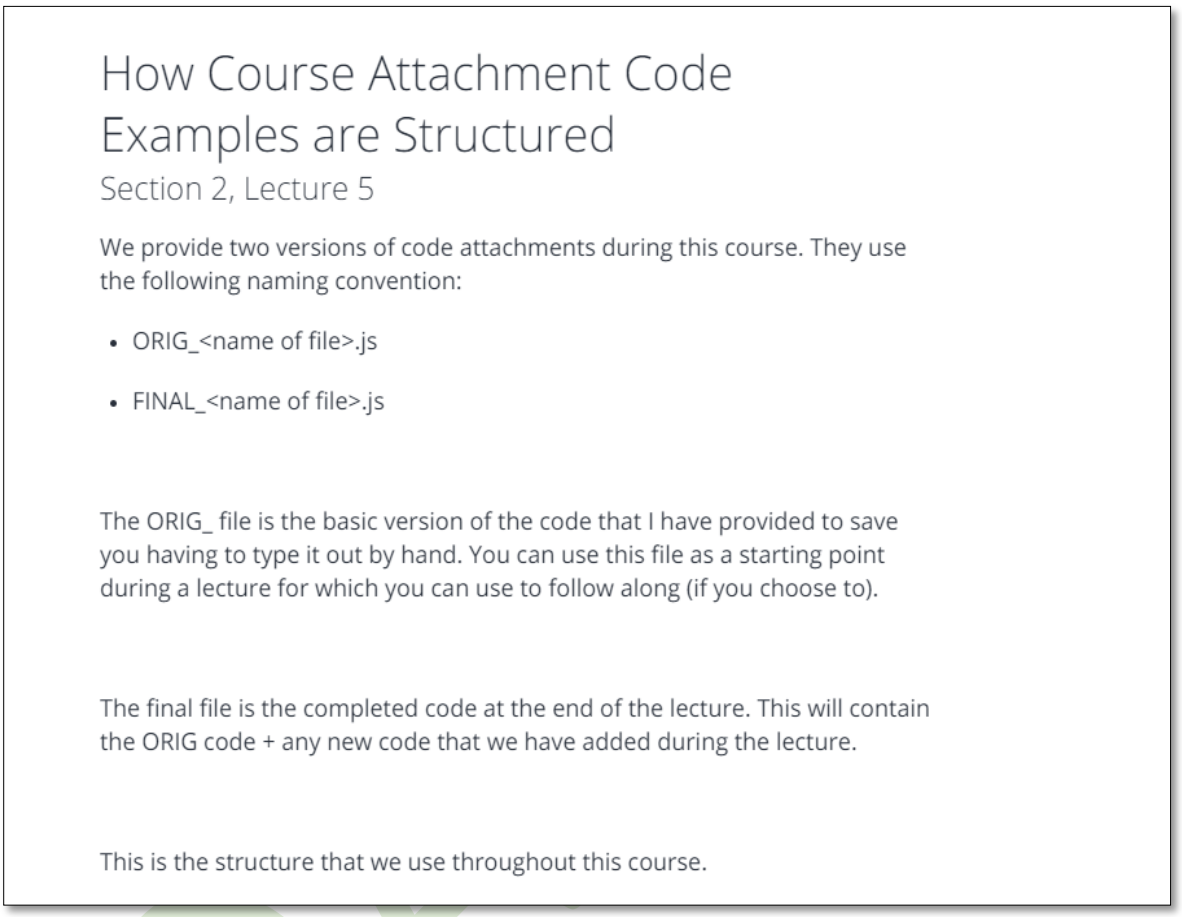
Key points:

- The test is lightening quick (far quicker than traditional Java based tests)
- 4 tests were completed and reported to the command window
- The total testing time for the four tests was around 15 seconds!
- Not only are the tests interacting with the target website, but a number of assertions are operating behind the scenes (positive and negative) to validate if the test scenarios were successful or unsuccessful

Lecture 3 – How Course Attachment Code Examples are Structured

Lecture Resources on Udemy:

- None



Lecture 4 – Free 200+ Page E-book Download - To help you revise

Lecture Resources on Udemy:

- None

As you're already reading the e-book, this lecture is listed here just to keep the lecture number ordering aligned to that on Udemy ☺.

Module 2 – Setup & Resources

Module 2 explains the tools that we will be using during this course and how to set them up.

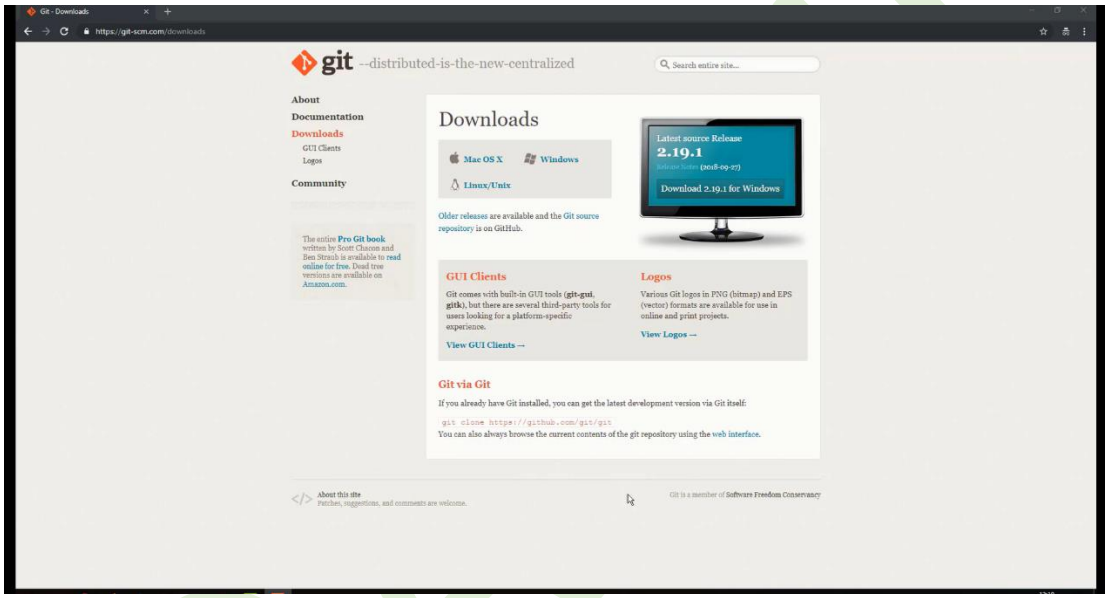
Lecture 5 - Gitbash Setup Windows

Lecture Resources on Udemy:

- https://git-scm.com/

What is Git Bash?

Git Bash for Windows is a package that is comprised of two parts: git which is a version control system (VCS) which tracks the file changes, commonly used for programming in a team setting. bash is a Linux-based command line (which has been ported over to Windows).



Instructions:

1. Find the download page and download the correct program based on your windows version (X86 or X64).
2. Bring the downloaded file to your desktop and then double click
3. Follow the installation instructions;
    a. Leave the default directory path on the first page
    b. Click the checkbox to “add additional icons” on the second page and click next
    c. When you come to the “Adjusting your PATH environment” section, change the default value to “Use Git from Git Bash only” and continue pressing next
    d. When you come to the “Configuring the terminal emulator to use Git Bash” section, select the value “Use Windows default console window” and click next
    e. On the next page also tick “enable symbolic links” and click next
    f. On the Configuring experimental options page, tick the two available options and hit Install
    g. Once the installation is complete, you will see a Git Bash icon on your desktop, select it to open the program

Lecture 6 – iTerm2 Setup Mac (MAC USERS ONLY)

Lecture Resources on Udemy:

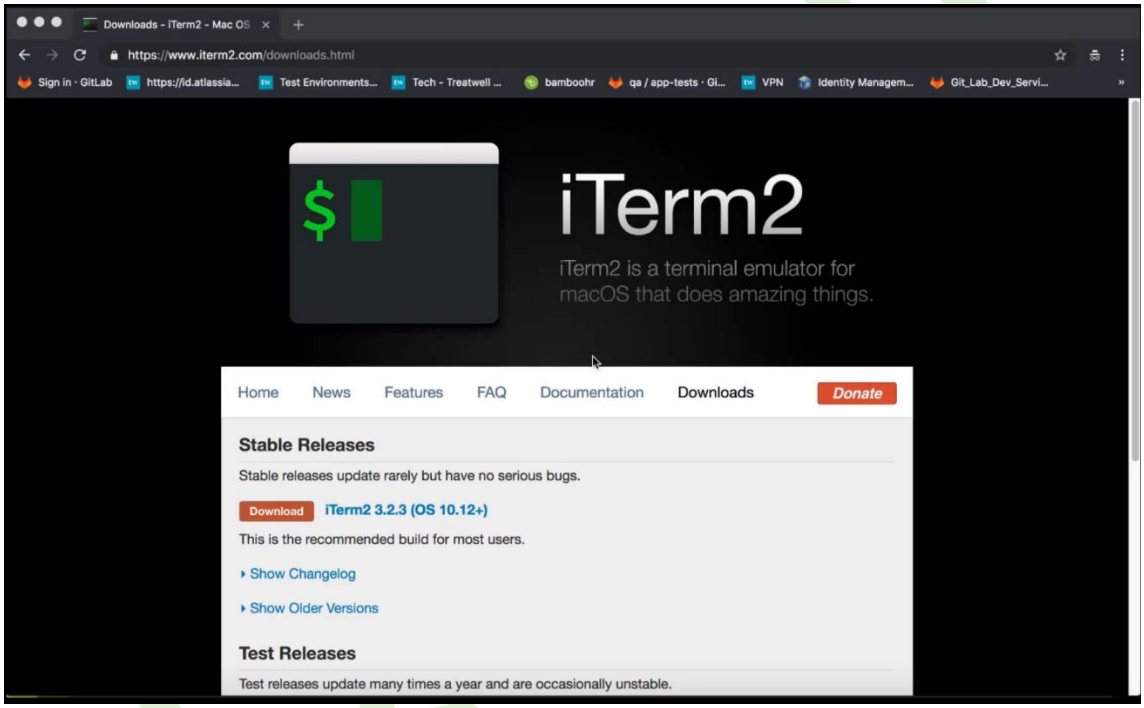
- https://iterm2.com/

What is iTerm2?

iTerm2 is a replacement for Terminal and the successor to iTerm. It works on Macs with macOS 10.10 or newer. iTerm2 brings the terminal into the modern age with features you never knew you always wanted. It's features rich and provides several additional features above the standard terminal which will make it easier to run our tests.

To see a full list of features that are included in iTerm2, use the following link:

- https://iterm2.com/features.html



Instructions:

1. Visit the following links:
    a. https://iterm2.com/downloads
2. Find the download button and press it
3. The file will get downloaded and once complete, drag the download .zip to your desktop
4. Extract the content of the .zip to your desktop
5. Double click on the iTerm2 program and you will be prompted with a message to "move application to the application folder" click 'Move to the application folder'.
6. The program should then open and be ready for you to use

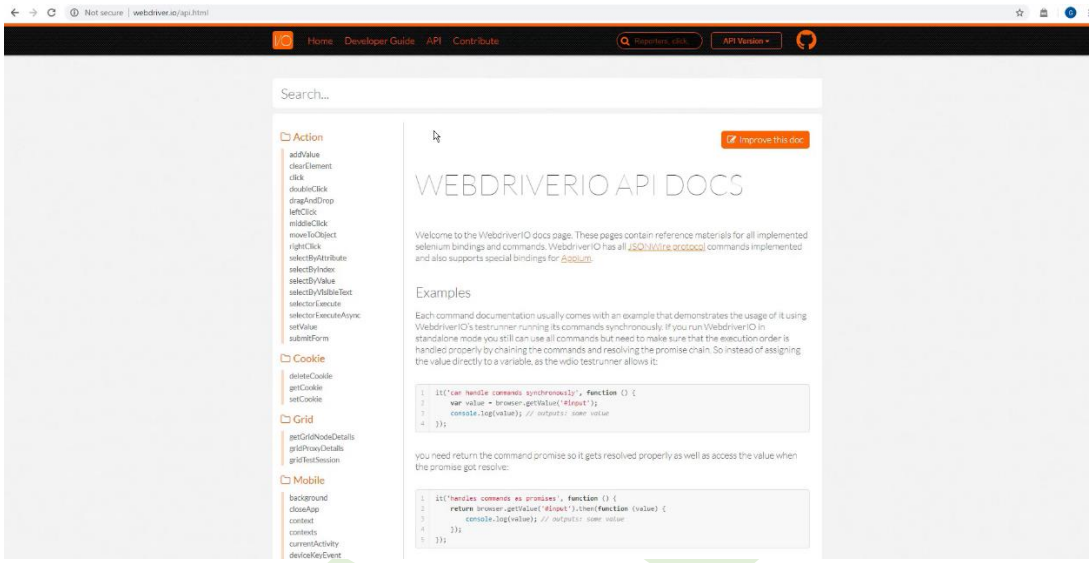
Lecture 7 – Webdriver IO API

Lecture Resources on Udemy:

- http://v4.webdriver.io/api.html

What is the Webdriver IO API?

The Webdriver IO API page contains reference materials for all implemented selenium bindings and commands. WebdriverIO has all JSONWire protocol commands implemented and also supports special bindings for Appium.



Key Points:

- Webdriver IO has its own API library that anyone can access by visiting www.webdriver.io/api.html
- The API library lists several commands that can be used when formulating our tests
- If you click on an example, say, Property.getTitle() the website will provide further information on what this command does and how it can be used
    - This example returns the title of the webpage shown in the browser during the test
    - The API page also provide a code example and what this does (briefly) is sets a browser URL that the test will visit, defines a variable called title and sets it a value using the browser.getTitle() method. The title of the webpage is then retrieved and written to the console log.

```
it('should get the title of the document', function () {
    browser.url('http://v4.webdriver.io');

    var title = browser.getTitle()
    console.log(title);
    // outputs the following:
    // "WebdriverIO - WebDriver bindings for Node.js"
});
```

If this seems a little confusing, don't panic! We go through this in much more detail as we progress.

Lecture 8 – NodeJS NPM Package JSON Setup

Lecture Resources on Udemy:

- www.npmjs.com
- www.nodejs.org/dist/latest-v8.x/
- https://docs.npmjs.com/files/package.json

What is NodeJS, NPM and JSON?

NodeJS is a JavaScript runtime environment which includes everything you need to execute a program written in JavaScript. It's built on Chrome's V8 JavaScript engine. Node.js' package ecosystem, npm, is the largest ecosystem of open source libraries in the world. This leads us onto;

npm which is the world's largest software registry. Open source developers from every continent use npm to share and borrow packages, and many organizations use npm to manage private development as well.

You can use npm to:

- Adapt packages of code for your apps or incorporate packages as they are.
- Download standalone tools you can use right away.
- Run packages without downloading.
- Share code with any npm user, anywhere.
- Restrict code to specific developers.
- Create Orgs (organizations) to coordinate package maintenance, coding, and developers.
- Form virtual teams by using Orgs.
- Manage multiple versions of code and code dependencies.
- Update applications easily when underlying code is updated.
- Discover multiple ways to solve the same puzzle.
- Find other developers who are working on similar problems and projects.

JSON provides the ability to exchange data between a browser and a server, the data can only be text. JSON is text, and we can convert any JavaScript object into JSON, and send JSON to the server. We can also convert any JSON received from the server into JavaScript objects. This way we can work with the data as JavaScript objects, with no complicated parsing and translations.

Instructions:

In this lecture we installed nodeJS and npm by:

- Going to the website (refer to the lecture resources above)
- Select "install npm' and click the Download Node.js and npm button for your operating system
- At the time of recording these lectures, the latest version of Node.js and npm were causing issues with some package dependencies (which I assume is due to the packages needing to be updated).
    - **If you come across the following error** " Error: fibers@2.0.2 install: 'node build.js || nodejs build.js' " then I recommend you downgrade to node version v8.12.0

- To find this version, use the following url: www.nodejs.org/dist/latest-v8.x/
- Select the download that aligns to your operating system and when the download is complete, move it over to your desktop and double click it to start the installation.
- Follow the wizard instructions by keeping the default settings and continuing until the software starts to install

- Once the installation has completed, open GitBash (or iTerm2 for mac users) and type the following command: 'node -v'
  - You should see a version return which confirms the install was successful and that node is running in the background:

```
GBruno@Gi MINGW64 ~
$ node -v
v8.12.0
```

Instructions:

An effective way of handling dependencies linked to our framework:

- When you install a npm package using npm install <package-name>, you are installing it as a dependency
- The package is automatically listed in the package.json file, under the dependencies list
- The package.json file lists what packages we require for our project to work, for example we will be using a package called Selenium Standalone, so we install Selenium Standalaone using GitBash/ iTerm2 and ensure the package is recorded so that when we deploy our project elsewhere or share it with someone, then they will know what dependencies are present in order to run our project.
- On your desktop, create a new folder and name it 'wedriverioFramework'. This will be the main folder that we will use going forward on this course.
- Open up GitBash or iTerm2 and move to the directory of the folder
  - Use the 'cd' command which stands for change directory.
  - So, in my case, I type the following:

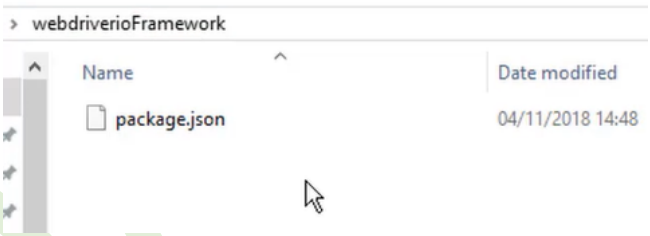
```
GBruno@Gi MINGW64 ~
$ cd Desktop

GBruno@Gi MINGW64 ~/Desktop
$ cd webdriverioFramework

GBruno@Gi MINGW64 ~/Desktop/webdriverioFramework
$
```

  - If you type the 'ls' command, it will list files in the current directory. At the moment we do not have any files in the current wedriverioFramework folder, so nothing will be returned on the command line
- Now we need to create the package.json file.
  - To do this, type 'npm init' and press enter
  - This will prompt a wizard like function where we begin creating our package file, where the command line will ask you a series of step-by-step questions
    - The first question will be to provide the **package name** (leave this as default) and press enter
    - **Version number** – leave this as the default value and press enter

- **Description** – This can be whatever you want it to be, but in my case I wrote "Webdriverio Framework" and pressed enter
- **Entry Point** – By default this is set as index.js (use the default value and press enter). The entry point is where our project will start when it runs (e.g. the code will initiate index.js on the first step).
- **Test Command** – Leave this as blank and press enter
- **Git Repository** – we are not going to be using one, so leave this as blank and press enter
- **Keywords** – leave this as default and press enter (You would use keywords that are relevant to your module and that you would expect people to use while searching for a module like yours)
- **Author** – enter your name and press enter
- **License** – leave this as default and press enter (you can use this field to enter a license type e.g. you do not want your project to be used for commercial uses, for example).
- Finally, you will be asked to review your entries and to confirm they are correct. Type 'yes' and press enter.
- The package.json file will have now been created in the background and you will be brought to a new line in GitBash/ iTerm2 as shown:

```
Is this OK? (yes) yes

GBruno@Gi MINGW64 ~/Desktop/webdriverioFramework
$
```

- Now, go back to your 'webdriverioFramework' folder on your desktop and look inside and you should see a package.json file:



- And if you open it using a text editor (sublime text or notepad++) you should see something like:

```
webdrivериoframework",
: "1.0.0",
ion": "Webdriverio Framework",
index.js",
: {
 "echo \"Error: no test specified\" && exit 1"

 "Gianni Bruno",
: "ISC"
```

Our dependencies will be added to this file going forward. To read more about the npm init setup (e.g. to get a deeper understanding of what Test Command or Git Repository can be used for), then refer to the following website:

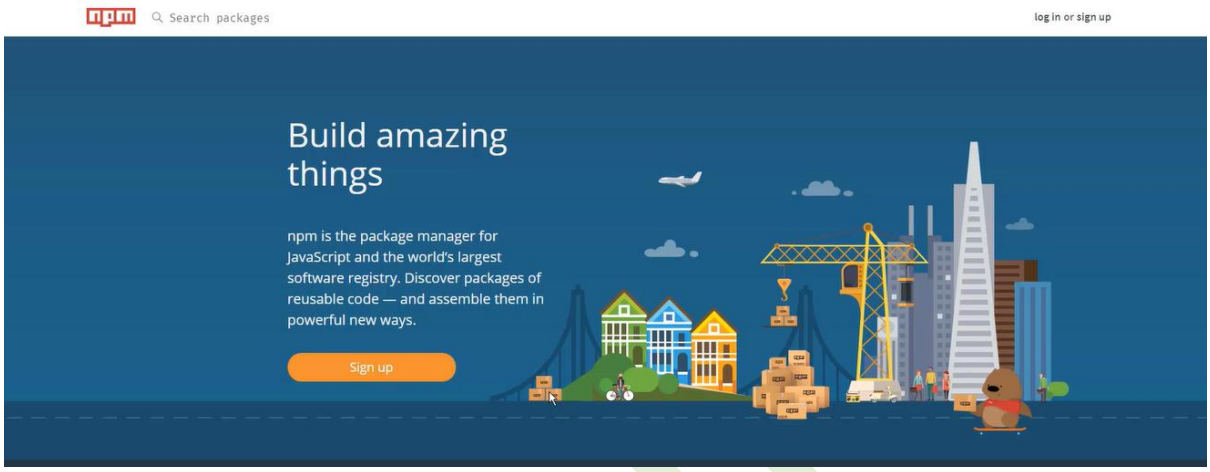
https://docs.npmjs.com/files/package.json

Lecture 9 – Handling Dependencies

Lecture Resources on Udemy:

- https://www.npmjs.com

What are Dependencies and how can we Handle them?

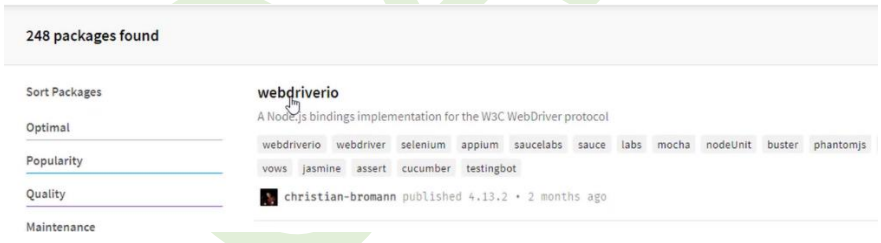


As mentioned, npm is a package manager for Javascript and it the worlds largest software registry. The registry consists of code that can be reused in your project (rather than having to rewrite it yourself!).

Use the following link to visit the npm website:

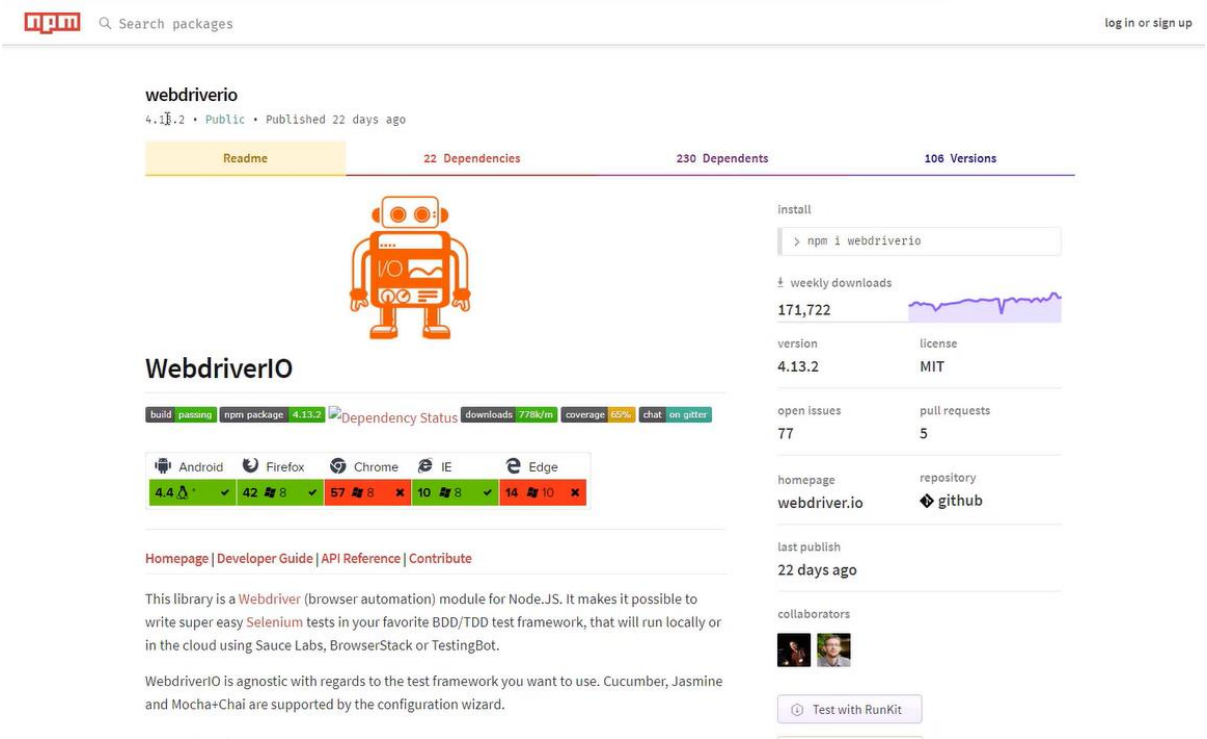
- https://www.npmjs.com

In the course video, we search for the phrase "webdriverio" and you will be returned search results that match your search term:



Continued next page

If we click the webdriverio result, we will be taken to the webdriverio package page:



You will see a short description, explaining the purpose of this package and also, take note of the dependencies tab:

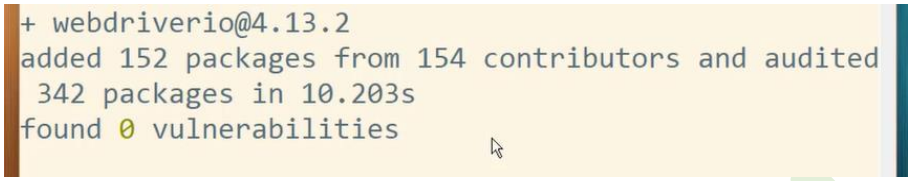
Dependencies are the other packages needed in order to use the webdriverio package

This means, if we were to install the webdriverio package as part of our project, it would also install all the other dependencies that webdriverio would need for it to work.

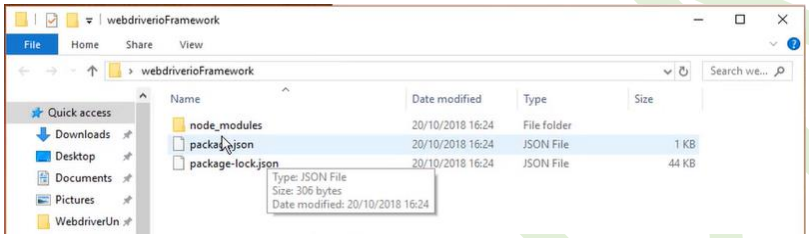
Instructions:

- Go back to GitBash/ iTerm2 and ensure you're in the webdriverioFramework directory
- If you write the command 'ls' it should show the file 'package.json' in your directory
- We are now going to install a package by using the following command;
    - npm install -save -dev webdriverio@4.13.2
        - **npm install** tells node.js that we are wanting to install a package
        - -**save** saves the package to our package.json file, meaning it would know what packages are required in order to use our project
        - **-dev** Those packages required in order to help develop your module are listed under the "devDependencies" property. These packages are not necessary for others to use the module, but if they want to help develop the module, these packages will be needed.
        - **webdriverio** is the package name we want to install
        - **@4.13.2** is used version number of webdriverio used during this course.
- Once we enter the command and press enter, the package and its dependencies will start installing:

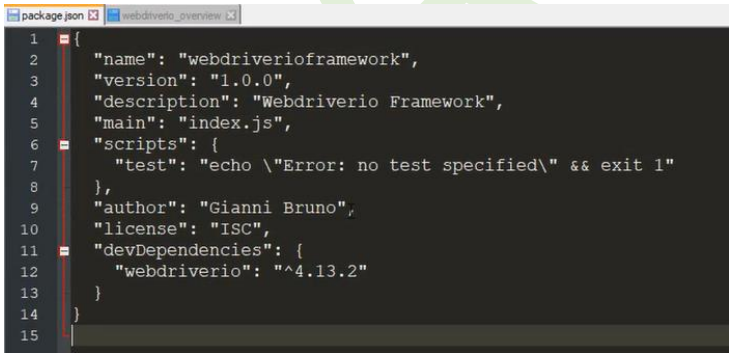
- Once the installation is complete, you will be shown a summary of the packages installed:



- These figures will include the dependencies needed in order to use webdriverio (the packages webdriverio needs in order to use it)
- If you go back to your webdriverioFramwork folder, you will see additional folders have been created, as shown:



- Open the node_modules folder and you will see all the modules (dependencies) needed that came as part of the package to use webdriverio
- Next, open up the package.json file and you will see that the webdriverio dependency has been added to it:



We go through the exact same process to install the 'mocha' package. Please refer to the lecture videos for a detailed explanation if required.

Lecture 10 – Selenium Standalone

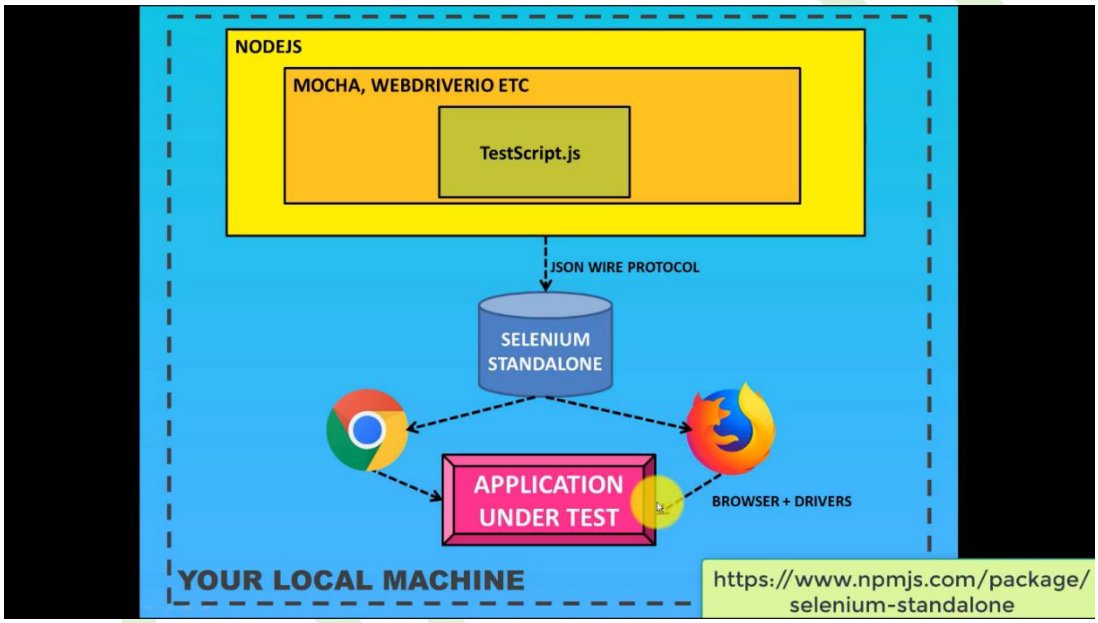
Lecture Resources on Udemy:

- www.npmjs.com/package/selenium-standalone

What is Selenium Standalone?

It's a node based CLI library for launching Selenium with WebDrivers support. Basically, it's a way for us to use Selenium which is needed to run our automation tests. As explained in the NodeJS/ NPM lecture, we can instruct our project to use dependencies and in this case we are calling the selenium-standalone library.

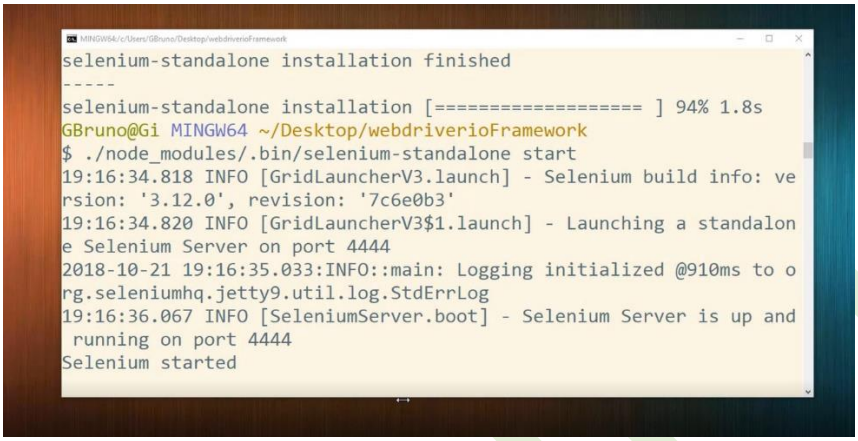
Selenium Standalone acts as a middle layer between our automation framework and our browsers. It passes on test messages to the relevant browser that will then run our test against the application that's in scope.



Instructions:

1. Open up the Gitbash or iTerm2
2. Navigate to our project root directory (mines: ~/Desktop/webdriverioFramework)
3. If you navigate to the folder directory manually, you should see there are two files already present (based on what we've done in previous lectures). These should be named:
    a. package.json
    b. package-lock.json
        i. If you open the package.json file, you will see there are currently no dependencies listed (under devDependencies)
4. Use the following command:
    a. Npm install --save --dev webdriverio@4.13.2 selenium-standalone@latest
    b. And press Enter
    c. You'll see the packages start installing
5. We can confirm the dependencies were successfully added by using the following command:

    a. ls ./node_modules/.bin/
    b. This should return the files within your project bin folder and you should see selenium standalone listed
6. We now need to install selenium-standalone. To do this run the following command (in the .bin directory):
    a. selenium-standalone install
7. Confirm the installation was successful by running the following command:
    a. selenium-standalone start



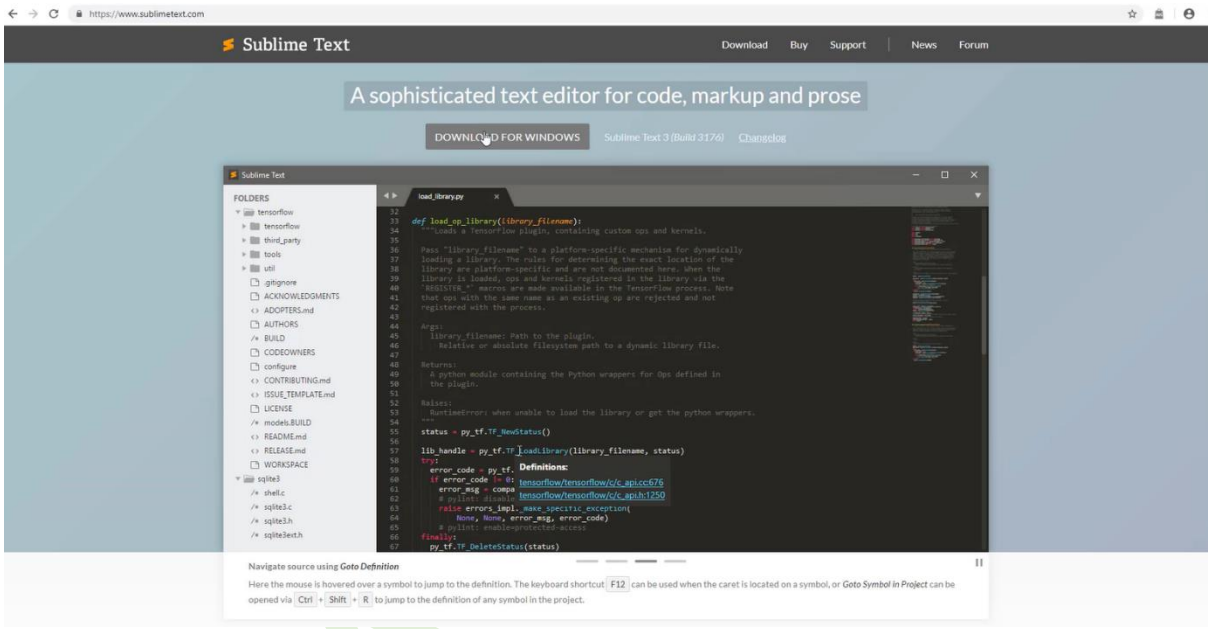
Lecture 11 – Sublime IDE Setup

Lecture Resources on Udemy:

- https://www.sublimetext.com/

What is Sublime IDE and how do we use it?

Sublime Text is a proprietary cross-platform source code editor with a Python application programming interface (API). It natively supports many programming languages and markup languages, and functions can be added by users with plugins, typically community-built and maintained under free-software licenses.



Instructions:

How to install Sublime Text

- Go to https://www.sublimetext.com/
- Click the download button for your operating system and let the application download
- Follow the installation instructions
- Open the application and;
    a. Add some code into the text editor (from the webdriver.io homepage)
    b. If keywords are not highlighted do the following;
        i. Go to 'View' > 'Syntax' > Select Javascript
    c. If you want to change the colour scheme, then do the following;
        i. Go to 'Preferences' > 'Colour Scheme' and then choose one that you like

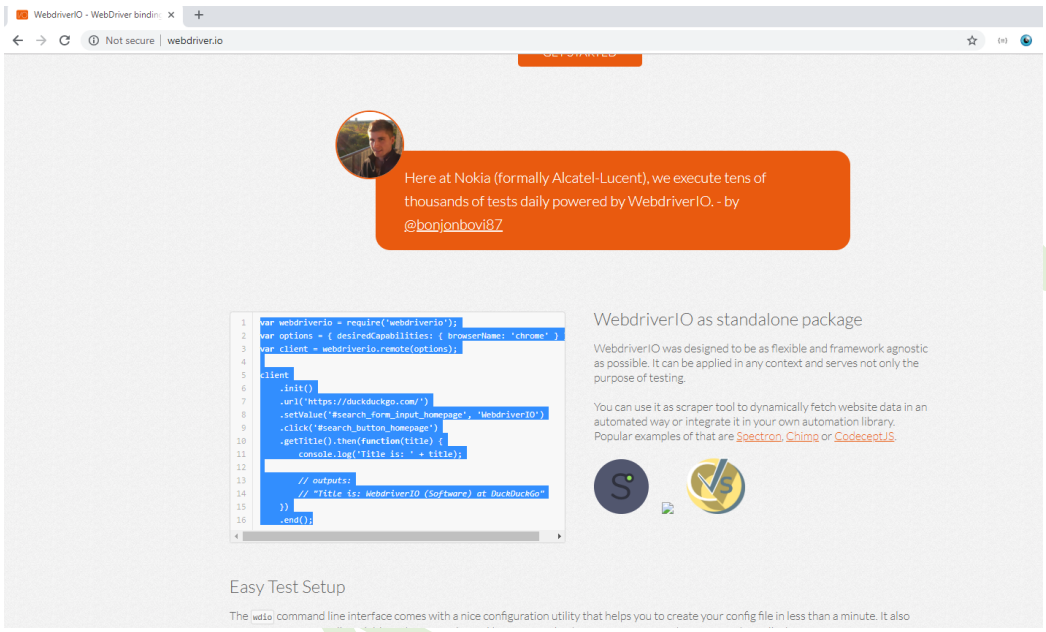
Module 3 – Creating Our First Tests

Module 3 will demonstrate how we can make our first two tests using Selenium and WebdriverIO. These tests are simple but are for demonstration purposes, so you get familiar on how tests can be developed.

Lecture 12 - Creating Our First Automation Test

Lecture Resources on Udemy:

- http://v4.webdriver.io/

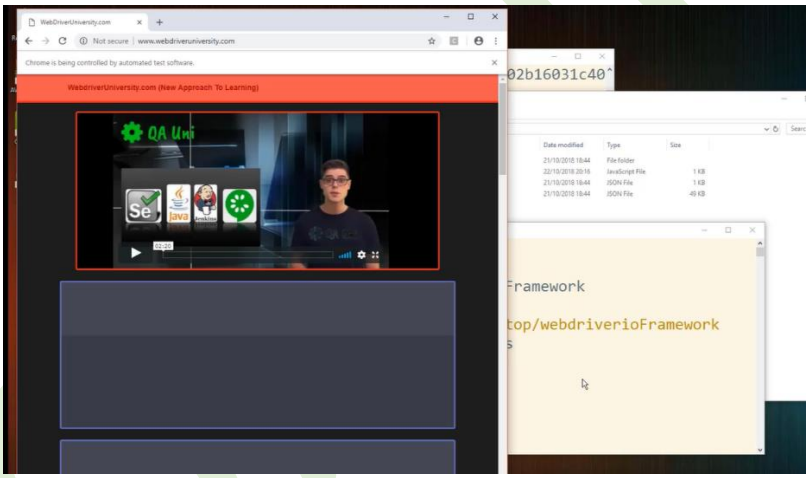


In this lecture we use one of the available demonstration tests found on the webdriver.io homepage.

Instructions:

1. We first go to the webdriver.io homepage (use the link above)
2. Around halfway down we can locate the demo test code, which we highlight and copy to our clipboard
3. Open sublime text
4. Paste the code into the text editor (if your code keywords aren't highlighted, Go to 'View' > 'Syntax' > Select Javascript
5. Change the URL to use www.webdriveruniversity.com
6. Change the .click parameter to use the login portal ID so that it reads:
    a. .click('#login-portal')
    b. Also remove the comment (highlighted in grey in the code) as this is not required
7. Now we need to save the file, so do the following:
    a. Select File > Save As > then go into our framework directory
    b. In my case this is located on my desktop and then 'webdriverioFramework' folder
    c. Call the file 'loginPortalTest.js'
        i. .js indicates this is a Javascript file
    d. The file should then be created and visible to you
8. Open up GitBash or iTerm
9. Move the command line into the project directory

a. use the command ‘cd’ (change directory), in my case this is;
b.
```
$ cd Desktop/webdrivierioFramework
```
10. Now we need to start up the Selenium Server by using the following command;
    a. ‘./node_modules/.bin/selenium-standalone start’ and press enter
    b. You should see something like;
    c.
```
MINGW64:/c/Users/GBruno/Desktop/webdrivierioFramework
 build info: version: '3.12.0', revision: '7c6e0b3'
20:17:47.025 INFO [GridLauncherV3$1.launch] - Launch
ing a standalone Selenium Server on port 4444
2018-10-22 20:17:47.197:INFO::main: Logging initiali
zed @646ms to org.seleniumhq.jetty9.util.log.StdErrL
og
20:17:48.314 INFO [SeleniumServer.boot] - Selenium S
erver is up and running on port 4444
Selenium started
```
11. Open another GitBash/ iTerm terminal window
12. Change directory to the same directory as before
13. To execute the test, you need to use the following command;
    a. node loginPortalTest.js and press enter



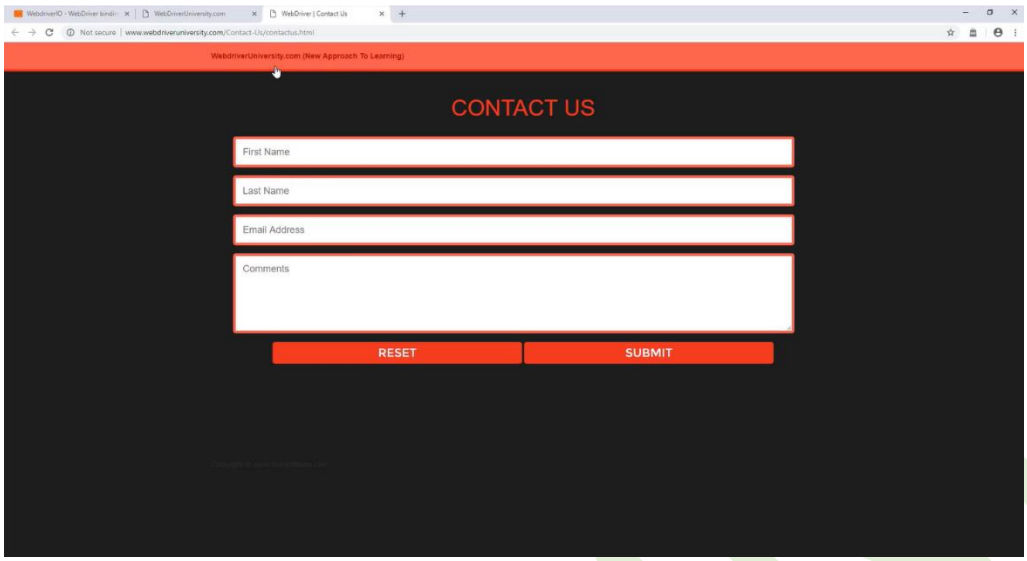
You should see the browser open and the title of the webpage is returned in the GitBash/ iTerm terminal window, displaying the title such as: Title is: WebDriverUniversity.com and then the browser closes.

Note how fast that test ran!

Lecture 13 - Creating Our Second Automation Test

Lecture Resources on Udemy:

- http://www.webdriveruniversity.com



In this lecture, we look at creating and executing a second test that uses the contact us form on webdriveruniversity.com. Contact forms on websites can be quite tricky because there is usually validation in place before it can be submitted (e.g. you need to fill in the mandatory fields before you can submit the request). We can formulate tests to ensure we test all scenarios but, in this case, we want to keep things simple to demonstrate how little code is needed to direct our tests to the contact us form.

Instructions:

1. Open our project directory folder
2. Copy and paste the existing loginPortal.js test and rename the new file 'contactUsTest.js'
3. Open the file using sublime text
    a. The first three lines of code initialises our driver instance and sets the browser to use as Chrome
    b. We keep the url code the same as we are using webdriveruniversity.com again
    c. We also want to show the page title in the console
4. Remove line 8 (the click button command) because this time we want to;
    a. Access webdriveruniveristy website
    b. Output the title of the homepage
    c. Get the test to click the contact us page link
5. We need to write a command that instructs our test to do this. We do this by;
    a. Adding the following line (on the line above the .end(); line
        i. .click("#contact-us")
            1. This uses a locator (id) that can be found using the google chrome inspector when highlighting over the contact us section on the homepage. This is what distinguishes this section of the homepage form other elements

b. We then add a pause of 3 seconds using the following;
    i. .pause(3000)
    ii. This forces our test to halt for 3 seconds after the contact us page has been clicked (so that we can see it – since it's so fast)

6. We then add **setViewPortSize** code which is important because this does the following:
    a. This command changes the viewport size of the browser. When talking about browser size we have to differentiate between the actual window size of the browser application and the document/viewport size of the website. The window size will always be bigger since it includes the height of any menu or status bars. The command tries to resize the browser multiple times (max 5 times) because Webdriver only allows to change the window size and doesn't take the viewport into consideration. This is handled by WebdriverIO internally.
    b. We set a width and height using the following code:
        i.
```
.setViewportSize({
    width: 1200,
    height: 800
})
```
7. Once you have added all the code above (refer to the lecture for the full code), Save the file.
8. Ensure the selenium server is up and running (refer to the notes in the previous lecture if it's not running)
9. Go to your second instance of GitBash/ iTerm and run the following command:
    a. node contactUsTest and then press Enter

You will see that the test executes, and the browser opens the webdriveruniversity.com homepage and then opens the contactus page!

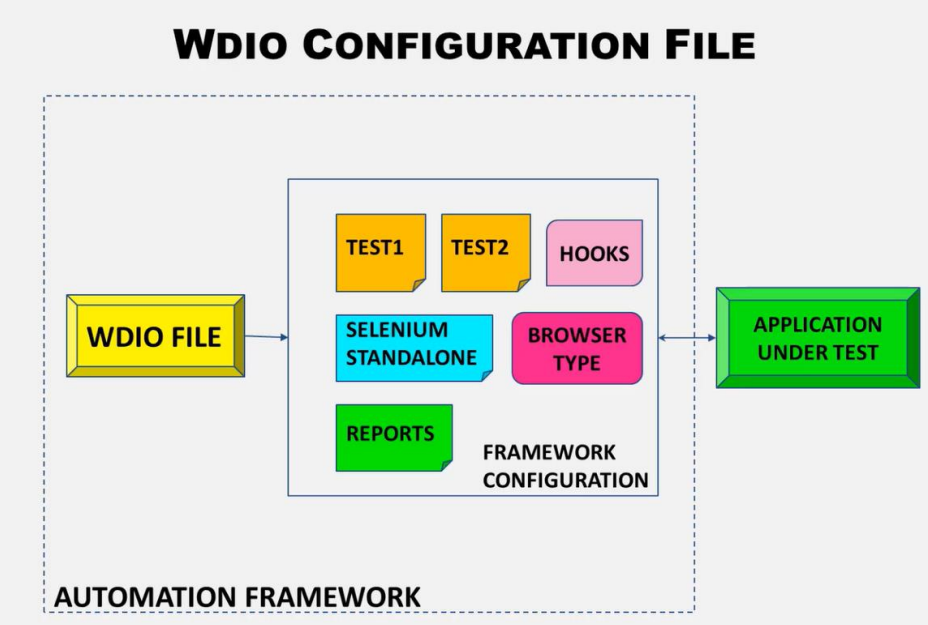
Module 4 – WDIO

This module covers WDIO (webdriver io test runner). WDIO allows us to simplify our tests by removing unnecessary, repeated code, as the WDIO file to take care of the configuration and initialization.

Lecture 14 - WDIO - Creation & Review

Lecture Resources on Udemy:

- http://v4.webdriver.io/guide/testrunner/gettingstarted.html



The WDIO Testrunner

The main purpose of WebdriverIO is end-to-end testing on a big scale. Webdriverio provides a test runner that helps you to build a reliable test suite that is easy to read and maintain. The test runner takes care of many problems you are usually facing when working with plain automation libraries. For one, it organizes your test runs and splits up test specs, so your tests can be executed with maximum concurrency. It also handles session management and provides a lot of features that help you to debug problems and find errors in your tests.

Key Points

- WDIO is the webdriverio Test Runner file
- We can use the WDIO to define settings that will be passed onto our tests
    - We can use it to target one or more tests
    - We can use hooks that are embedded in the WDIO file (e.g. display a message before a test)
    - We can use the WDIO file to start and stop the selenium server
    - We can use it to handle different types of browsers
    - We can use it to handle reports
    - And much more

Instructions:

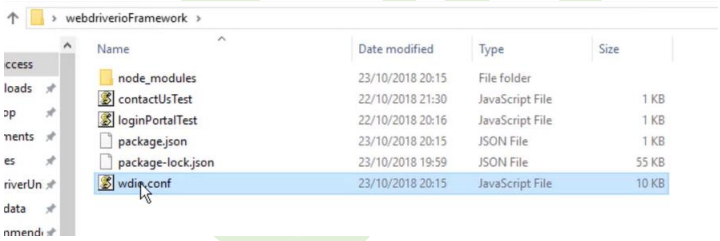
How to create the WDIO file

1. Open up GitBash or iTerm2
2. Make sure you are in the webdriverioFramework directory (use cd to change)
3. Type 'ls node_modules/.bin/' – this will show the files in the node_modules bin folder
4. You will see several modules but the one we want to use is WDIO
5. Type the following command './node_modules/ .bin/wdio'
    a. This will prompt the WDIO configuration helper
    b. You can select the options by using the up and down arrows on your keyboard and selecting enter
6. Select the following options:
    a. Step 1 – '**On my local machine'** + press enter
    b. Step 2 – '**Mocha'** + press enter
    c. Step 3 – '**Y'** (for framework adapter) + press enter
    d. Step 4 – Change default directory to '**./tests/**/*.js'** + press enter
    e. Step 5 – Select the default report type (**Dot**) + press enter
    f. Step 6 – Select '**Selenium-standalone**' + press enter
    g. Step 7 – Select '**Silent**' + press enter
    h. Step 8 – For the screenshot option, keep this as the **default** selection + press enter
    i. Step 9 – Change the base URL to '**www.webdriveruniversity.com**' + press enter

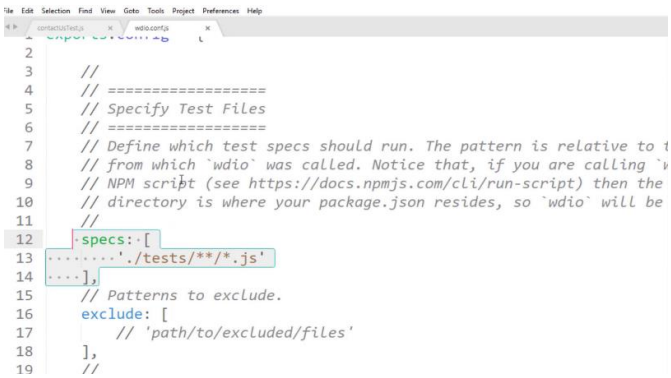
This should then complete the WDIO installation and you should see the following screenshot:

```
Installing wdio packages:

Packages installed successfully, creating configuration file...

Configuration file was created successfully!
To run your tests, execute:

$ wdio wdio.conf.js
```

To confirm this has successfully completed, you can navigate to our webdriverioFramework folder to view the files. You should see the wdio.conf file present:



If you open the file using SublimeText or Notepad++, you should see something like the following code:



Lecture 15 - WDIO - Triggering Tests

Lecture Resources on Udemy:

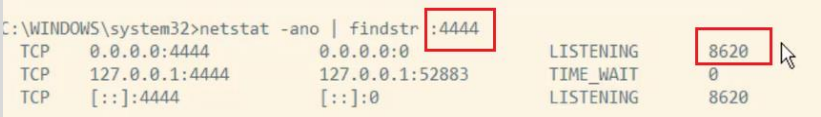
- None

In this lecture, we look at triggering tests using the WDIO file.

Tip – If windows users come across the following error when starting selenium

```
Selenium started
20:42:54.013 ERROR [SeleniumServer.boot] - Port 4444 is busy
, please choose a free port and specify it using -port optio
n
```

- Open cmd (command line)
- Type the following command:
  - **netstat –ano | findstr :4444** ← change the port number here based on what the error throws
  - This will show the PORTs that are in use e.g.



- Then type the following command:
  - **taskkill /PID 8620 /F** ← change 8620 to the listening port shown in your CMD window
  - **You will then be shown a message like:**

```
C:\WINDOWS\system32>taskkill /PID 8620 /F
SUCCESS: The process with PID 8620 has been terminated.
```

Reason: The reason we sometimes need to do this is because the port might be in use from a past session of selenium. We need to kill the process using the commands above.

Instructions:

We need to start the selenium server. We can do this by typing the following command in GitBash/ iTerm2:

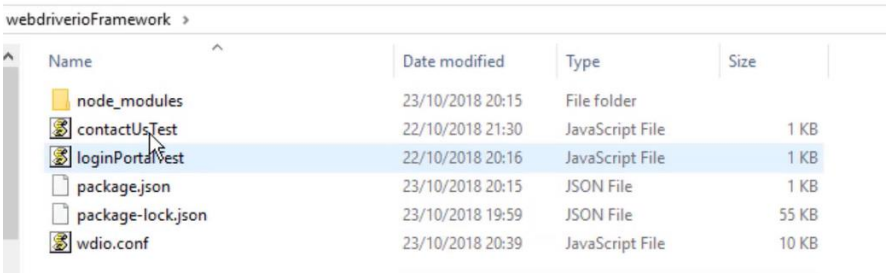
1. Make sure to be in the .bin directory:

```
GBruno@Gi MINGW64 ~/Desktop/webdriverioFramework
$ ./node_modules/.bin/
```

2.
3. Type '**selenium-standalone start**' which will start selenium:

```
Selenium started
```

4. If you look at the webdriverioFramework directory, you will see two tests are present from past lectures:

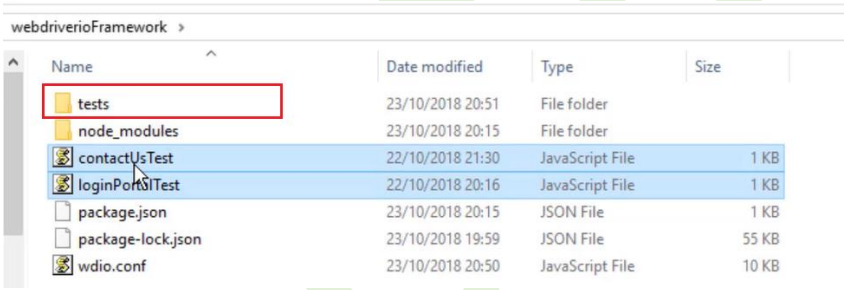


a. Open the wdio.conf file with your file editor
b. Scroll through the file contents and you will come across the following:

```
specs: [
    './tests/**/*.js'
],
```

   i. When setting up the WDIO file, we declared that our tests would reside in the 'tests' directory.
   ii. We need to create the folder and place our tests within it

c. Go back to the webdrivierioFramework folder and create a new folder called **tests**, as shown:



d. Then place our two tests 'contactUsTest' and 'loginPortalTest' into the tests folder

5. Open a second instance of GitBash / iTerm2 and move into the webdrivierioFramework directory
   a. Type in the following command:
      i. ./node_modules/ .bin/wdio
      ii. You should then see Chrome browser open

The tests will not execute at this point – so don't panic! The next lecture explains how we can execute the two tests.

Lecture 16 - WDIO - Contact Us & Login Portal Test

Lecture Resources on Udemy:

- None

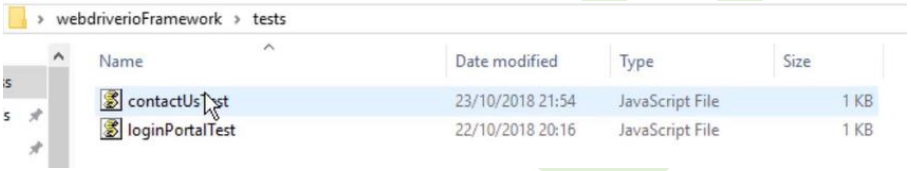
In this lecture, I start showing you how we can use the WDIO file to trigger our 'contactUsTest' and 'loginPortalTest' tests.

Key Points:

- We have already created our WDIO file
- We have placed our two tests into the tests folder that we recently created
- We have completed the setup steps when initialising WDIO
- We are now able to run the tests using WDIO that opens the Chrome browser
- We now need to config our tests so that WDIO can execute the 'contactUsTest' and 'loginPortalTest' tests

Instructions:

1. Go to the tests folder and open the contactUsTest file:



2. The current file looks like the following:

```
contactUsTest.js
1  var webdriverio = require('webdriverio');
2  var options = { desiredCapabilities: { browserName: 'chrome' } };
3  var client = webdriverio.remote(options);
4
5  client
6      .init()
7      .setViewportSize({
8          width: 1200,
9          height: 800
10     })
11     .url('http://www.webdriveruniversity.com/')
12     .getTitle().then(function(title) {
13         console.log('Title is: ' + title);
14     })
15     .click("#contact-us")
16     .pause(3000)
17     .end();
```

3. Since we now can take advantage of WDIO, it means that much of the code above is no longer necessary, meaning we can change the above code to:

4. Once you have amended the file, save it.
5. Then open our second test, loginPortalTest
6. The current code looks like this:

7. As mentioned, since we now can take advantage of WDIO, it means that much of the code above is no longer necessary, meaning we can change the above code to:

8. Once you have amended the file, save it.

For a more detailed explanation of the code amendments, please refer to the lecture video

Lecture 17 - WDIO - Executing Our New & Improved Tests

Lecture Resources on Udemy:

- None

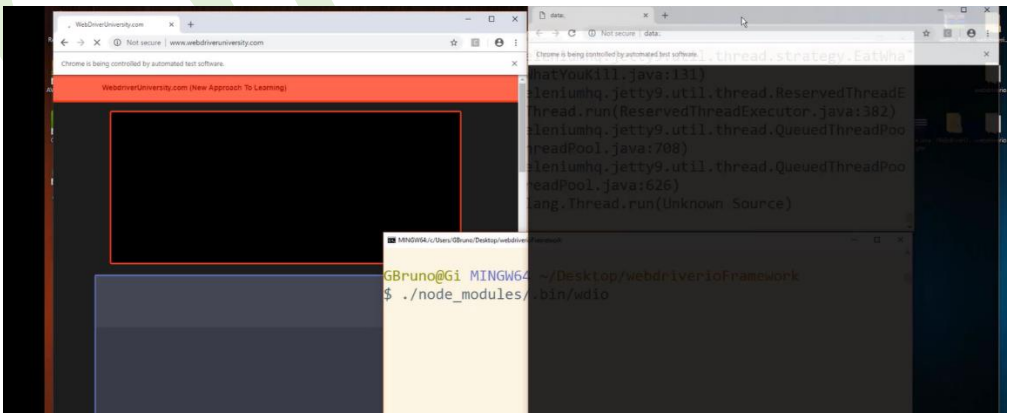
In this lecture, we continue to work towards getting out WDIO file to execute our 'contactUsTest' and 'loginPortalTest' tests.

Key Points:

- We have now amended our existing 'contactUsTest' and 'loginPortalTest' tests by removing unnecessary code
- This is because the WDIO file takes care of a lot of the initialization and setup, meaning it's no longer needs to be defined within the individual tests
- This also means that we no longer have to amend each and every test going forward (abiding to the DRY rule – **D**on't **R**epeat **Y**ourself which is good coding practise). It makes maintainability far easier going forward.

Instructions:

1. Before we move on, let's review our two tests side-by-side using Sublime
2. Open the 'contactUsTest' and 'loginPortalTest' tests
3. On Sublime, select View > Layout > Column: 2
4. Drag one of the tests to the second column
5. Take a minute to review our recent changes; this includes,
    a. Removing the initialization code from both of our test cases
    b. Both tests are now easier to read and contain less code
6. Close the tests and go back to GitBash / iTerm2 where two windows should be open
7. Ensure one session has the selenium server running (if it's not running, refer to lecture 13)
8. In the second GitBash / iTerm2 window, ensure you're in the webdriverioFramework directory
9. Now we trigger the WDIO file; using,
    a. **./node_modules/ .bin/wdio** + press enter
    b. You should see both tests flash up; as shown,



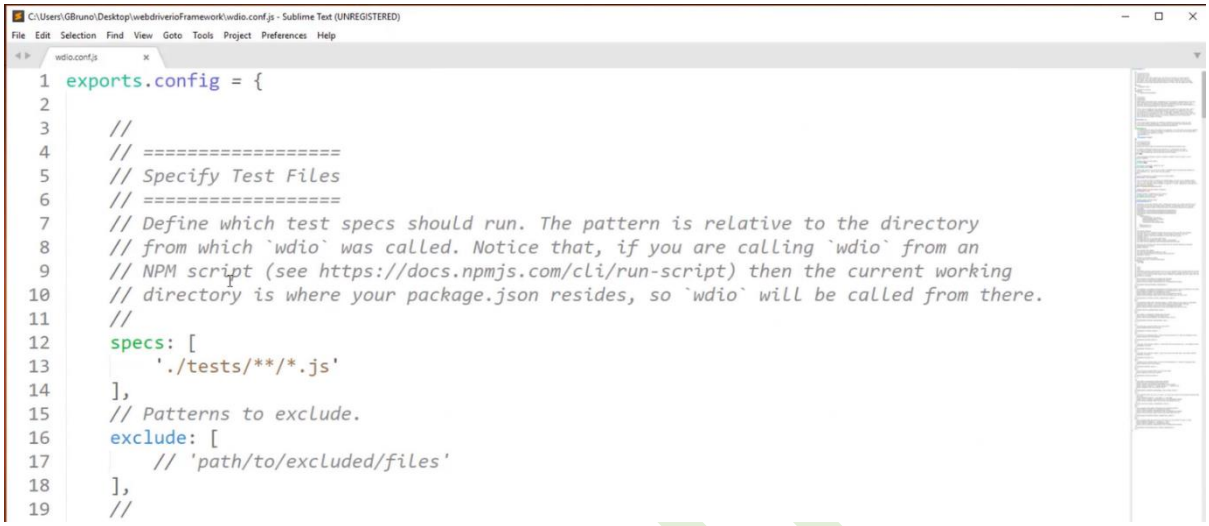
Take note of how quick the tests run. Webdriverio is very efficient and tests complete extremely quickly. Also notice how two browser windows opened, showing both test cases were executed.

Lecture 18 - WDIO - Reviewing the WDIO File

Lecture Resources on Udemy:

- http://v4.webdriver.io/guide/testrunner/configurationfile.html

In this lecture, we take a more detailed look at the WDIO file and review some of the key sections of code.



Instructions:

1. Navigate to the webdriverioFramework folder and open the WDIO file using your text editor (e.g. Sublime Text)
2. Scroll through the file and read some of the comments associated to the code blocks to gain an understanding what the code does
3. On **line 58** we currently have the following code:
    a. **Sync: true,**
    b. Change this to **Sync: false,** and save your file
    c. We will explain Sync mode in detail in upcoming lectures

Key Points:

- The WDIO file allows us to set a number of configuration settings that will help us improve our tests
- **Code lines 12-18** contains a reference to where our tests reside. It points to the location of where our test files can be located. You can also exclude tests that you do not want to run from the directory.
- **Code lines 41-48** shows the capabilities section where you can set the maximum number of instances of a browser that should run at any given time concurrently. Here you can also set the type of browser that you want to use ('chrome') in our case as this is the most popular browser used today
- **Code line 58** – specifies sync mode. In brief, it's a helper module to run WebdriverIO commands synchronously. It overwrites global functions depending on the test framework (e.g. for Mocha *describe* and *it*) and uses Fibers to make commands of WebdriverIO using the wdio testrunner synchronous. We will cover this in detail in upcoming lectures.

- **Code lines 61-63** - defines the type of logging you wish to record when running tests. We have set this to silent for now, but you could use the following options in the future:
    - **verbose**: everything gets logged
    - **silent**: nothing gets logged
    - **command**: url to Selenium server gets logged (e.g. [15:28:00] COMMAND GET "/wd/hub/session/dddef9eb-82a9-4f6c-ab5e-e5934aecc32a/title")
    - **data**: payload of the request gets logged (e.g. [15:28:00] DATA {})
    - **result**: result from the Selenium server gets logged (e.g. [15:28:00] RESULT "Google")
- **Code line 74** – sets the screenshotPath location where screenshots will be recorded that are taken during out tests
- **Code line 83** – sets the waitforTimeout value which is the default timeout for all waitForXXX commands.
- **Code lines 134-173** – contains the Hooks section. We go through this in more detail later (module 15) but, in brief, you can use this to interfere with the test process in order to enhance it and build services around it. You can either apply a single function to it or an array of methods. If one of them returns with a promise, WebdriverIO will wait until that promise got resolved to continue.
    - So, for example, we may want to do something before our tests run. We can use beforeSession method within the hooks section which gets executed just before initialising the webdriver session and test framework. It allows you to manipulate configurations depending on the capability or spec.

You can read more about the configuration in greater detail using the official webdriverio documents located here: http://v4.webdriver.io/guide/testrunner/configurationfile.html

Module 5 – Mocha

This module covers Mocha. Mocha (no, not the coffee ☺) is a feature-rich JavaScript test framework running on Node.js and in the browser, making asynchronous testing simple and fun. Mocha tests run serially, allowing for flexible and accurate reporting, while mapping uncaught exceptions to the correct test cases.

Lecture 19 - Mocha – Introduction

Lecture Resources on Udemy:

- https://mochajs.org/

In this lecture, I introduce you to Mocha. Here I explain its purpose, why we should use it and the benefits.

Key Points:

- Mocha comes pre-bundled with two function calls; these are,
  - **describe()** – is simple a way to group our tests in Mocha. It enables us to create a series of tests together. describe() takes two arguments. The first is the name of the test group. The second is the call back function which is a function that is to be executed after another function has finished executing.
  - **it()** – is a way to describe the individual test case which is nested in the describe() block. it() should be described in a way that makes sense for the given test case.
  - An example of describe and it is shown below:

```
describe('Test Contact Us form WebdriverUni', function() {
    it('Should be able to submit a successful submission via contact us form',
function(done) {
        browser.setValue('[name="first_name"]', 'Joe');
        browser.setValue('[name="last_name"]', 'Blogs');
        browser.setValue('[name="email"]', 'joe_blogs@mail.com');
        browser.setValue('textarea', 'How much does product x cost?');
        browser.click('#form_buttons .contact_button:nth-of-type(2)');
    });

    it('Should not be able to submit a successful submission via contact us form as
all fields are required', function(done) {
        browser.setValue('[name="first_name"]', 'Joe');
        browser.setValue('[name="last_name"]', 'Blogs');
        browser.setValue('[name="email"]', 'joe_blogs@mail.com');
        browser.click('#form_buttons .contact_button:nth-of-type(2)');
```

- In the image above, we are conducting two inner tests (notice the use of two it() definitions) within one describe block.
- We have defined a describe value and stated:
    - Test contact us form webdriveruni
    - This means we want to test the contact us form that resides on my webdriveruniveristy website.
    - **By the way - all fields are required in the form in order to submit the form successfully**
- Next, review the it() statements
- The **first** should be able to submit a successful submission via the contact us form
    - We then provide values for the following fields:
        - first_name
        - last_name
        - email
        - textarea
    - We then select the submit button using the browser.click() function.
- The **second** should be an unsuccessful submission because we only provide values for the following fields:
    - Values provided are:
        - first_name
        - last_name
        - email
    - Notice how this test does not include the textarea field
    - We then try to select the submit button using the browser.click() function.

We are expecting the first it() test to pass as we have provided values for all the mandatory fields. We expect the second if() test to fail because we have not provided a value for the textarea field (which is mandatory).

Lecture 20 - Mocha – Structuring & Combining Tests Part 1

Lecture Resources on Udemy:

- https://www.npmjs.com

In this lecture we'll start using Mocha to form our test cases. We are going to combine our two existing tests into one .js file and then use Mocha to construct our tests.

Instructions:

How to install Mocha

1. We need to download Mocha using the npm website. Go to https://www.npmjs.com and search for Mocha
2. The first result will be mocha and if you click on the link, you will be shown a description of this package and more importantly, you will see an "install" section that contains the following command: **npm i mocha**
3. Go back to GitBash/ iTerm2 and make sure to be in the webdriverioFramework directory
4. We are going to use a similar command to the one above to install Mocha except ours would be slightly different:
    a. Type: **npm install -save-dev mocha@latest**

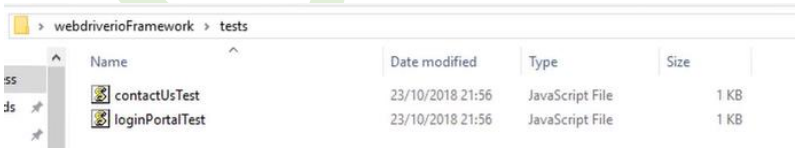
```
GBruno@Gi MINGW64 ~/Desktop/webdriverioFramework
$ npm install -save-dev mocha@latest
```

    b. This will install the Mocha packages and you will see our package.js file update to contain the new dependency
    c.

Instructions:

How to combine our two tests

1. Use windows explorer (for windows) or Finder (on Mac) and navigate to our "tests" folder as shown:



2. We need to create a new file. Select a new notepad (.txt) file to begin with and call it webdriverUniversityTest.js
    a. Make sure the file extension has been replaced from .txt to .js – meaning your OS now recognises this as a Javascript file instead of a text file. To confirm this, right click the file and look at the properties.
3. Open up the webdriverUniversityTest.js file using Sublime Text
4. We now start constructing the file.
    a. We add a "describe" and "it" block to the new file
        i. "describe" explains what we are trying to perform in this test

ii. “it” explains the purpose of the specific test case

b. We then bring across the contactUsTest.js code

c. We change “browser” to “return browser”

5. By the end of this lecture, your webdriverUniversityTest.js should look similar to the screenshot below:

We continue with this in the next lecture

Lecture 21 - Mocha – Structuring & Combining Tests Part 2

Lecture Resources on Udemy:

- None

In this lecture we continue with the changes being explained in part 1

Instructions:

Continuing to combine our two tests:

1. Now we need to include the loginPortalTest code into our webdriverUniversityTest.js file
2. Use windows explorer (for windows) or Finder (on Mac) to open up the existing loginPortalTest.js file
3. Copy all the code to the clipboard
4. Go back to the webdriverUniversityTest.js file and add another "it" block of code under the existing block of code from the previous lecture (see screenshot below for a preview of how your code should look)
5. The second "it" description should read:
    a. "check that the login button opens the login portal page"
6. Now copy the code from your clipboard underneath the second "it" line
7. Again, make sure to change "**browser**" to "**return browser**"
8. Your code should now look like this:

```
webdriverUniversityTest.js
describe("Verify whether webdriveruniversity links on homepage work correctly", function() {
    it("check that the contact us button opens the contact us page", function(done) {
        return browser
        .setViewportSize({
            width: 1200,
            height: 800
        })
        .url('http://www.webdriveruniversity.com/')
        .getTitle().then(function(title) {
            console.log('Title is: ' + title);
        })
        .click("#contact-us")
        .pause(3000)
    });

    it("check that the login button opens the login portal page", function(done) {
        return browser
        .url('http://www.webdriveruniversity.com/')
        .click('#login-portal')
        .getTitle().then(function(title) {
            console.log('Title is: ' + title);
        })
    });
});
```

9. Save your file (file > save)

By this point, we have now combined our two individual test files 'contactUsTest' and 'loginPortalTest' into a single webdriverUniversityTest.js file and have used the Mocha "describe" and "it" keywords to separate and simplify our two tests. Next, we will look at executing it.

Lecture 22 - Mocha – Reviewing & Executing Our New & Improved Tests

Lecture Resources on Udemy:

- None

In this lecture we review our changes from part 1 and 2 and execute our new and improved tests.

Our current test looks like this:

webdriverUniversityTest.js

```
describe("Verify whether webdriveruniversity links on homepage work correctly", function() {
    it("check that the contact us button opens the contact us page", function(done) {
        return browser
        .setViewportSize({
            width: 1200,
            height: 800
        })
        .url('http://www.webdriveruniversity.com/')
        .getTitle().then(function(title) {
            console.log('Title is: ' + title);
        })
        .click("#contact-us")
        .pause(3000)
    });

    it("check that the login button opens the login portal page", function(done) {
        return browser
        .url('http://www.webdriveruniversity.com/')
        .click('#login-portal')
        .getTitle().then(function(title) {
            console.log('Title is: ' + title);
        })
    });
});
```

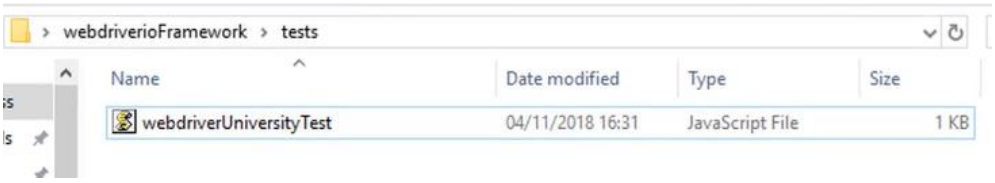
Key Points:

- Mocha provides a great way to break down our tests into individual sections and provides a great way to provide descriptions of our overall test case and our individual tests.
- Notice how much cleaner our code looks and how much more readable it is
- Maintainability has also been greatly improved, as now we can amend our tests using this single file

Instructions:

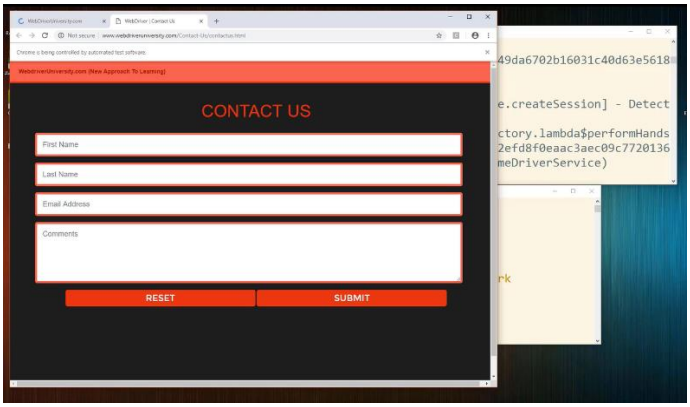
Cleaning up our retired test files and executing our tests

1. We no longer need the 'contactUsTest' and 'loginPortalTest' files, since we've combined these tests into the webdriverUniversityTest.js file.
2. Use windows explorer (for windows) or Finder (on Mac) to delete them from our 'tests' directory so that only the below file remains:



3. Open GitBash / iTerm2 and move to the webdrivenioFramework folder
4. In another instance of GitBash / iTerm2, start the selenium server

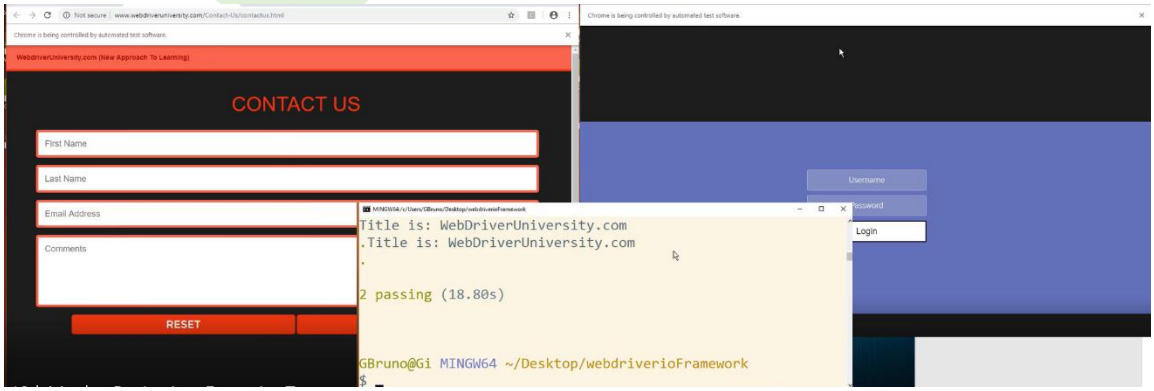
5. Return to your first instance of GitBash / iTerm2 and write the following command:
    a. ./node_modules/ .bin/wdio
    b. This will initiate our tests:



6. I make one small change to our webdriverUniversityTest.js file so that we can see the second test execute (as it's too quick to see). If you wish to do the same, add the following code:

```
describe("Verify whether webdriveruniversity links on homepage work correctly", function() {
    it("check that the contact us button opens the contact us page", function(done) {
        return browser
        .setViewportSize({
            width: 1200,
            height: 800
        })
        .url('http://www.webdriveruniversity.com/')
        .getTitle().then(function(title) {
            console.log('Title is: ' + title);
        })
        .click("#contact-us")
        .pause(3000)
    });

    it("check that the login button opens the login portal page", function(done) {
        return browser
        .url('http://www.webdriveruniversity.com/')
        .click('#login-portal')
        .getTitle().then(function(title) {
            console.log('Title is: ' + title);
        })
        .pause(3000)
    });
});
```

7. I think initiate the test again using command:
    a. ./node_modules/ .bin/wdio
8. You should then see the tests execute similar to the screenshots shown below:



As you can see, two tests have passed, and the output of the tests is summarised in GitBash ☺

Module 6 – WDIO Sync Mode

Lecture 23 - WDIO – Simplify Tests and Sync Mode

In this lecture, we take a look at Sync mode and the difference between Synchronous and asynchronous requests.

Lecture Resources on Udemy:

- http://v4.webdriver.io/guide/getstarted/v4.html

In this lecture we look at Sync Mode and go through the difference between Synchronous and Asynchronous tests.

```
// Example 1 - Synchronous
var result = database.query("SELECT * FROM exampleTable");
console.log("Query has finished");
console.log("Hello World");

// Example 2 - Asynchronous
database.query("SELECT * FROM hugetable", function(result) {
console.log("Query has finished");
});
console.log("Hello World ");

OUTPUT:
//Example1
Query has finished
Hello World

//Example2
Hello World
Query has finished
```

If we look at example 1 – we can see that we have created a variable that runs a query to retrieve data (selects all records from table exampleTable). We then log two statements to the console, "Query has finished" and "Hello World".

Now, if we look at the output from Example 1 towards the end of the image, we see that the ordering of the output is shown as:

Query has finished
Hello World

Because this request is synchronous, it means that the first statement has to complete before the next line is executed.

If we look at example 2 – we are using an asynchronous request by calling database.query directly and then use a function to output the result. We then have a second console.log that sites outside the function block.

If we review the console output for this example, we see one key difference:

Hello World
Query has finished

Notice that the ordering of the above is different. The code does not have to wait for the first part to complete before progressing with the next part. This is the key difference between the two request types.

Webdriverio introduced version four of its code recently. One of the key differences between now and previous versions is that now, webdriverio is all synchronous.

It's all synchronous

One of the probably biggest changes is the way WebdriverIO handles async in the wdio testrunner now. Many developers that moved away from Java to run their e2e tests in Node.js struggled with async handling of commands in their test suites. Also beginners had to learn the concept of promises to properly build up reliable tests. This is now over. All commands now block the execution of the test process until they've resolved. No `then` calls have to be done anymore, we can just assign the direct response of the Selenium server to a variable, like:

```
describe('webdriver.io page', function() {
    it('should have the right title', function () {
        browser.url('/');
        var title = browser.getTitle();
        assert.equal(title, 'WebdriverIO - WebDriver bindings for Node.js');
    });
});
```

WebdriverIO uses the popular Fibers package which is a Node.js plugin and compatible with all operating systems and various Node.js versions. It is also used by other big projects like Meteor and is pretty stable and in active development. If you have used generators in v3 you can easily upgrade to v4 by removing the asterisks in your spec definitions as well as all `yield` words in your tests (there is no support for generators in v4 anymore).

We continue to look at sync requests in the next lecture

Lecture 24 - WDIO – Configuring Our Tests to Use Sync Mode

In this lecture, we look at changing our WDIO.conf file to show you how we can run our tests asynchronously, if needed.

Lecture Resources on Udemy:

- http://v4.webdriver.io/guide/testrunner/configurationfile.html

Instructions:

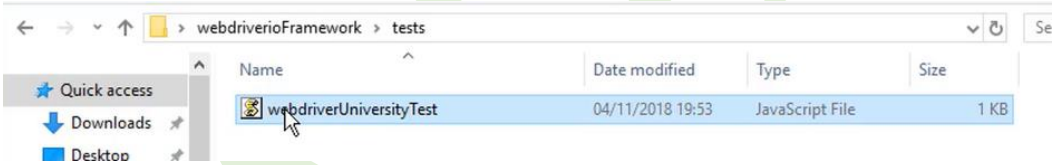
1. As you know, a lot of our configuration is placed in our WDIO.conf testrunner class file. There is a setting for 'sync' as shown:

```
55      // By default WebdriverIO commands are executed in a synchronous way using
56      // the wdio-sync package. If you still want to run your tests in an async way
57      // e.g. using promises you can set the sync option to false.
58      sync: false,
59      //
```

2. Change the value from false to true, as shown:

```
55      // By default WebdriverIO commands are executed in a synchronous way using
56      // the wdio-sync package. If you still want to run your tests in an async way
57      // e.g. using promises you can set the sync option to false.
58      sync: true,
59      //
```

3. Once you have made the change above, save the file and close Sublime Text
4. We now need to make changes to our webdriverUniversityTest.js folder, located here:



5. Open this file using Sublime Text
6. The first change is that we no longer need to use the following:

```
3       return browser
```

7. Delete this line
8. Do the same for line 16:

```
16          return browser
```

9. We can now call browser directly, so add the following to your first "it" test:

```
1  describe("Verify whether webdriveruniversity links on homepage work correctly", function() {
2      it("check that the contact us button opens the contact us page", function(done) {
3          browser.setViewportSize({
4              width: 1200,
5              height: 800
6          })
7          browser.url('http://www.webdriveruniversity.com/')
8          browser.getTitle().then(function(title) {
9              console.log('Title is: ' + title);
10         })
11         browser.click("#contact-us")
12         browser.pause(3000)
13     });
```

10. We can make the same changes to our second "it" test, so also make the following changes:

```
15     it("check that the login button opens the login portal page", function(done) {
16         browser.url('http://www.webdriveruniversity.com/')
17         browser.click('#login-portal')
18         browser.getTitle().then(function(title) {
19             console.log('Title is: ' + title);
20         })
21         browser.pause(3000)
22     });
23 });
24
```

11. The next change is to remove the following *.then* command as shown from our first "it" test:

```
7         browser.url('http://www.webdriveruniversity.com/')
8         browser.getTitle().then(function(title) {
9             console.log('Title is: ' + title);
10        })
11        browser.click("#contact-us")
12        browser.pause(3000)
13    });
```

12. Change this to the following:

```
7         browser.url('http://www.webdriveruniversity.com/');
8         var title = browser.getTitle();
9         console.log('Title is: ' + title);
10        browser.click("#contact-us");
11        browser.pause(3000);
12    });
```

13. We can then do something similar to our second "it" test block. It currently looks like this:

```
14    it("check that the login button opens the login portal page", function(done) {
15        browser.url('http://www.webdriveruniversity.com/')
16        browser.click('#login-portal')
17        browser.getTitle().then(function(title) {
18            console.log('Title is: ' + title);
19        })
20        browser.pause(3000)
21    });
```

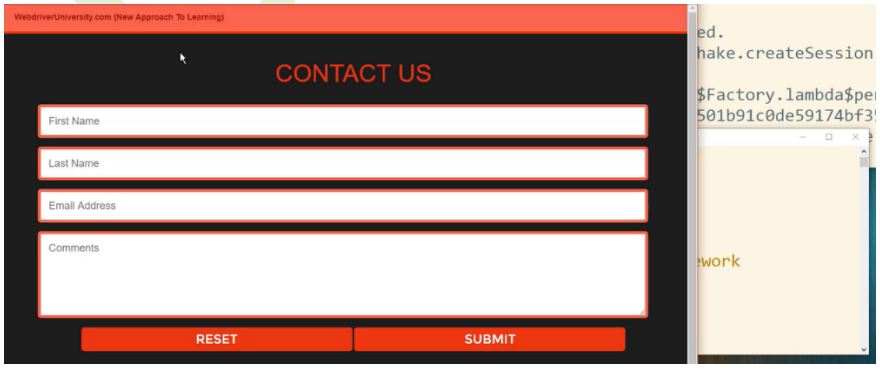
14. Change this to the following:

```
14    it("check that the login button opens the login portal page", function(done) {
15        browser.url('http://www.webdriveruniversity.com/');
16        browser.click('#login-portal');
17        var title = browser.getTitle();
18            console.log('Title is: ' + title);
19        browser.pause(3000);
20    });
```

15. Once you have made the changes above, save your file and then close the file
16. Open up GitBash/ iTerm2 and move into the webdriverioFramework directory
17. Run the following command to start the selenium server:
    a. `$ ./node_modules/.bin/selenium-standalone start`
18. Open up a second instance of GitBash/ iTerm2 and move into the webdriverioFramework directory
19. Run the following command to execute our test:

```
GBruno@Gi MINGW64 ~/Desktop/webdriverioFramework
$ ./node_modules/.bin/wdio
```

20. You should see the test begin with the following output:



21. If you review the console output, you should see two tests have passed, as shown:

```
2 passing (20.20s)
```

22. You should also see the title of the two pages used in our two tests.

Module 7 – Selenium Standalone & NPM Scripts

Lecture 25 - Selenium Standalone & NPM Scripts

Lecture Resources on Udemy:

- None

In this lecture we looked at the Selenium Standalone Service where I showed you how we could use npm scripts to enhance our framework.

Key Points:

- Before this lecture, every time we wanted to execute a test, we had to manually start the selenium server in a secondary GitBash/ iTerm2 window
- This is repetitive and a poor use of our time, so we can automate this process using npm scripts
- This way, the server will automatically start when we execute our tests to run

Instructions:

1. Use windows explorer (for windows) or Finder (on Mac) to open the webdriverioFramework directory
2. Next, open up the WDIO file using Sublime text
3. Scroll down to the services section (use the search function to look up “services”)
4. The part we are interested in is the Test Runner Services section
    a. Uncomment (remove the //) from line 114 (services: [ ];)
    b. Then add the following code inside the square brackets:
        i. ‘selenium-standalone’
    c. You code should now look something like this:

```
// Test runner services
// Services take over a specific job you don't want to take care of. They enhance
// your test setup with almost no effort. Unlike plugins, they don't add new
// commands. Instead, they hook themselves up into the test process.
 services: ['selenium-standalone'],
// Framework you want to run your specs with.
// The following are supported: Mocha, Jasmine, and Cucumber
// see also: http://webdriver.io/guide/testrunner/frameworks.html
```

5. We now need to install the selenium-standalone-service package
6. Open up GitBash / iTerm2 and navigate to our webdriverioFramework directory
7. Type the following command
    a. **npm install wdio-selenium-standalone-service -save-dev** + press enter
    b. This will install the package
8. Now we need to create a npm script.
9. Open up the package.json file using sublime text
10. On line 7 you should see with have a “test” section that looks like the below:

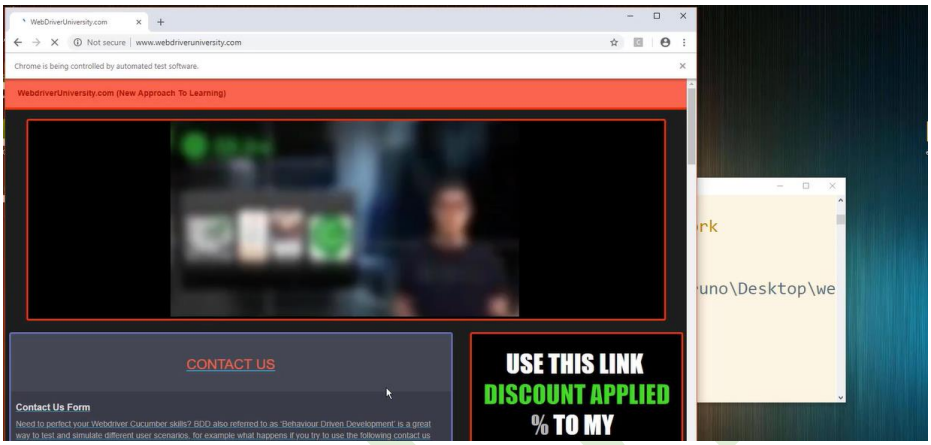
```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1"
},
```

a. We need to remove the echo command and type “wdio”. Your code should then look like this:

```
"scripts": {
  "test": "wdio"
},
```

b. Once you have completed the changes above, save your file and close sublime text

11. Now go back to GitBash/ iTerm2 and execute the test using the command:
    a. **npm test** + press enter
    b. You will see the test now runs without having to manually start the selenium server as shown ☺ :



Module 8 – Environments & Base URL

Lecture 26 - Base URL Setup

In this module, we look at setting a base URL and handling multiple environments/ websites at runtime.

Lecture Resources on Udemy:

- None

In this lecture we looked at setting a base URL in our WDIO file. This means we stick to the DRY (**D**on't **R**epeat **Y**ourself) rule by not having to define a URL in each and every test case. We can also alter the base URL during runtime (if needed).

Key Points:

- Before this lecture, we were defining a URL in every test (e.g. to webdriveruniversity.com)
- This is bad practise as we are repeating the same code multiple times
- A better solution would be to use a base URL
- This means we can then refer to the base URL going forward, removing code duplication

Instructions:

How to set up and use a base URL

1. Use windows explorer (for windows) or Finder (on Mac) to open the webdrivenioFramework directory
2. Next, open up the WDIO file using Sublime text
3. At the top of the WDIO file, add a new line of code using the following syntax:
    a. **var baseUrl = ' http://www.webdriveruniversity.com';**

```
1  var baseUrl = 'http://www.webdriveruniversity.com';
2
```

    b. We are creating a new variable and setting the value of the variable to the webdriveruniversity website.
4. Then we need to find the baseUrl syntax in the WDIO file (use the search function and search for "baseURL" – it should be around line 82)
    a. Currently this looks like:

```
82      baseUrl: 'www.webdriveruniversity.com',
```

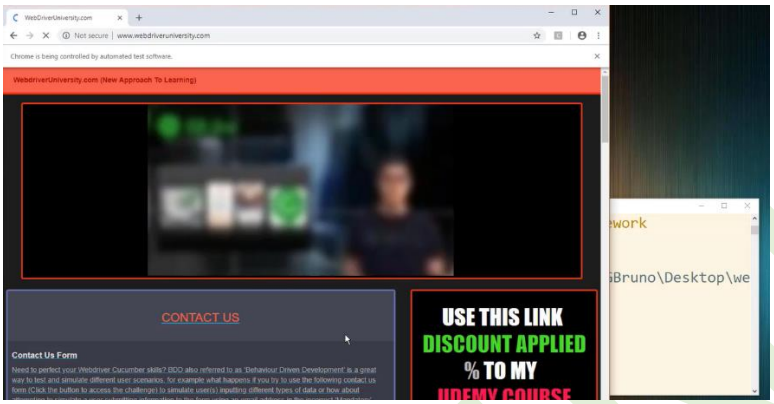
    b. We need to change this value to now use the variable from line 1:

```
82      baseUrl: baseUrl,
```

5. Now use windows explorer (for windows) or Finder (on Mac) to open the webdrivenioFramework directory and move into the tests folder and open the webdriverUniversityTest.js test using Sublime Text
6. We now need to amend line 7 and 15 as this refers to the browser.url and currently sets this to the webdriveruniversity.com website:

```
7       browser.url('http://www.webdriveruniversity.com/');
15      browser.url('http://www.webdriveruniversity.com/');
```

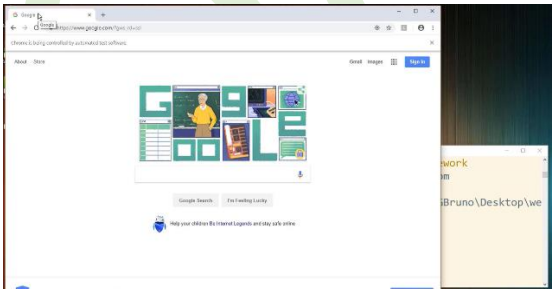
7. Change both lines above to read:
```
browser.url('/');
```
8. Save the file and close Sublime Text
9. Open up GitBash / iTerm2 and navigate to our webdriverioFramework directory
10. Now we are going to execute the test using the following command:
    a. **npm test** + press enter
    b. You should see that the test now executes and it's now using the baseUrl to direct the tests to the webdriveruniversity.com website!



Instructions:

How to use a different URL at runtime

1. When executing a test using npm … on GitBash/ iTerm2 like so:
```
GBruno@Gi MINGW64 ~/Desktop/webdriverioFramework
$ npm test
```
2. We can amend the command to use a different URL, for example:
```
GBruno@Gi MINGW64 ~/Desktop/webdriverioFramework
$ npm test -- --baseUrl=http://www.google.com
```
3. This should force our test to go to www.google.com but our test should report a failure, as our test cases ('contactUsTest' and 'loginPortalTest' tests) are unable to pass using the google homepage:



Lecture 27 - Handling Multiple Environments During Runtime

Lecture Resources on Udemy:

- None

In this lecture, we looked at handling multiple environments during runtime. This is a common scenario, as most companies will use multiple environments to perform different tests in the real world.

For example, a company may have a website to perform sprint related work. They may have another website to perform regression testing. In this lecture, we look at adding an IF statement in our WDIO file to handle this situation.

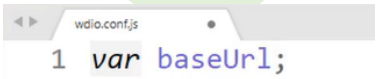
Key Points:

- Companies often test against different instances of their website
    - Regression testing may be performed in once instance; and
    - Sprint related works may be performed in another
- We can handle this situation by setting an IF condition in our WDIO file
- This will mean, our tests can be instructed to run on the target environment we intend to test

Instructions:

1. Use windows explorer (for windows) or Finder (on Mac) to open the webdriverioFramework directory
2. Next, open up the WDIO file using Sublime text
3. At the top of the file, we currently have the baseUrl pointing go the webdriveruniversity website:

4. We need to amend this code by removing the value of the baseUrl variable, like so;

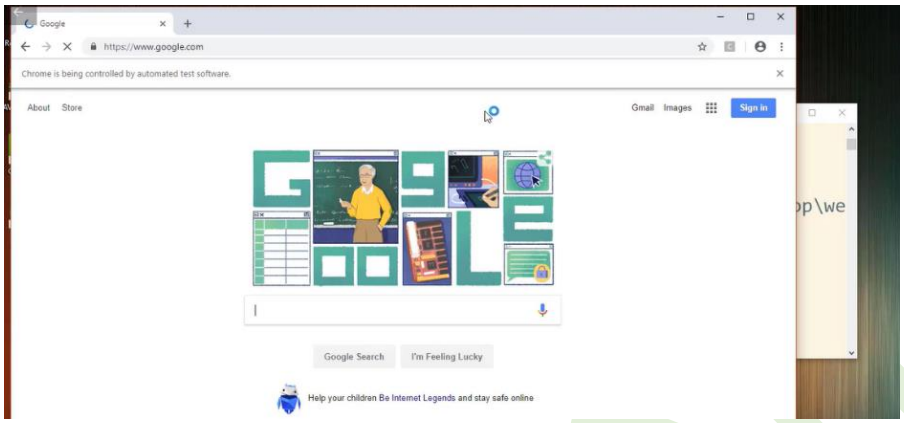


5. We then use an IF statement that uses process.env.SERVER
    a. The process.env global variable is injected by the Node at runtime for your application to use and it represents the state of the system environment your application is in when it starts.

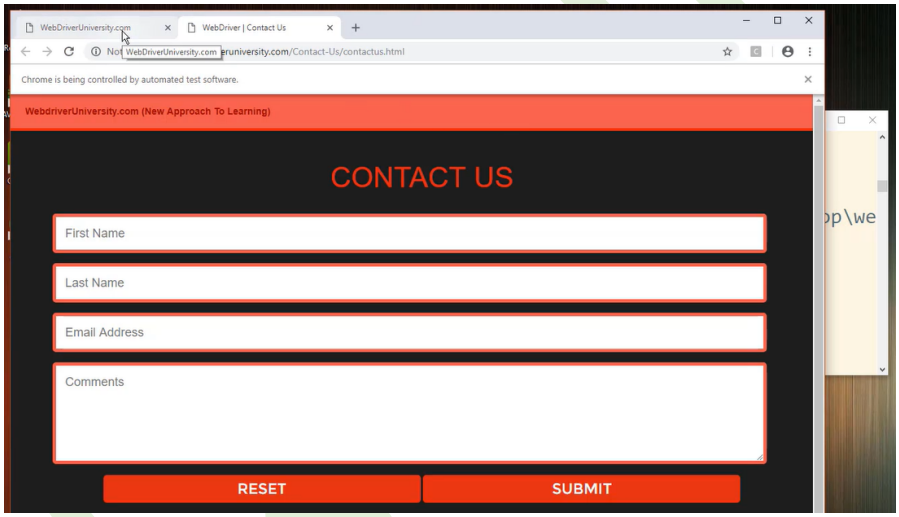
```
3 if(process.env.SERVER === 'prod') {
4     baseUrl = 'https://www.google.com';
5     } else {
6         baseUrl= "http://www.webdriveruniversity.com";
7     }
8
```

6. So, in our case, we are looking to see if we are running a Production server and if so use Google as our baseUrl, otherwise use the webdriveruniversity

7. Save your WDIO file with the above changes made
8. Open up GitBash / iTerm2 and navigate to our webdriverioFramework directory
9. Now we are going to execute the test using the following command:
    a. **SERVER=prod npm test** + press enter
10. You should see that our tests are now attempting to use www.google.com because we have declared the server as “prod” when executing the test:



11. Now if we try and run the test again without declaring the “prod” server, the ‘else’ part of the statement will be used, meaning our tests should refer to webdriveruniversity.com
    a. Use command: **npm test** + press enter:



This lecture showed you how easy it is to handle multiple environments. Tests can be directed to a target environment of your choice when initiating a test.

Module 9 – Logging

Lecture 28 - Logging During Runtime

Useful logs can provide the developer, especially when someone has to debug/maintain someone else's code and can tremendously help when trying to understand what the code actually does. In this module we cover logging.

Lecture Resources on Udemy:

- http://v4.webdriver.io/guide/getstarted/configuration.html#logLevel

In this lecture we looked at logging additional information when running our tests. This will greatly help when debugging our code.

Remember, there are different types of logging that's possible in webdriverio:

- Level of logging verbosity.
  - **verbose**: everything gets logged
  - **silent**: nothing gets logged ← what we've been using up to this point
  - **command**: url to Selenium server gets logged (e.g. [15:28:00] COMMAND GET "/wd/hub/session/dddef9eb-82a9-4f6c-ab5e-e5934aecc32a/title")
  - **data**: payload of the request gets logged (e.g. [15:28:00] DATA {})
  - **result**: result from the Selenium server gets logged (e.g. [15:28:00] RESULT "Google")

Instructions:

1. Open up GitBash / iTerm2 and navigate to our webdriverioFramework directory
2. Type in the following command:
   a. **npm test -- -- logLevel=verbose** + press enter

3. This will trigger our tests but also produce Verbose reports where everything now gets logged
4. Scroll through the console information and notice at all the data that is now being logged/ recorded
   a. It records the commands that are being used (POST etc.)
   b. It records a session instance
   c. It records the height and width of our browser
   d. ID information (e.g. the test is looking for a specific element with a particular id)
   e. And much more

5. This information is useful to us because it gives a clear step by step process of the what our test is doing when it's processing
6. We can also use this information to identify the cause of problems when our code doesn't perform as expected

If you want a more detailed explanation of logging, please refer to the official webdriverio documentation, here: http://v4.webdriver.io/guide/getstarted/configuration.html#logLevel

I tend to use Silent or Verbose logging during my day job but further reading on the other types of testing can be found using the official documentation linked above.

Module 10 – Node Assertions

Lecture 29 - Implementing Node Assertions

The assert module provides a way of testing expressions. For example, if the expression evaluates to 0, or false, an assertion failure is being caused, and the program is terminated. Think of assertions as a condition (e.g. we expect result A to be returned and if result A does not return then the test should fail).

In this module, we take a look at node assertions and explain what is it and how it can be used to improve our tests.

Lecture Resources on Udemy:

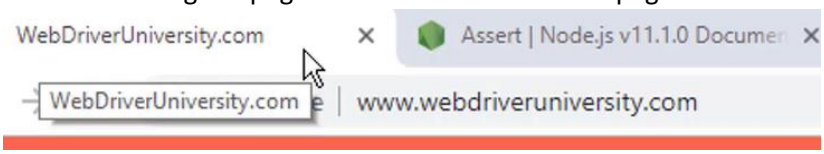
- https:/nodejs.org/api/assert.html

Key Points:

- In this lecture we looked node assertions
- A node assertion is a way to check a result completes as expected.
- For example, we might want to navigate to a particular website and we can prove the website is correct by confirming the URL address

Instructions:

1. Open up GitBash / iTerm2 and navigate to our webdriverioFramework directory
2. Go to the tests folder and open up our webdriverUniversityTest.js file using Sublime Text
3. In your web browser, visit the following webpage:
    a. https://nodejs.org/api/assert.html
    b. This webpage is the api documentation for assert. This is what we will be integrating into our test to ensure we get the expected results when executing our test
4. The api that we will be using in this lecture is **assert.equal**():
    - assert.doesNotThrow(fn[, error][, message])
    - assert.equal(actual, expected[, message])
    - assert.fail([message])
5. This method takes two arguments:
    a. **actual** – which will be the actual output of a test
    b. **expected** – which will be the result we are expected
        i. We can then return a message
6. We are now going to include an assertion into our test to validate that we are on the correct webpage (www.webdriveruniversity.com) when our test runs
    a. We will be using the page title of the website homepage to validate against

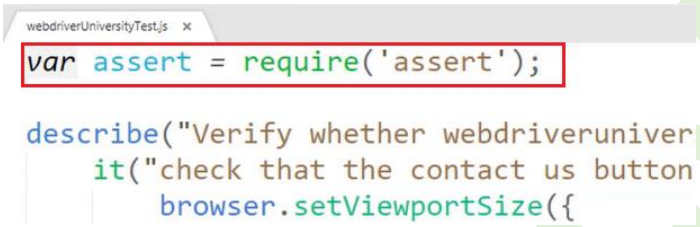


    b. As you can see above, the website title is “WebDriverUniversity.com”

7. Go back to our test (webdriverUniversityTest.js) and look at the current block of code:

```
describe("Verify whether webdriveruniversity links on homepage work correctly", function() {
    it("check that the contact us button opens the contact us page", function(done) {
        browser.setViewportSize({
            width: 1200,
            height: 800
        })
        browser.url('/');
        var title = browser.getTitle();
        console.log('Title is: ' + title);
        browser.click("#contact-us");
        browser.pause(3000);
    });
```

8. We are going to add an assertion to check that we are visiting the correct webpage by checking the returned title with the expected title
9. First, we must add the assert library to our test. We do this at the very top of the code, as shown:



10. Then we add the assertion into our code block, like shown:

```
var assert = require('assert');

describe("Verify whether webdriveruniversity links on homepage work correctly", function() {
    it("check that the contact us button opens the contact us page", function(done) {
        browser.setViewportSize({
            width: 1200,
            height: 800
        })
        browser.url('/');
        var title = browser.getTitle();
        assert.equal(title, 'WebDriverUniversity.com');
        console.log('Title is: ' + title);
        browser.click("#contact-us");
        browser.pause(3000);
    });
```

11. In the code above, we are doing the following
    a. We are using the api method **assert.equal()** as explained on the api documentation webpage earlier
    b. We are returning the webpage title as the first argument (**title**)
    c. We are then providing an expected result value (**'WebDriverUniversity.com'**)
    d. When we run our test, we expect the title to match the expected result value and if a mismatch is found, it should be reported
12. Next, we add the same code to our second test code block as shown:

```
    it("check that the login button opens the login portal page", function(done) {
        browser.url('/');
        browser.click('#login-portal');
        var title = browser.getTitle();
        assert.equal(title, 'WebDriverUniversity.com');
        console.log('Title is: ' + title);
        browser.pause(3000);
    });
```

13. And then we save the file

14. We are now ready to run the test by opening GitBash / iTerm2, ensuring we are in the webdriverUniversityFramework directory and then running the following command:
    a. **npm test**
    b. The test should start processing
    c. We are then provided with the following output which shows the two titles of the pages related to our tests:

15. Now we make one further change to our webdriverUniversityTest.js file using Sublime Text by changing the following line:

```
browser.url('/');
var title = browser.getTitle();
assert.equal(title, 'WebDriverUniversity2.com');
console.log('Title is: ' + title);
browser.click("#contact-us");
browser.pause(3000);
```

16. Notice that have added a '2' in the URL. This is because we want the webpage title and the expected result to mismatch (to demonstrate to you what happens)
17. Save the file and restart the test using GitBash / iTerm2 using command:
    a. **npm test**
18. The test will again run but this time take a close look at the command line output:

```
npm ERR! Test failed.  See above for more details.

GBruno@Gi MINGW64 ~/Desktop/webdrivierioFramework
$
```

19. This time a failure has reported and if we scroll up, we see:

20. One test has failed because the page title did not match the expected page title
21. Now go back to the .js file, remove the "2" from the URL and save (remove the error)

Module 11 – Chai

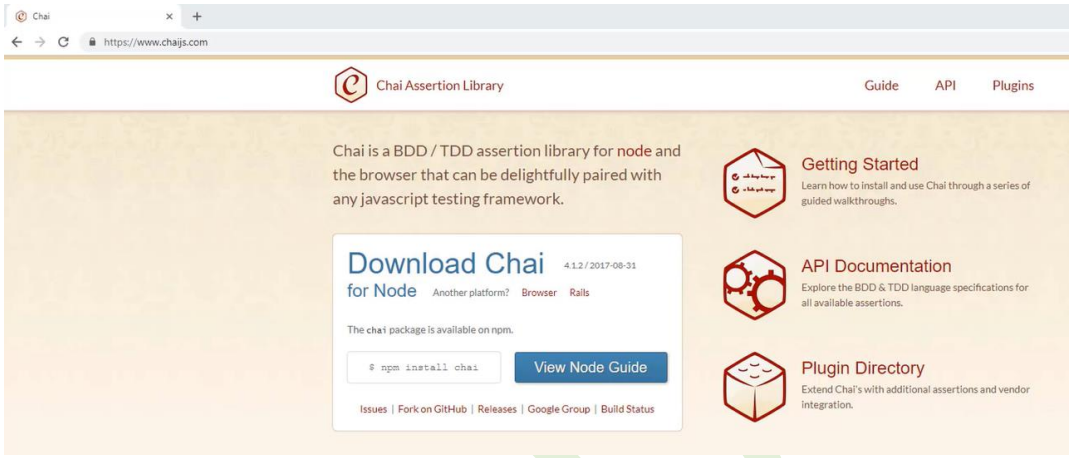
In this module we take a look at Chai. Chai is a BDD / TDD assertion library for node and the browser that can be delightfully paired with any Javascript testing framework.

Lecture 30 – Chai – Advanced Assertions & Code Example – Part 1

In this lecture we start adding advanced assertions into our tests using Chai. This module is broken down into sections and we keep improving our tests as we advance.

Lecture Resources on Udemy:

- https://www.chaijs.com/



Chai is a type of assertion that allows us to simply and improve the overall process of adding assertions to our tests.

Key Points:

- Chai is a BDD / TDD assertion library for node
- Chai has several interfaces, given the developer choice to find one they are most comfortable using
- The Chai BDD style provides us an expressive language that is easy to read
- The Chain TDD style provides us a classical feel
- Examples of the Chai styles are shown below:

- You can read more about the above styles using these links:
  - **Should** - https://www.chaijs.com/guide/styles/#should
  - **Expect** - https://www.chaijs.com/guide/styles/#expect
  - **Assert** - https://www.chaijs.com/guide/styles/#assert

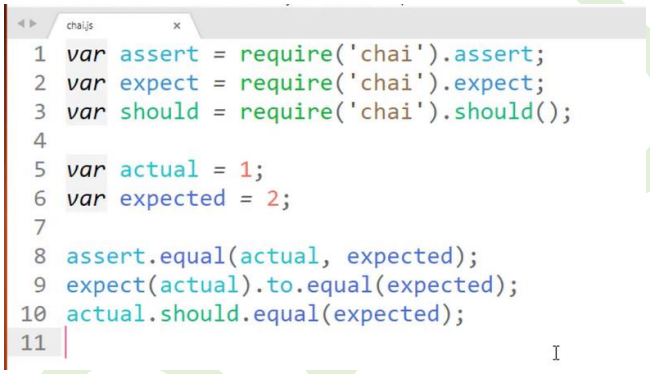
Instructions:

Adding the Chai library to our project

1. Open up GitBash / iTerm2 and navigate to our webdriverioFramework directory
2. Type in the following command:
    a. **npm install chai@latest -save-dev** + press enter
    b. This will start installing the Chai libraries and adding the Chai dependency to our package.json file

Demonstration of Chai (simple test example)

1. Use windows explorer (for windows) or Finder (on Mac) to open the webdriverioFramework directory and move into the "tests" folder directory
2. Open up Sublime Text and create a new file (leave it blank for now) and save it to our 'tests' folder.
    a. Call the file 'chai.js' (note the file should be called chai but the file type should be a Javascript .js file)
3. Change the language of the syntax in Sublime for the file to JavaScript using
    a. View > Syntax > JavaScript
4. Add the following code to your file (I explain what each line below the image):



    a. The first three lines (lines 1-3) adds the libraries and assigns them to variables that are required for our Chai assertions
    b. Lines 5 and 6 create two more variables and assigns them values of 1 (for the *actual* variable) and 2 (for the *expected* variable). We then use these variables further below.
    c. Line 8 demonstrates the usage of the '**assert'** style that uses the 'equal' function. Equal takes two arguments, the first is the actual value and the second is the expected value
        i. So, in our case, the above test is similar to writing assert.equal(1, 2);
    d. Line 9 demonstrates the usage of the '**expect'** style that is similar to line 8 but we provide the *expect* function with the *actual* value and then use *.to.equal* to compare it to the *expected* variable value.
        i. So, in our case, the above test is similar to writing expect(1).to.equal(2);
    e. Line 10 demonstrates the usage of the '**should'** style. This again is similar but this time we use the variable *actual* first then the .should.equal methods and then provide an argument using the *expected* variable
5. Save your file with the above changes. We then execute the test in the next lecture.

Lecture 31 – Chai - Advanced Assertions & Code Example – Part 2

In this lecture we continue on from the previous lecture and execute the newly created test (chai.js).

Lecture Resources on Udemy:

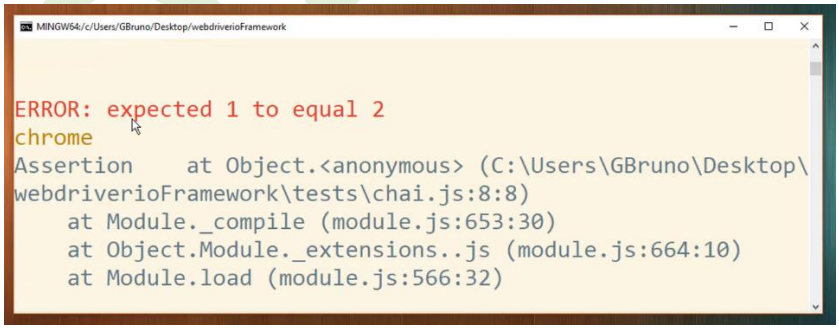
- https://www.chaijs.com/

Instructions:

Executing our newly created test

1. Open up GitBash / iTerm2 and navigate to our webdriverioFramework directory
2. Important – We now have two tests in the 'tests' folder, and so we need to target only one of those tests (chai.js)
3. We can do this by using the following command:
    a. **npm test -- -- spec=tests/chai.js** + press enter
    b. The above command tells node to look for a specific test by giving the spec command a directory and test name value (tests/ is the directory where the test resides, chai.js is the name of the test)
4. You should see this test produces an error, as shown:

5. If we scroll up further, we can see details of the failure:



6. The error states we expected 1 to equal 2. This is what was expected as the first assertion in our code shows the following:

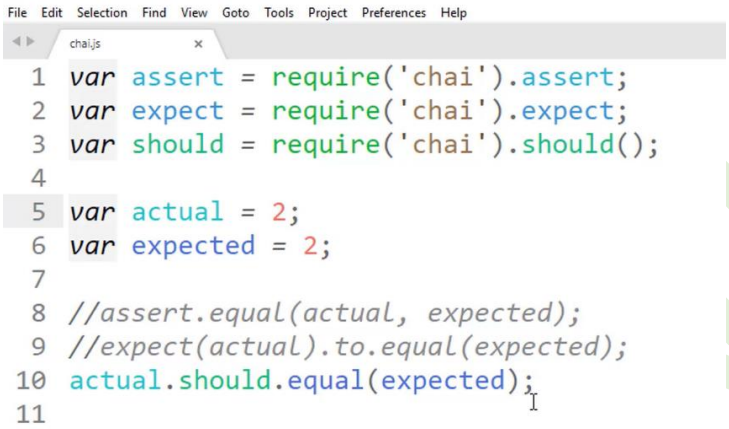
```
var actual = 1;
var expected = 2;

assert.equal(actual, expected);
```

7. The value 1 does not equal 2, hence the error has been produced. For this test to pass, we would have to assign the actual value a value of 2 or the expected value a value of 1 (as the actual value should match the expected value)

8. Note that for this test, the further assertions (expect and should) would not execute since the failure was reported on the first assertion

In the video, I then comment out the first assertion so that we can confirm the 2nd and 3rd assertion also works. I will not repeat this here as the result will be the same but please refer to the lecture video if you want to see this again

9. I then change the actual variable value to 2 and uncomment one of the assertions to test the change
    a. Your code should look like this:



    b. Save the file and go back to GitBash/ iTerm2
10. Now let's run the test again using the following command:
    a. npm test -- -- spec=tests/chai.js + press enter
    b. You should now see no failures are reported!

Lecture 32 – Chai - Adding Assertions to our Tests

In this lecture we start adding Chai Assertions to our webdriverUniversityTests.js file by adding Chai assertions using the knowledge we have picked up from the previous two lectures.

Lecture Resources on Udemy:

- https://www.chaijs.com/

Key Points:

- Now that we know how to use Chai assertions, we start to implement them into our tests
- Chai assertions will make it far easier to add intelligence into our tests

Instructions:

Amending our webdriverUniversityTest.js file to use Chai assertions

1. Use windows explorer (for windows) or Finder (on Mac) to open the webdrivenioFramework directory and move into the "tests" folder directory
2. We no longer need the chai.js test file, so delete it
3. Then, open up the webdriverUniversityTest.js file using sublime test
4. At the top of our file, we currently use the following:

```
1 var assert = require('assert');
```

5. As we are now going to use Chai assertions instead of the Node Assert class, we can change the above code to:

```
1 var expect = require('chai').expect;
```

6. Then, on line two, add the following line:

```
2 var should = require('chai').should();
```

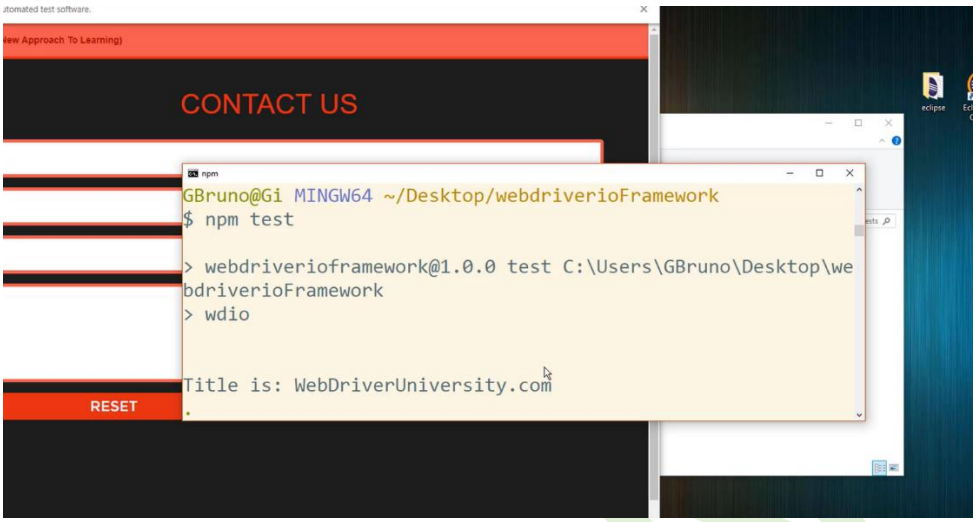
   a. We are simply adding the *expect* and *should* Chai libraries to our test

7. Now we are ready to amend our test to make use of the Chai assertions

```
describe("Verify whether webdriveruniversity links on homepage work correctly", function() {
    it("check that the contact us button opens the contact us page", function(done) {
        browser.setViewportSize({
            width: 1200,
            height: 800
        })
        browser.url('/');
        var title = browser.getTitle();
        assert.equal(title, 'WebDriverUniversity.com');          // Remove
        expect(title).to.equal('WebDriverUniversity.com');       // Add
        console.log('Title is: ' + title);
        browser.click("#contact-us");
        browser.pause(3000);
    });

    it("check that the login button opens the login portal page", function(done) {
        browser.url('/');
        browser.click('#login-portal');
        var title = browser.getTitle();
        assert.equal(title, 'WebDriverUniversity.com');          // Remove
        title.should.equal('WebDriverUniversity.com');           // Add
        console.log('Title is: ' + title);
        browser.pause(3000);
    });
});
```

8. I have highlighted the new lines of code that we have to add and remove

9. Once the above changes have been made, save the file and go back to GitBash / iTerm2
10. Make sure to be in the webdriverioFramework directory
11. Then type the following command:
    a. **npm test** + enter



12. You should see the test starts running (as shown above)

13. And if you look at the console, you should see that two tests have now passed ☺

This is how easy it is to add Chai assertions into our tests. During these tests, we checked to see both webpage titles returned from each test equalled “www.WebDriverUniversity.com”. We used both the Chai **expect** and Chai **should** styles.

Lecture 33 – Chai - Centralizing Assertions Using WDIO File

In this lecture we look at refactoring and centralizing our assertions using the WDIO file.

Lecture Resources on Udemy:

- https://www.chaijs.com/

Key Points:

- It's important to think about good code practise when writing tests
- Our tests should avoid repeated code, and, in this lecture, we look at refactoring our code so that it improves maintainability going forward
- We can do this by making use of our WDIO.conf file

Instructions:

1. Use windows explorer (for windows) or Finder (on Mac) to open the webdriverioFramework directory, then open the tests folder
2. Locate the webdriverUniversityTests.js file and open it using Sublime Text

Take the following scenario:

Say we have 50 different test files that we use to test various websites. Each one of our files may use the *expect* and *should* variables, as shown:

```
var expect = require('chai').expect;
var should = require('chai').should();
```

That means we would need to write this code in each and every file. It also means if we make changes to this code, we would need to amend each file! This is bad practise. Instead we should store the above (common) code somewhere and link to it. This is what we are going to do now.

3. Now also open the WDIO.conf file in Sublime Text
4. Scroll down the file and find the *before* hook which can be located at around line 171:

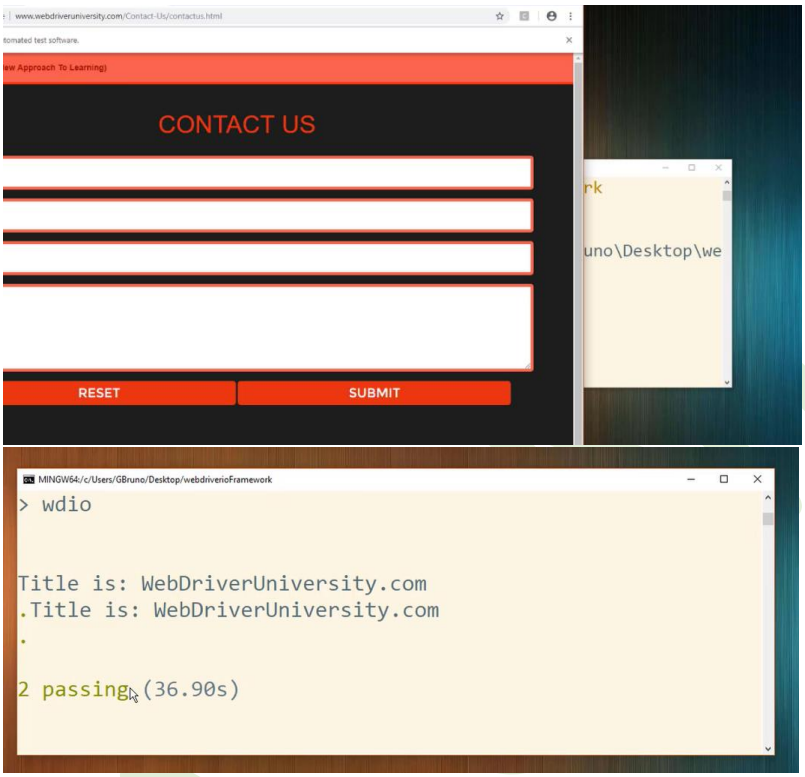
```
170      */
171     // before: function (capabilities, specs) {
172     // },
173     /**
174      * Runs before a WebdriverIO command gets executed.
```

5. Uncomment this code by removing the backslashes then go back to the webdriverUniversityTests.js file and **cut** the first to lines of code to your clipboard
6. Next, past the code into the WDIO file as shown:

```
170      */
171     before: function (capabilities, specs) {
172         expect = require('chai').expect;
173         should = require('chai').should();
174     },
175     /**
```

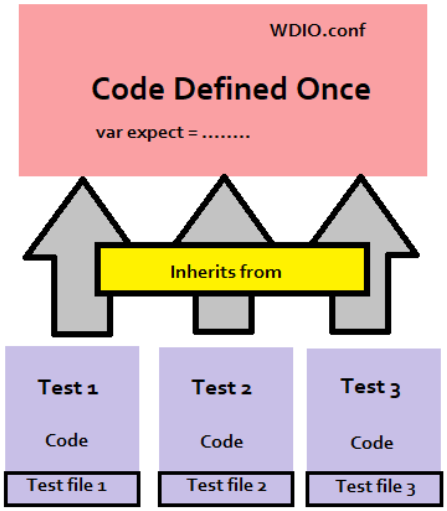
   a. Ensure the remove the "var" code just before the *expect* and *should* words, as this is no longer needed
7. Go back to the webdriverUniversityTests.js file and ensure the first two lines are no longer present and then save the file
8. Then go back to the WDIO.conf file and save the changes that we've made to that

9. Now we are ready to test the changes. We can do this by:
    a. Going back to GitBash/ iTerm2 and move into the webdriverioFramework directory (if not already in it)
    b. Type the following command:
        i. **npm test** + press enter
        ii. This should execute the tests



10. As you can see, the tests passed as expected.

In this lecture, we removed repeatable code from our test file and placed the common code into our WDIO.conf file. This makes sense, as it makes our code more maintainable going forward. We no longer have to write the same code in any future test files, as we used the before hook in the WDIO file to inject the chai require code during the initiation stage.



Module 12 – Pause, Debug Mode & Selectors

Lecture 34 - Pause Command Part 1

The Pause command pauses execution for a specific amount of time. It is recommended to **not** use this command to wait for an element to show up. In order to avoid flaky test results, it is better to use commands like waitforExist or other waitFor commands (we cover this in a later module, 17).

Lecture Resources on Udemy:

- http://v4.webdriver.io/api/utility/pause.html
- http://www.webdriveruniversity.com/Ajax-Loader/index.html

Key Points:

- The Pause command can be used to stop the execution of a test for a specific (defined) amount of time
- It is not recommended to be used to wait for elements to appear during a test. For this we will use other commands that we go through in Module 17.
- In this lecture, we create a simple test using the AJAX element from http://www.webdriveruniversity.com/Ajax-Loader/index.html and will use the Pause command to wait for the AJAX loader to complete loading before attempting to press the button once visible

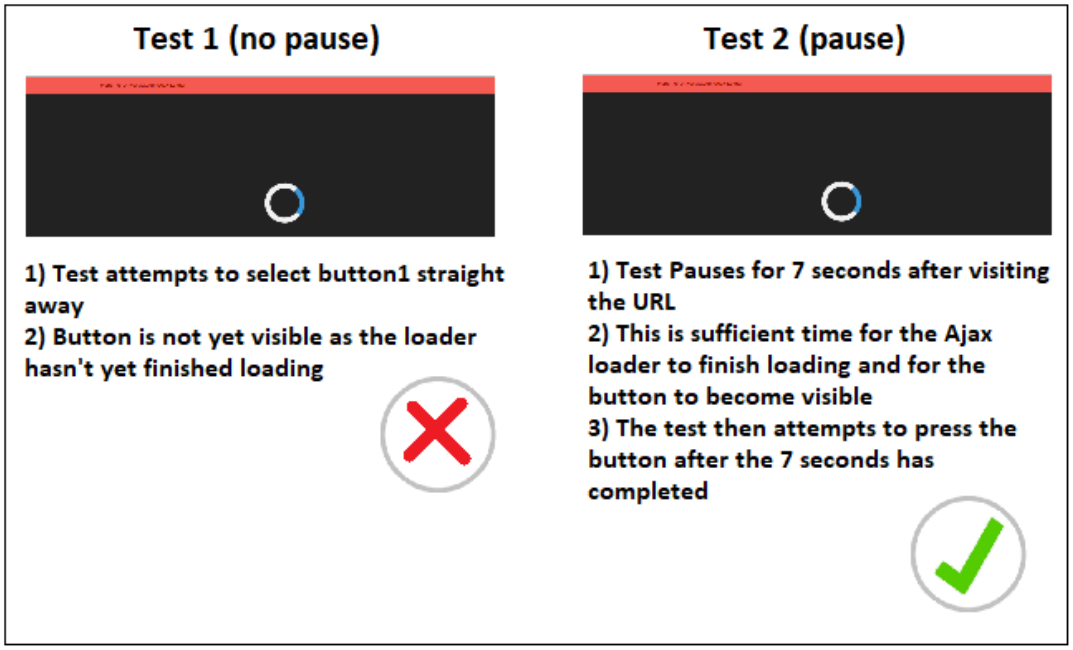
Instructions:

1. Use windows explorer (for windows) or Finder (on Mac) to open the webdriverioFramework directory, then open the tests folder
2. We need to create a new test file:
    a. Open Sublime Text
    b. Click Save As from the File menu button
    c. Change the path of where the file is to be saved to our "tests" folder e.g. it will be similar to C:\Users\Gianni\webdriverioFrameworks\tests
    d. Call the file "ajaxClickTest.js" and click save
    e. You should now see a new file in our "tests" folder based on the above
3. Ensure the file is still open in Sublime Test and type the following code (I explain what this code does below the image):

```
describe('Test that the button is clickable ince the Ajax loader completes loading', function() {
    it('Attempt to click the button asap', function(done) {
        browser.url('/Ajax-Loader/index.html');
        browser.click('#button1');
    });

    it('Attempt to click the button after 7 seconds', function(done) {
        browser.url('/Ajax-Loader/index.html');
        this.timeout(20000);
        browser.pause(7000);
        browser.click('#button1');
        browser.pause(7000);
    });
});
```

4. Here we are using Mocha "describe" and "it" to declare what we intend to test and what we expect each test to do.

5. The first “it” test tries to click the button straight away. We wouldn’t expect this to work because the Ajax loading needs to complete loading before the button becomes visible
6. The second “it” test does something similar, but this time it uses a Pause. The Pause stops the test continuing until the declared pause time has been reached. This allows the Ajax loader to finish loading before we move onto trying to select the button.
    a. The Pause value is in milliseconds. 7000 equates to 7 seconds.
7. To make it easier to understand, here is a diagram explaining what we are trying to do:



Lecture 35 - Pause Command Part 2

This lecture continues from the previous lecture. Here we recap on what we have done in Part 1 and attempt to execute the test to see the results.

Lecture Resources on Udemy:

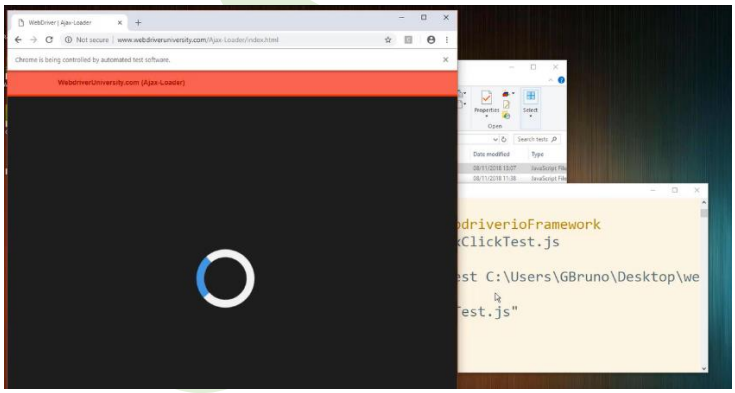
- http://v4.webdriver.io/api/utility/pause.html
- http://www.webdriveruniversity.com/Ajax-Loader/index.html

Key Points:

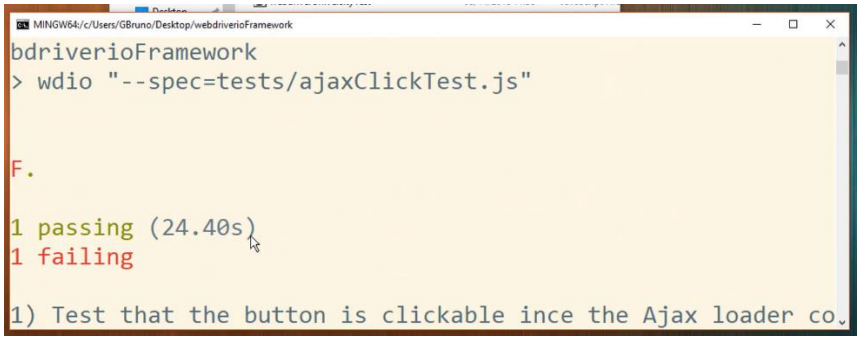
- We have created a new test called ajaxClickTest.js
- We have used Mocha to declare a "description" of the test and included two "it" test cases
- The second "it" test uses the Pause command to instruct the second test to wait for the Ajax loader to finish loading (so the button appears) before going ahead and clicking the button
- We expect test 1 to fail (due to the button not being visible right away) and we expect test 2 to pass (due to there being enough time for the button to appear using the Pause)

Instructions:

1. Now that we have written the code, we need to save the changes
    a. Click the save button and then close Sublime Text
2. Open up GitBash / iTerm2 and navigate to our webdriverioFramework directory
3. Remember! Because we have two tests in our tests folder, we now have to instruct which test to execute. We do this by using the following command:
    a. **npm test -- -- spec=tests/ajaxClickTest.js** + press enter
4. This will execute the test as shown:



5. Once the test has completed, lets review the console output:



6. As you can see, one test has failed (the one without a Pause) and the second has passed (one with the Pause – as the button was visible and selectable at the time)

Lecture 36 - Debug Mode

In this lecture, we take a look at debug mode. Debug mode allows you to debug any issues that you may have during runtime. Check out the debug documentation using the link below for further reading.

Lecture Resources on Udemy:

- http://v4.webdriver.io/api/utility/debug.html#Usage

Key Points:

- This command helps you to debug your integration tests
- It stops the running browser and gives you time to jump into it and check the state of your application (e.g. using the dev tools)
- Your terminal transforms into a REPL interface that will allow you to try out certain commands, find elements and test actions on them

Instructions:

Note – In this section we amend the default WDIO file "waitforTimeout" setting. WebdriverIO provides multiple commands to wait on elements to reach a certain state (e.g. enabled, visible, existing). These commands take a selector argument and a timeout number which declares how long the instance should wait for that element to reach the state. The waitforTimeout option allows you to set the global timeout for all waitFor commands so you don't need to set the same timeout over and over again.

How to add debug mode to our tests

1. First, we need to amend our WDIO.conf file
    a. Use windows explorer (for windows) or Finder (on Mac) to open the webdriverioFramework directory and open the WDIO.conf file using Sublime Text
        i. We need to amend the current "waitforTimeout" setting which is currently set to 10000 milliseconds (10 seconds) to 99999999 (99999 seconds) when using Debug mode
        ii. This is so we have enough time to assess/ review the debug prompt before continuing during a test
        iii. We do not amend the default waitforTimeout directly. We instead use a variable to look for when we use debug mode and then change the timeout value when true
    b. Add the following line of code on line 9, as shown:

```
9    var timeout = process.env.DEBUG ? 99999999 : 10000;
```

        i. This instructs our test to use a longer wait time (9999999) when in debug mode
        ii. process.env.Debug is an environment variable that indicates whether we are in Debug mode or not
        iii. The **?** (aka ternary operator/ conditional operator), is used as shorthand for an if...else statement in JavaScript
        iv. If we are in Debug mode, then use a timeout of 99999999 otherwise use the default timeout value of 10000

c. Finally, we need to make one more change to our WDIO file and that's to amend the Mocha options section as shown:

i. **Was**:

```
138     // Options to be passed to Mocha.
139     // See the full list at http://mochajs.org/
140     mochaOpts: {
141         ui: 'bdd',
142
143     },
```

ii. **Now**:

```
138     // Options to be passed to Mocha.
139     // See the full list at http://mochajs.org/
140     mochaOpts: {
141         ui: 'bdd',
142         timeout: timeout
143     },
```

iii. This change passes our options to Mocha so that Mocha is aware of our timeout value during runtime

iv. Once the above changes have been made, save the WDIO.conf file

2. Go back to windows explorer (for windows) or Finder (on Mac) to open the webdriverioFramework directory and open the webdriverUniversityTest.js file using Sublime Text

a. We will now amend our test to make use of Debug mode

b. The webdriverio documentation gives us the following code example

Example

debug.js

```
1  it('should demonstrate the debug command', function () {
2      browser.setValue('#input', 'FOO')
3
4      browser.debug() // jumping into the browser and change value of #input to 'BAR'
5
6      var value = browser.getValue('#input')
7      console.log(value) // outputs: "BAR"
8  })
```

c. It shows us that we need to use the following code to use debug mode:

i. **browser.debug()**

3. Amend our test to include the following:

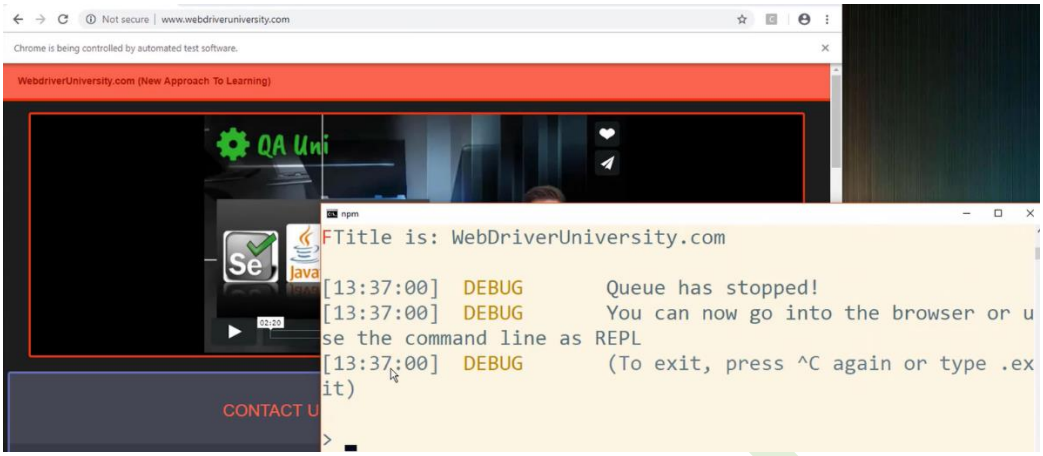
```
1  describe("Verify whether webdriveruniversity links on homepage work correctly", function() {
2      it("check that the contact us button opens the contact us page", function(done) {
3          browser.setViewportSize({
4              width: 1200,
5              height: 800
6          })
7          browser.url('/');
8          var title = browser.getTitle();
9          expect(title).to.equal('WebDriverUniversity.com');
10         console.log('Title is: ' + title);
11         browser.debug();
12         browser.click("#contact-us");
13         browser.pause(3000);
14     });
```

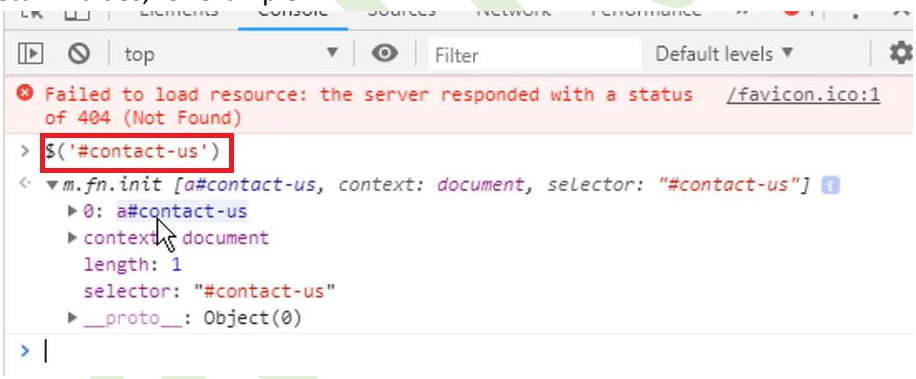
a. This will instruct our first "it" test to use debug when the test is executed

4. Once you have added the code above, save the file and close it
5. Open up GitBash / iTerm2 and navigate to our webdriverioFramework directory
6. Type in the following command:

a. **DEBUG=true npm test -- --specs=webdriverUniversityTest.js** + press enter

7. This will execute the test and you should see the following:



8. The command line is showing that Debug most is being used and the test has paused at this stage
9. This is great to review and debug code whilst in the middle of a test because now we can:
    a. Press F12 (developer mode on Chrome) on the browser window that's currently on webdriveruniversity.com
    b. We can then inspect elements before proceeding or use JavaScript using the console tab to return values, for example:
        i. 

        ii. Here we are using JavaScript to search for data relating to the element that has an ID of #contact-us
10. We can then end debug mode by following the instructions on GitBash/ iTerm2. For windows, you simply press **CTRL + C** or type **.exit**
    a. The test will then continue on

Lecture 37 - Creating Selectors using Ranorex

In this lecture, we take a look at creating selectors. Selectors are used to be able to select elements on a website. An example would be to select a particular button on a webpage. We need to instruct our tests to use the button and we do this by referring to the locator associated to the button.

There is a great Chrome addon that makes creating selectors so much easier. It's called Ranorex Selocity and it can be used to automatically generate selectors.

Lecture Resources on Udemy:

- https://chrome.google.com/webstore/detail/ranorex-selocity/ocgghcnnjekfpbmafindjmijdpopafoe?hl=en

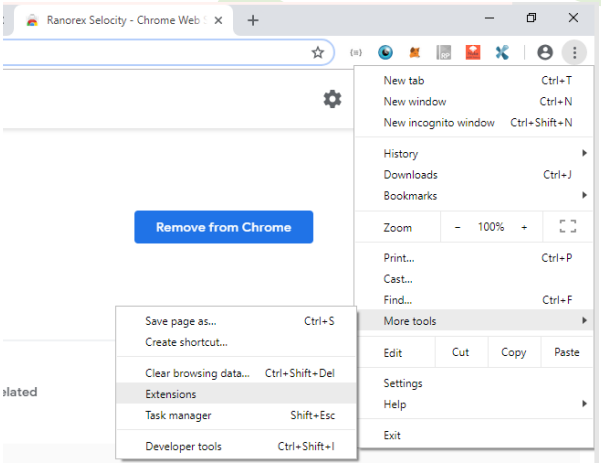
Key Points:

- Selectors are references/ patters that we can use to select a particular element on a website
- Those familiar with html/CSS would have used classes and ids. These are examples of selectors.
- There is a chrome extension called Ranorex Selocity that automatically generates selectors for us on a webpage
- Once a selector has been defined, we can then instruct our tests to use them

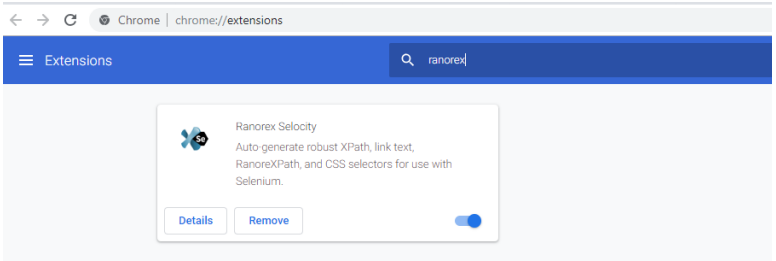
Instructions

Installing the Ranorex Selocity addon to Chrome:

1. Open up an instance of Chrome
2. Select settings > More Tools > Extensions:

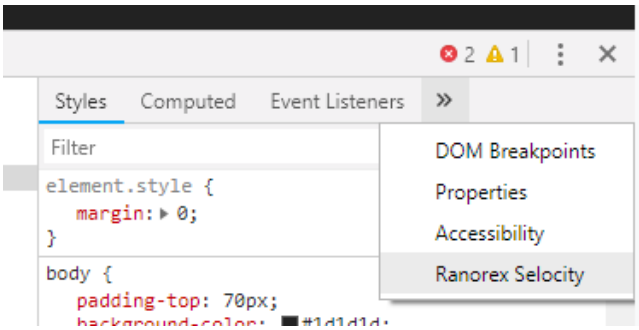


3. Search for Ranorex Selocity and install the extension:

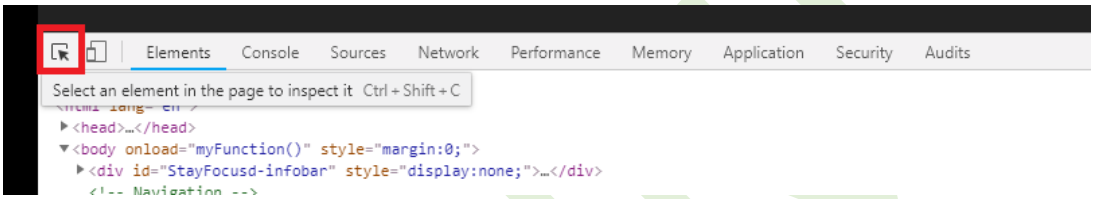


How to use Ranorex Selocity:

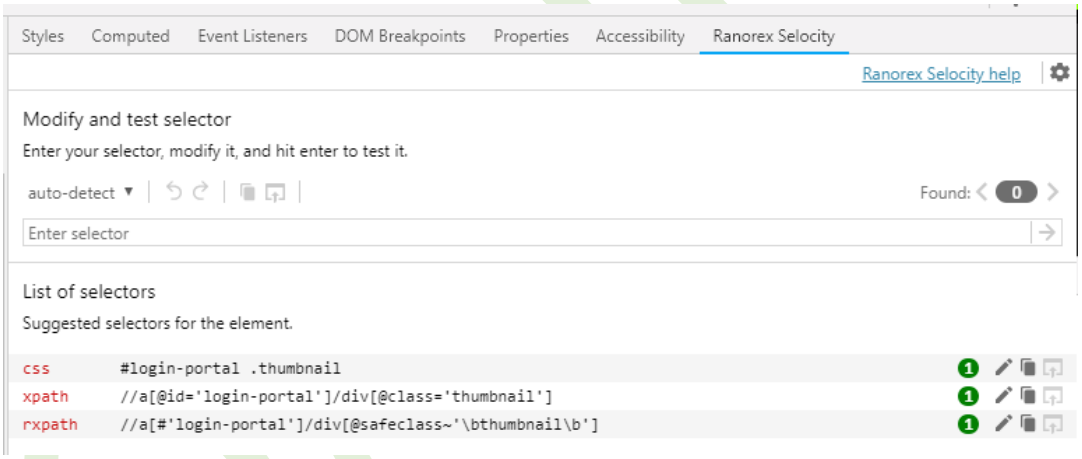
1. Open up a webpage (e.g. www.webdriveruniversity.com)
2. Switch over to developer mode (press F12) using Chrome
3. There will be a Ranorex option as shown:



4. Once the Ranorex window is open, press the Chrome Inspector Selector button:



5. Select an element (such as the login portal section) and the following data is returned:



6. This provides locators that we can use to select the element

Please refer to the lecture video for a more detailed explanation on this section as there is too much information to list.

The video covers:

- How to search to see how many instances of an element are in use
- How we can create a locator using Ranorex
- Explanation of Ranorex commands (e.g. how to search a particular section of the Document Object Model for an element)

Module 13 – Targeting & Skipping Specific Tests

Lecture 38 - Targeting Specific Tests

In this lecture, we looked at how we could target specific tests. The Mocha documentation provides examples of how we can run only the specified suite or test-case of our choosing. This is useful because currently when we run a test, it tests all cases within the test file. This is different from when we select which test file to use (as shown previously) as we are now focusing on a more granular level where we now choose the specific test to run within a single file when multiple tests exist.

Lecture Resources on Udemy:

- https://mochajs.org/#exclusive-tests

Key Points:

- We are able to target specific tests within a single .js file
- This is different from selecting a specific test file (as shown previously)
- It's common to have a number of tests within a single .js file and we can use Mocha's Exclusive Tests command to specific the exact test we want to run
- Mocha provides an Exclusive Tests command that makes this possible (see link above)

Instructions:

How to use Exclusive Tests to target a specific test in our ajaxClickTest.js file

1. Use windows explorer (for windows) or Finder (on Mac) to open the webdriverioFramework directory, move into the tests folder and open the ajaxClickTest.js file using Sublime Text
2. Our current file looks like this:

ajaxClickTest.js

```
describe('Test that the button is clickable ince the Ajax loader completes loading', function() {
    it('Attempt to click the button asap', function(done) {
        browser.url('/Ajax-Loader/index.html');
        browser.click('#button1');
    });

    it('Attempt to click the button after 7 seconds', function(done) {
        browser.url('/Ajax-Loader/index.html');
        this.timeout(20000);
        browser.pause(7000);
        browser.click('#button1');
        browser.pause(7000);
    });
});

```

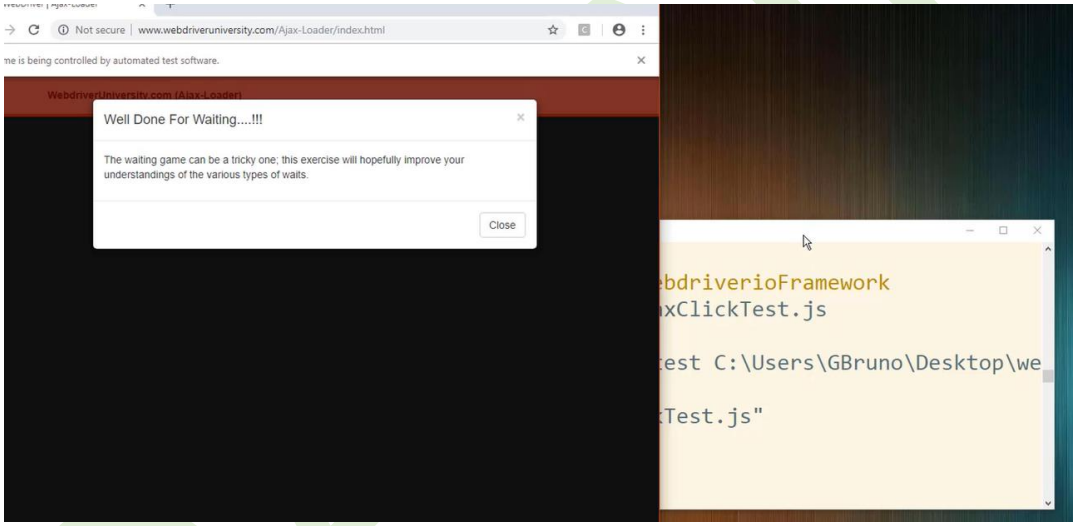
   a. So far, we know that the first "it" test fails (due to insufficient time for the button to appear) and the second "it" test passes (as we delayed the click using a Pause)

Continued Next Page

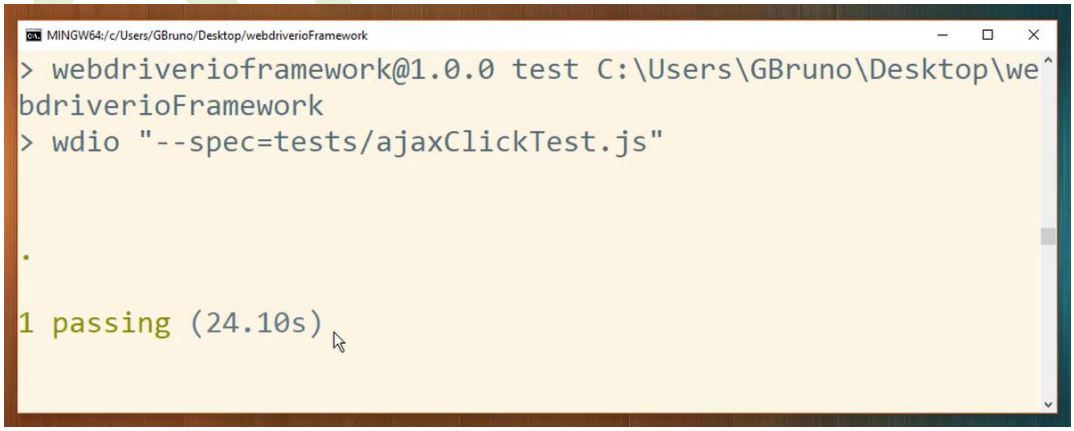
3. We can now add the following to the second “it” case so that our test only executes the second “it” case:

```
describe('Test that the button is clickable ince the Ajax loader completes loading', function() {
    it('Attempt to click the button asap', function(done) {
        browser.url('/Ajax-Loader/index.html');
        browser.click('#button1');
    });

    it.only('Attempt to click the button after 7 seconds', function(done) {
        browser.url('/Ajax-Loader/index.html');
        this.timeout(20000);
        browser.pause(7000);
        browser.click('#button1');
        browser.pause(7000);
    });
});
```

4. Once you have added .only before the second “it” statement, save your file
5. Open up GitBash / iTerm2 and navigate to our webdriverioFramework directory
6. Run the following command:
    a. **npm test -- -- specs=tests/ajaxClickTest.js** + press enter
    b. This should start the test (as shown):



7. Now let’s review the command line output:



8. As you can see, only one test output has been reported and as expected, it passed. This is because we limited our test to only run the second “it” test, as we specified the **.only** command in the .js file.

Lecture 39 - Skipping Specific Tests

In this lecture, we looked at how to skip specific tests. This is similar to the previous lecture but sometimes skipping a specific test is the better option when we have a number of tests we do want to run and only want to skip one (or a few) tests.

Lecture Resources on Udemy:

- https://mochajs.org/#exclusive-tests

Key Points:

- We can also skip specific tests using the Mocha Exclusive Tests command (see link above)
- This is useful for when we only want to exclude one or a few tests (saves adding .only to multiple tests when the majority of tests are to run)

Instructions:

How to use Exclusive Tests to skip a specific test in our ajaxClickTest.js file

1. Use windows explorer (for windows) or Finder (on Mac) to open the webdriverioFramework directory, move into the tests folder and open the ajaxClickTest.js file using Sublime Text
2. Our current file looks like this:

ajaxClickTest.js

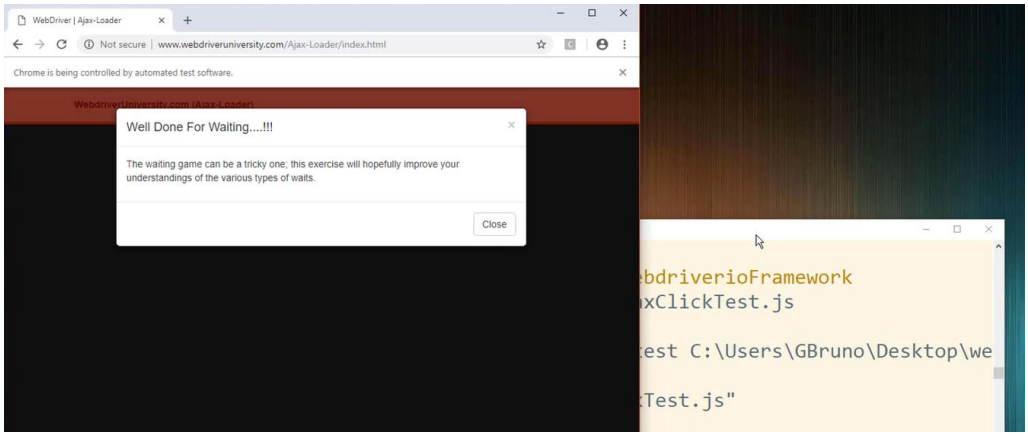
```
1  describe('Test that the button is clickable ince the Ajax loader completes loading', function() {
2      it('Attempt to click the button asap', function(done) {
3          browser.url('/Ajax-Loader/index.html');
4          browser.click('#button1');
5      });
6
7      it.only('Attempt to click the button after 7 seconds', function(done) {
8          browser.url('/Ajax-Loader/index.html');
9          this.timeout(20000);
10         browser.pause(7000);
11         browser.click('#button1');
12         browser.pause(7000);
13     });
14 });
15
```

3. Remove the .only code from line 7
4. Then add the following to line 2:

ajaxClickTest.js

```
1  describe('Test that the button is clickable ince the Ajax loader completes loading', function() {
2      it.skip('Attempt to click the button asap', function(done) {
3          browser.url('/Ajax-Loader/index.html');
4          browser.click('#button1');
5      });
6
7      it('Attempt to click the button after 7 seconds', function(done) {
8          browser.url('/Ajax-Loader/index.html');
9          this.timeout(20000);
10         browser.pause(7000);
11         browser.click('#button1');
12         browser.pause(7000);
13     });
14 });
15
```

5. Once you have made the change above, save the file
6. Open up GitBash / iTerm2 and navigate to our webdriverioFramework directory
7. Run the following command:
    a. **npm test -- -- specs=tests/ajaxClickTest.js** + press enter
    b. This should start the test (as shown):



8. And if we review the command line output, we should see:

9. The output confirms we have passed one test and skipped one test. This is as expected, because we declared a .skip command to the first “it” test case and so only wanted the second “it” case to run.

Module 14 – CSS Extraction

In this module we took a look as CSS extraction. CSS extraction is when we return CSS information of an element from a HTML page. An example of this is to find out the styling associated to a H1 element for example.

Lecture 40 - getCssProperty Command

In this lecture, we looked at the webdriverio getCssProperty. This is a property that we can use to target the CSS styling of an element that will then return CSS data associated to it.

Lecture Resources on Udemy:

- http://v4.webdriver.io/api/property/getCssProperty.html
- http://www.webdriveruniversity.com/

Key Points:

- This getCssProperty gets a CSS property from a DOM-element selected by given selector
- The return value is formatted to be testable
- Colours gets parsed via rgb2hex and all other properties get parsed via CSS-value
- We use this property in the following way:
    - browser.getCssProperty(selector,cssProperty);
        - Where the first argument (*selector*) is the element selector; and
        - The second argument (*cssProperty*) is the CSS property that we want to return (e.g. height or color)

Instructions:

In this lecture, we used www.webdriveruniversity.com to review the .css properties of the homepage. We did this by using the Chrome inspector tool (press F12 on the website) and on the right-hand side you will see CSS properties and values once an element has been selected using the Inspector "selector" function (found on the top left of the inspector panel and hovering/selecting an element using the cursor). In this test we are going to return the height and width (which are .CSS properties) of the homepage carousel.

1. Open up Sublime Text and create a new file. Before proceeding, save the file by giving it a name of cssExtractTest.js and save it into our webdriverFramework/tests folder
2. Then, reopen the file and write the code shown in the following diagram; the code is explained below:
    a. We added a **describe** keyword and included a description of the test we wish to perform – "Test the webdriveruniversity homepage"
    b. We then added **it** and provided a description of "Output the height of the homepage carousel"
    c. We access the webdriveruniversity homepage by adding: **browser.url('./')**, remember that because we have defined the baseUrl in our WDIO file, we don't need to reference the full URL
    d. We then added a pause of 2 seconds using: **browser.pause(2000);**
    e. We then created a variable that will store the CSS value by adding the following: **var divCarouselHeight = browser.getCssProperty('udemy-promo-thumbnail', 'height');**

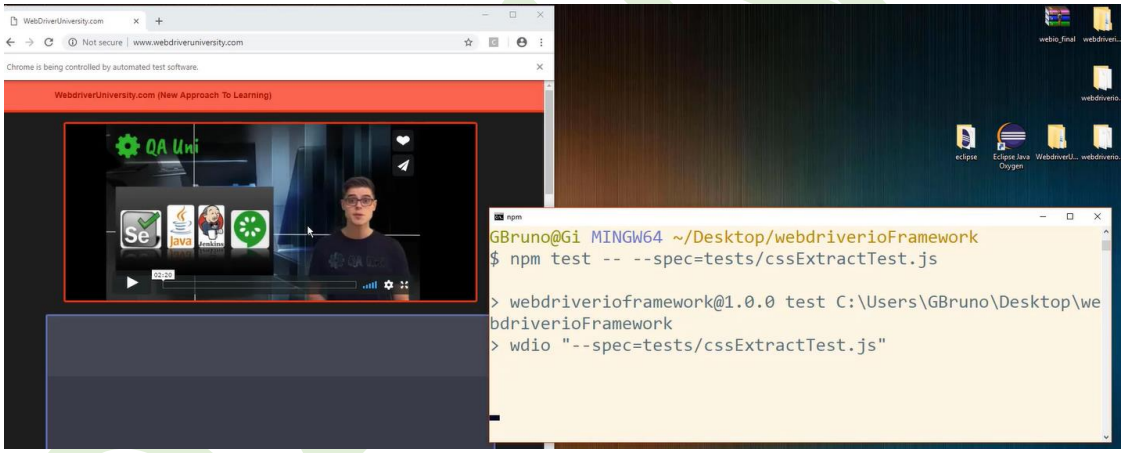
i. Here we have set the first argument to udemy-promo-thumbnail which is the ID selector of the carousel from the webdriveruniversity website
ii. We then provide a second argument 'height' where we are looking to return the height .CSS property value

f. We then add the following line: **console.log(divCarouselHeight);** which returns the value of the divCarouselHeight value to the console once the .CSS value has been extracted

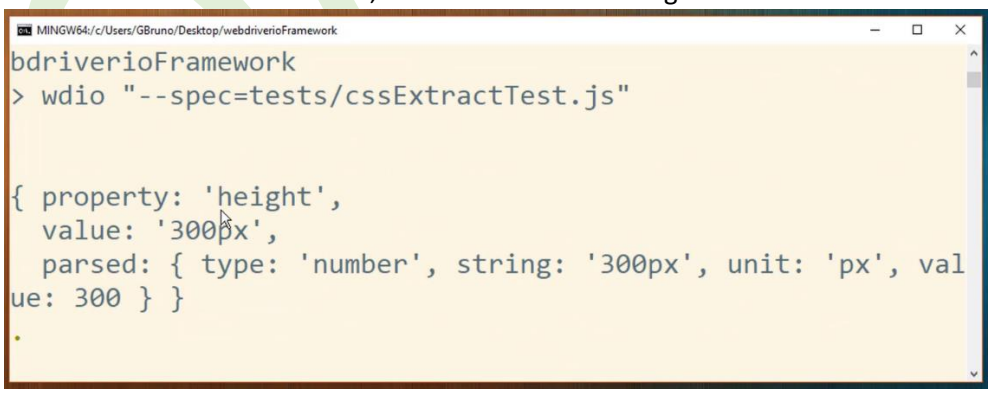
3. Your code should look like this at this point:

```
ajaxClickTest.js    cssExtractTest.js
1  describe('Test the webdriveruniversity homepage', function() {
2      it('Output the height of the homepage carousel', function() {
3          browser.url('./');
4          browser.pause(2000);
5          var divCarouselHeight = browser.getCssProperty('#udemy-promo-thumbnail', 'height');
6          console.log(divCarouselHeight);
7      });
8  });
```

4. Open up GitBash / iTerm2 and navigate to our webdriverioFramework directory
5. Run the following command:
6. **npm test -- -- specs=tests/cssExtractTest.js** + press enter
7. You should see the test run, as shown below:



8. Then if we refer to the console, we will see the following:



9. As you can see, the CSS data has been outputted for the Carousel element that has the .udemy-promo-thumbnail id ☺

Module 15 – Mocha Hooks

In this module we took a look at Hooks, which are blocks of code that run before or after each scenario. You can define them anywhere in your project or step definition layers, using the methods Before, After, beforeEach and afterEach.

Mocha Hooks allows us to better manage the code workflow and helps us to reduce the code redundancy. We can say that it is an unseen step, which allows us to perform our scenarios or tests.

Lecture 41 - Mocha Hooks Part 1

In this lecture, you are introduced to Mocha Hooks. The Mocha documentation provides us with some useful information and demonstrates how we can use them to set preconditions and clean up our tests after they've run.

Lecture Resources on Udemy:

- https://mochajs.org/#hooks

Key Points:

- Mocha provides four different hook methods:
  - **before** – which runs before all tests in the code block
  - **after** – which runs after all tests in the code block
  - **beforeEach** – which runs before each test in the code block
  - **afterEach** – which runs after each test in the code block
- This is useful for when we need to test certain things at different times during a test
- An example would be a contact us form as there are a number of different combinations to test (e.g. one test might not attempt entering a mandatory email address, whilst another might not include a surname). We can use hooks to handle these different combinations (for example, after a form submission)

Instructions:

Adding Mocha hooks to a new test file

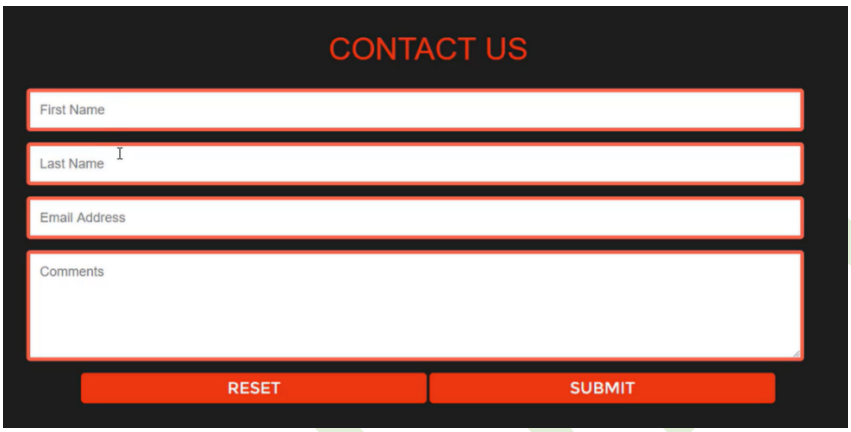
1. To save you having to write out all the code by hand, I have provided you with a skeleton file of the code used during this lecture. Download it from the course resources section on Udemy and we then add to it
2. Download the file locally and open it in Sublime Text
3. Save the file by giving it a name of "contactUsTest.js" and save it to our tests folder (webdriveruniversity/tests)
4. Next, review the file and notice the following:
    a. We have added four "it" blocks that sit inside a describe block
        i. We are testing four different scenarios;
            1. Should be able to submit a successful submission via contact us form
            2. Should not be able to submit a successful submission via contact us form as all fields are required
            3. Should not be able to submit a successful submission via contact us form as all fields are required
            4. Should not be able to submit a successful submission via contact us form as all fields are required

5. We also have a forEach (code line 1) defined at the top of the file. This is a function that is to take place before each test begins.

```
beforeEach(function() {
    browser.url('/Contact-Us/contactus.html');
})
```

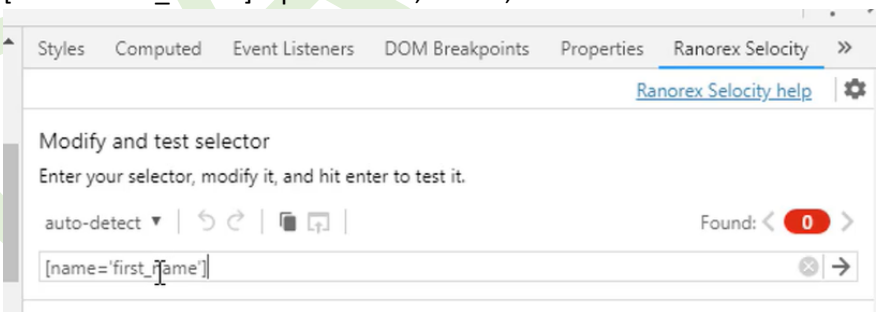
   a. This instructs our test to use a browser URL that points to the webdriveruniversity contact us form. Remember we don't have to define the full URL path (e.g. www.webdriveruniversity.com/) because we have already defined this as our base URL in our WDIO.conf file

6. Now we need to add locators into our test so that we can pass values that will be used as input for the contact us form



   a. As you can see, there are four fields and two buttons
   b. We need to instruct our test to add data to these fields before submitting

7. Open up the Chrome inspector (F12) on the webdriveruniversity website contact us page and select the First Name field using the selector tool
8. Then, use Ranorex Selocity (with the First Name field selected) and create a CSS selector by entering the following:
   a. [name= 'first_name'] + press enter, like so;



   b. You should then see a dotted line appear around the element affected by this change:

   c. Go back to Sublime Text and add the following code inside our first "it" block, as shown:

```
8   it('Should be able to submit a successful submission via contact us form', function(done) {
9     [name='first_name']
10    });
```

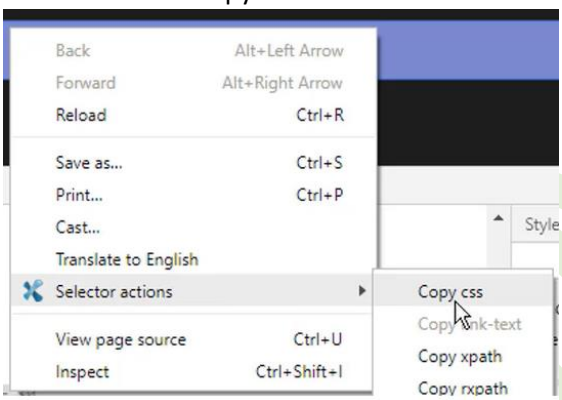
9. We repeat this process for the last name, email address and comments fields, use the names:

a. **[name='last_name']** for the last name field
b. **[name='email']** for the email field
c. **[name='message']** for the comment field

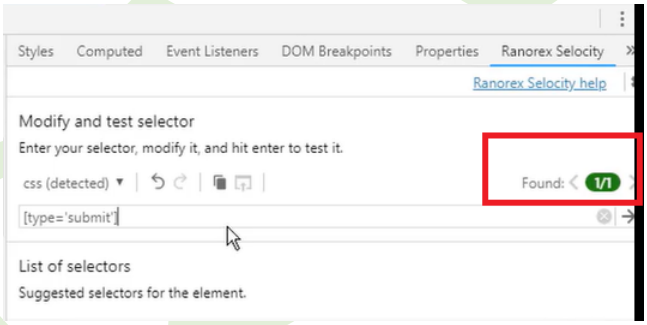
10. Copy the selector fields that were created above and add them to your .js test file, as shown:

```
8   it('Should be able to submit a successful submission via contact us form', function(done) {
9     [name='first_name']
10    [name='last_name']
11    [name='email']
12    [name='message']
13    });
```

11. Now finally, we need to create a locator for the submit button. I do this by showing you an alternative way to get the locator name:
    a. Right click on the submit button on the contact us form
    b. Select actions > Copy css



    c. This will copy the css selector for the submit button to your clipboard (you can check to see if the css selector is unique by reviewing the matches found on the Ranorex panel, see below):



12. Now go back to Sublime Text and paste from your clipboard into your .js file like shown:

```
it('Should be able to submit a successful submission via contact us form', function(done) {
  [name='first_name']
  [name='last_name']
  [name='email']
  [name='message']
  [type='submit']
  });
```

We continue with Mocha hooks in the next lecture

Lecture 42 - Mocha Hooks Part 2

In this lecture we continue on from Part 1 and delve further into creating tests using Mocha hooks. We've now already created the initial test skeleton and have included locators into our first test so that we can locate the fields of interest from the contact us form.

Lecture Resources on Udemy:

- http://v4.webdriver.io/api/action/setValue.html

Key Points:

- We now need to be able to interact with the webpage itself
- This means we need to be able to pass values to the contact us page and we can do this using the Webdriverio setValue method
- setValue sends a sequence of key strokes to an element (clears value before).
    - You can also use unicode characters like Left arrow or Back space. WebdriverIO will take care of translating them into unicode characters.
- We also need to be able to click the submit button and we do this using the webdriverio click command

Instructions:

How to use setValue and how we can add it into our tests

1. If not already open, open the contactUsTest.js file using Sublime Text
2. We then need to amend our first "it" test by replacing the existing code with:

```
it('Should be able to submit a successful submission via contact us form', function(done) {
  browser.setValue("[name='first_name']",'Joe');
  browser.setValue("[name='last_name']",'Blogs');
  browser.setValue("[name='email']", 'joe_blogs@mail.com');
  browser.setValue("[name='message']", 'How much does product x cost?');
  browser.click("[type='submit']");
  });
```

    a. Here we have used browser.setValue and provided the locator as the first argument (i.e. '[name]='first_name') and the value in the second argument (i.e. 'Joe')
    b. We did this for the next three contact us fields so that the last_name, email and message are parsed values
    c. Lastly, we used browser.click and provided the submit button locator as the argument. This instructs our test to click the button once the above lines have processed
3. Remember that because we are testing a successful test case in this "it" statement, we are providing values for each field in order for a successful field submission to take place. We then build different test scenarios for the other "it" tests.
4. Next, we move onto the next "it" statement where we are attempting to test the following scenario:
    a. "Should not be able to submit a successful submission via the contact us form as all fields are required"
    b. Remember, the all fields on this form are mandatory and so we would expect a failed submission is all fields are NOT provided
5. We amended the **second** "it" statement to contain the following:

```
it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
  browser.setValue("[name='first_name']",'Mike');
  browser.setValue("[name='last_name']",'Woods');
  browser.setValue("[name='email']", 'mike_woods@mail.com');
  browser.click("[type='submit']");
  });
```

   a. This is the same code as the first "it" test, BUT we have not included the **comment** line and so would expect the submission to fail

6. We then moved onto the **third** "it" statement where we are again attempting to test another scenario. This time we want to parse only two fields and would expect the submission to fail:

```
it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
  browser.setValue("[name='first_name']",'Sarah');
  browser.setValue("[name='email']", 'sarah_woods@mail.com');
  browser.click("[type='submit']");
  });
```

   a. Notice that we have not provided the last_name or the comment fields in this test
   b. The form should not submit successfully due to all fields being mandatory

7. Finally, we amend the **last** "it" statement to contain the following:

```
it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
  browser.setValue("[name='first_name']",'Jim');
  browser.setValue("[name='last_name']",'Jomes');
  browser.click("[type='submit']");
  });
```

   a. Notice that we do not provide an email address or comment field in this test
   b. The form should not submit successfully due to all fields being mandatory

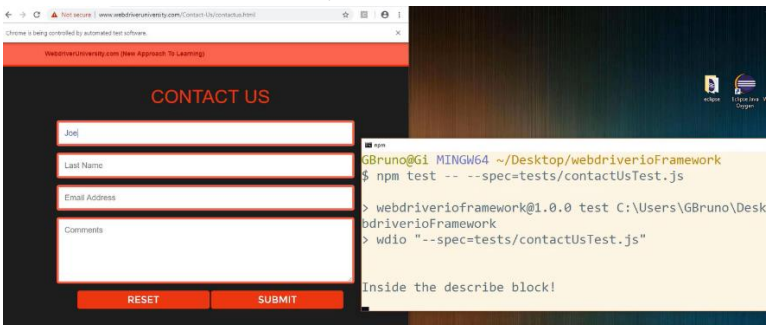
Just to recap, we are testing a number of contact us form combinations. We have one positive test (the first "it" statement) that should allow us to submit the form successfully and we then have three negative tests (the second, third and final "it" statements) that should not allow a successful form submission because all mandatory fields are not provided.

8. Our last change was to add the following block of code into our test (just after the describe block):

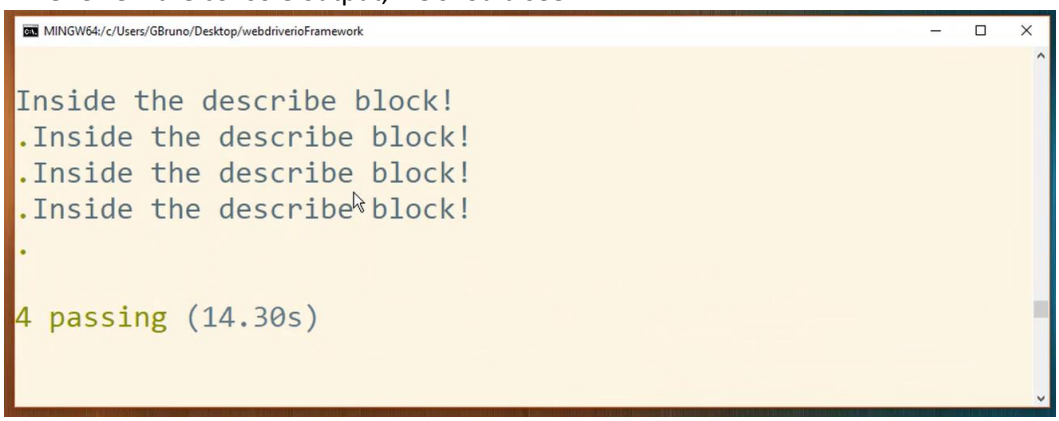
```
1  beforeEach(function() {
2      browser.url('/Contact-Us/contactus.html');
3  })
4
5  describe('Test Contact Us form WebdriverUni', function() {
6      beforeEach(function() {
7          console.log('Inside the describe block!');
8      })
9
10   it('Should be able to submit a successful submission via contact us form', function(done) {
```

   a. This will report to the console when our test is in the describe block (just to indicate where our test is)

9. Once you have made the changes above, save your file and close Sublime Test
10. Open up GitBash / iTerm2 and navigate to our webdriverioFramework directory
11. Run the following command:
12. **npm test -- -- specs=tests/contactUsTest.js** + press enter
13. You should see the test run, as shown below:



14. If we review the console output, we should see:



15. As you can see, all four tests have passed! This is what we are expecting because of the following:
    a. Test 1 (the first “it”) we were testing a **successful** submission as al fields have been provided
    b. Test 2 (the second “it”) we were testing an **unsuccessful** submission because we were missing the comment field
    c. Test 3 (the third “it”) we were testing an **unsuccessful** submission because we were missing the last_name and comment fields
    d. Test 4 (the fourth “it”) we were testing an **unsuccessful** submission because we were missing the email field
    e. Also note that the “Inside the describe block!” wording has been outputted four times. This is because we used beforeEach which initiates before each test begins.

Module 16 – Handling Browser Window Tabs

In this module we took a look at handling browser window tabs. It common to select a link of a webpage that opens up an additional tab. Our tests need to be able to handle this situation.

Lecture 43 - Tabs Part 1

In this lecture we look at how we can handle multiple browser tabs using the webdriverio API. The specific command we are interested in is the switchTab command.

Lecture Resources on Udemy:

- http://v4.webdriver.io/api/window/switchTab.html#Usage

Key Points:

- Sometimes we need to handle multiple tabs when performing our tests
- Webdriverio api documentation has a useful command called switchTab
- We can use this command to switch focus to a particular tab/ window handle

Instructions:

Improving our webdriverUniversityTest.js file

1. Use windows explorer (for windows) or Finder (on Mac) to open the webdriverioFramework directory, move into the tests folder and open the webdriverUniversityTest.js file using Sublime Text
2. Our current test looks like the following (part highlighted in Red explained below)

```
webdriverUniversityTest.js
1  describe("Verify whether webdriveruniversity links on homepage work correc
2      it("check that the contact us button opens the contact us page", funct
3          browser.setViewportSize({
4              width: 1200,
5              height: 800
6          })
7          browser.url('/');
8          var title = browser.getTitle();
9          expect(title).to.equal('WebDriverUniversity.com');
10         console.log('Title is: ' + title);
11         browser.debug();
12         browser.click("#contact-us");
13         browser.pause(3000);
14     });
15
16     it("check that the login button opens the login portal page", function
17         browser.url('/');
18         browser.click('#login-portal');
```

3. For the test highlighted, we are:
    a. Navigating our test to the webdriveruniversity.com homepage
    b. Setting a variable that stores the webpage title
    c. We perform a test where we expect the title to be returned to that defined in the code
    d. We log the title to the console
    e. We have set a line to use debug mode
    f. We then click the contact-us link which opens an additional tab to the contact us form
    g. We then pause the test for three seconds

4. The remainder of this tests looks like this (part highlighted in Red explained below):

```
15
16      it("check that the login button opens the login portal page", function
17          browser.url('/');
18          browser.click('#login-portal');
19          var title = browser.getTitle();
20          title.should.equal('WebDriverUniversity.com');
21          console.log('Title is: ' + title);
22          browser.pause(3000);
23      });
24  });
25
```

5. For the test highlighted above, we are:
    a. Checking to see if the login button opens the login portal page
    b. We navigate to the webdriveruniversity website
    c. Click on the login-portal button
    d. We define a variable that should get the page title
    e. We perform a test where we expect the title to be returned to that defined in the code
    f. We log the title to the console
    g. We then pause the test for three seconds
6. We then made the following changes to the code:
    a. We changed the location of the browser.click('#login-portal') from code line 18 to just before the browser.pause(3000) (on code line 22)
    b. We deleted the browser.debug() code from line 11 (as we no longer need it)
7. Your test at this stage should look like:

```
7           browser.url('/');
8           var title = browser.getTitle();
9           expect(title).to.equal('WebDriverUniversity.com');
10          console.log('Title is: ' + title);
11
12          browser.click("#contact-us");
13
14          browser.pause(3000);
15      });
16
17      it("check that the login button opens the login portal page", function
18          browser.url('/');
19          var title = browser.getTitle();
20          title.should.equal('WebDriverUniversity.com');
21          console.log('Title is: ' + title);
22
23          browser.click('#login-portal');
24
25          browser.pause(3000);
```

8. We are now ready to handle the tabs by adding code on lines 13 and 24
9. Add the following (lines 13, 14, 15 and 17) so that it looks like this:

```
12          browser.click("#contact-us");
13          var tabIds = browser.getTabIds();
14          console.log(tabIds);
15          browser.switchTab(tabIds[1]);
16          browser.pause(3000);
17          browser.close();
18      });
```

    a. We have made the following changes:
        i. Created a new variable that stores the tab ID

ii. We then log the tab ID to the console
iii. We then switch the browser tab using the switchTab command and pass it the tabIds variable defined above
iv. Then we close the browser window once the test is complete

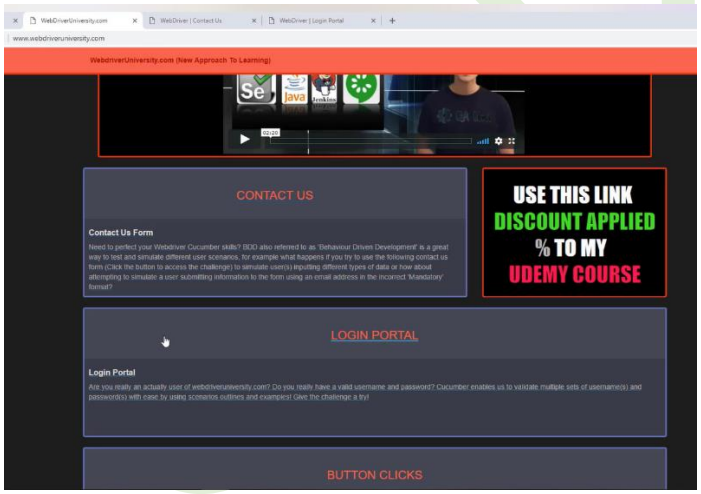
b. Then the test will move onto the next "it" code block
c. We then apply the same logic to the second "it" test and your code should look like this:

```
20    it("check that the login button opens the login portal page", function
21        browser.url('/');
22        var title = browser.getTitle();
23        title.should.equal('WebDriverUniversity.com');
24        console.log('Title is: ' + title);
25
26        browser.click('#login-portal');
27        var tabIds = browser.getTabIds();
28        browser.switchTab(tabIds[1]);
29        browser.pause(3000);
```

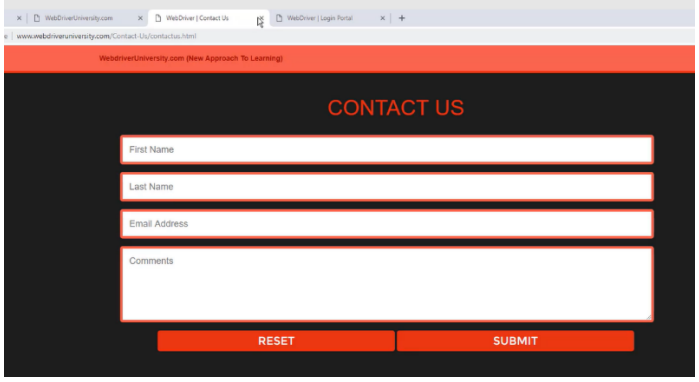
d. We made the following changes:
i. Created a new variable that stores the tab ID
ii. We then log the tab ID to the console
iii. We then switch the browser tab using the switchTab command and pass it the tabIds variable defined above

To make it easier to understand how we envision our test to perform, we expect the following:

1. The test is going to go to webdriveruniversity.com:

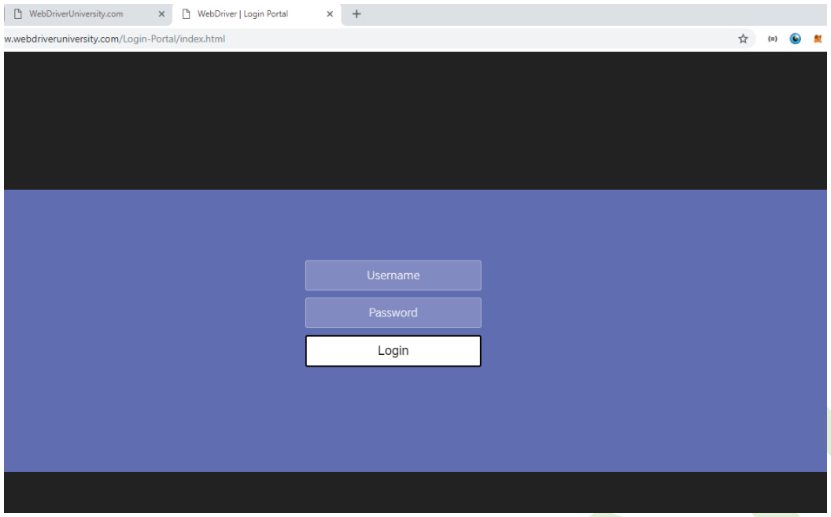


2. Our test will then click the contact us link which will open a new browser window



3. The test will complete the first test from the first "it" code block and then close the tab

4. The second test (second “it” block) will start
5. The test will go back to the webdriveruniversity.com homepage
6. It will then click the login portal link which will open a new browser tab as shown:



7. The second test will then complete

Please refer to the lecture video for a more detailed explanation if required.

We continue with Mocha hooks in the next lecture

Lecture 44 - Tabs Part 2

In this lecture we continued on from Part 1 but this time we added a couple of assertions to ensure the correct tab was in focus during out tests. So, for example, in our first "it" test we want to check to see if the contact us page opens in a new browser window. We can add an assertion into the test to check the URL of the newly opened page and can go a step further by checking the webpage title.

Key Points:

- We want to check to see if the correct webpage is opened when opening a new webpage in an additional tab
- We can do this by adding a few assertions into our test
- We test to see if the webpage URL includes a phrase that we were expecting by using the browser.getUrl() method
- We then check the webpage title to see if the title matches the value that we were expecting

Instructions:

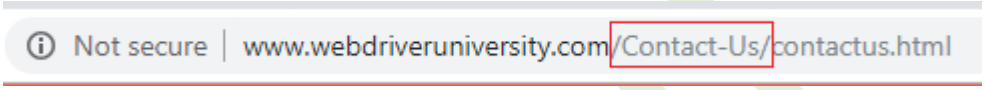
Adding assertions to our webdriverUniversityTest.js file

1. If you haven't already got the file above open, use windows explorer (for windows) or Finder (on Mac) to open the webdrivenioFramework directory, move into the tests folder and open the webdriverUniversityTest.js file using Sublime Text
2. We added the following code (which I explain below):

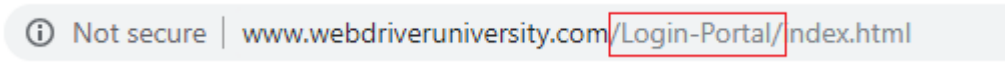
webdriverUniversityTest.js

```
describe("Verify whether webdriveruniversity links on homepage work correctly", function() {
    it("check that the contact us button opens the contact us page", function(done) {
        browser.setViewportSize({
            width: 1200,
            height: 800
        })
        browser.url('/');
        var title = browser.getTitle();
        expect(title).to.equal('WebDriverUniversity.com');
        console.log('Title is: ' + title);

        browser.click("#contact-us");
        var tabIds = browser.getTabIds();
        console.log(tabIds);
        browser.switchTab(tabIds[1]);

        var title2 = browser.getTitle();
        expect(title2).to.equal('WebDriver | Contact Us');          1

        var url = browser.getUrl();
        expect(url).to.include('Contact-Us', 'URL Mismatch');       2
        browser.close();
    });

    it("check that the login button opens the login portal page", function(done) {
        browser.url('/');
        var title = browser.getTitle();
        title.should.equal('WebDriverUniversity.com');
        console.log('Title is: ' + title);

        browser.click('#login-portal');
        var tabIds = browser.getTabIds();
        browser.switchTab(tabIds[1]);

        var title2 = browser.getTitle();
        expect(title2).to.equal('WebDriver | Login Portal');        3

        var url = browser.getUrl();
        expect(url).to.include('Login-Portal', 'URL Mismatch');     4
    });
});
```

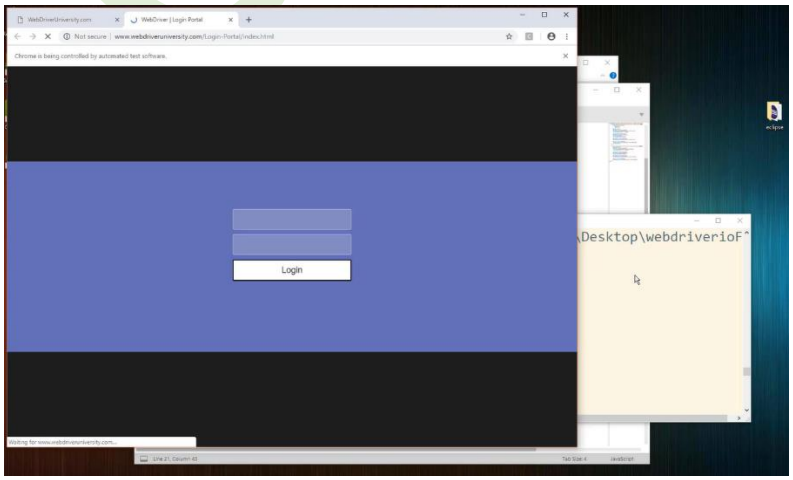
3. Point Number 1
    a. Here we have added a new variable called title2 that uses the browser.getTitle() method which will return the webpage title
    b. We then added an assertion using the *expect* method where we are checking to see if the returned variable above is the same as the title we are expecting (Webdriver | Contact-Us). If a match is not found, then we output the value "URL Mismatch"
4. Point Number 2
    a. We have created a new variable called url and then use the browser.getUrl() method which will store the webpage URL from the browser
    b. We again create another assertion using the *expect* method but this time we have used the **to.include** method which looks to see if a given phrase is included in the value provided. We are expecting the URL to include the following text "Contact-Us" as the URL to the contact us page should be the following:



    c. If the URL doesn't match the value in our test, then "URL Mismatch" should be reported
5. Point Number 3
    a. We now amend our second "it" test and make similar changes as explained in Point 1, except this time we are focusing on the Login Portal webpage. We used the exact same assertion in Point 1 but this time we expect the webpage title to return the value "Webdriver | Login Portal"
6. Point Number 4
    a. Similar to Point 2, we added another assertion using the to.include method and expect the URL to contain the following value: "Login-Portal". This is because the URL to the login portal page should be:



7. We also removed the browser.pause(3000) lines as this is no longer required since you've already seen the tests perform (demonstration purposes to slow down the tests)
8. Once you have made the changes above, save your file and close Sublime Text
9. Open up GitBash / iTerm2 and navigate to our webdrivierioFramework directory
10. Run the following command:
11. **npm test -- -- specs=tests/webdriverUniversityTest.js** + press enter
12. You should see the tests run, as shown below:



13. Once the tests have completed, the command line should read:

   a. Two tests have passed and were completed in 15.10 seconds
   b. This confirms the URL of the webpages opened in the additional browsers match the values we were expecting (otherwise the test would have failed and reported the issue)
   c. It also confirms the webpage title matches the values we were expecting

In the lecture video I demonstrated a failure scenario when the URL doesn't match the expected value that we defined. The command line output a failure exception message and showed the cause of the problem. Please refer to the lecture video if you wish to see this again.

Module 17 – Verify Elements

In this module we take a more detailed look at the webdriverio API and go through some useful and popular functions. By having a good understanding of this area will build your testing toolset and will allow you to create comprehensive automation tests for a number of different scenarios.

Lecture 45 - isExisting Part 1

In this lecture we take a look at the isExisting function in detail. This function checks to see if at least one element is existing using a selector we provide. For example, we may want to check to see if a h1 heading exists on the webpage and our test will return either true or false.

Lecture Resources on Udemy:

- http://v4.webdriver.io/api.html
- http://v4.webdriver.io/api/state/isExisting.html
- http://www.webdriveruniversity.com/Hidden-Elements/index.html

Key Points:

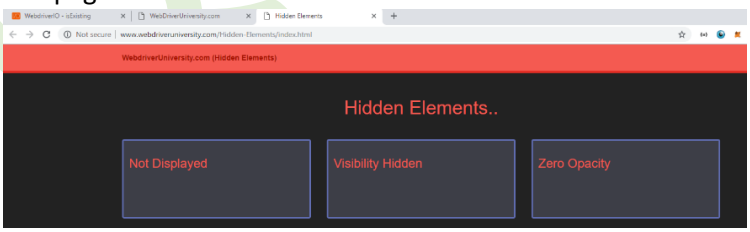
- We take a look at the webdriverio API documents, specifically the isExisting function
- This function can be used to check if an element exists on a page
- The function will return a true (if found) or false (if not) result

How to use this function:

browser.isExisting(selector); ← where *selector* is the element selector that we use to identify an element (e.g. a unique ID)

Instructions:

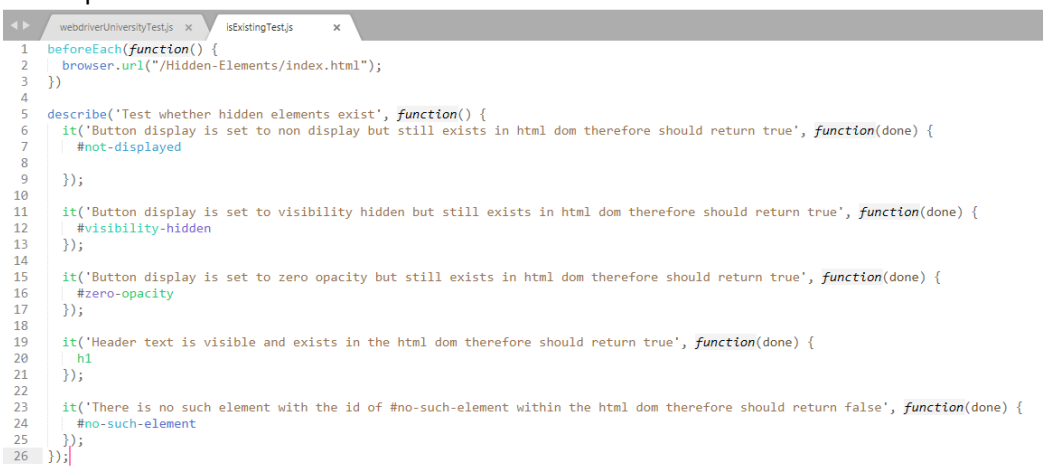
1. First, we reviewed the webdriveruniversity.com hidden elements page as we are going to be using this to test the isExisting function
2. This page can be found here:
    a. http://www.webdriveruniversity.com/Hidden-Elements/index.html
    b. The page looks like this:



    c. This page has three div elements that have a button which aren't initially visible:
        i. Not Displayed
            1. Has an ID of not-displayed
            2. Has a CSS property of display: none; which prevents the button from showing
        ii. Visibility Hidden
            1. Has an ID of visibility-hidden
            2. Has a CSS property of visibility: hidden which too prevents the button from showing

iii. Zero Opacity
    1. Has an ID of zero-opacity
    2. Has a CSS property that sets the opacity of the button to zero, meaning it would still be clickable but not visible to the user

3. Go to the lecture resources on Udemy and download the code attachment. Copy it over to your tests folder and save it with a name of isExistingTest.js
4. Then open the document with Sublime Text and it should look like:



```
webdriverUniversityTest.js   isExistingTest.js

beforeEach(function() {
  browser.url("/Hidden-Elements/index.html");
})

describe('Test whether hidden elements exist', function() {
  it('Button display is set to non display but still exists in html dom therefore should return true', function(done) {
    #not-displayed

  });

  it('Button display is set to visibility hidden but still exists in html dom therefore should return true', function(done) {
    #visibility-hidden
  });

  it('Button display is set to zero opacity but still exists in html dom therefore should return true', function(done) {
    #zero-opacity
  });

  it('Header text is visible and exists in the html dom therefore should return true', function(done) {
    h1
  });

  it('There is no such element with the id of #no-such-element within the html dom therefore should return false', function(done) {
    #no-such-element
  });
});
```

   a. We have placed a forEach Mocha hook at the top of the test which will load the URL (using the base URL set in the WDIO.config file) + /Hidden-Elements/index.html which will send our test to the hidden element's webpage on webdriveruniversity.com
   b. This test has a description of: "Test whether hidden elements exist" as we are trying to test whether the hidden elements exist on the page
   c. We then have multiple "it" tests. These do the following;
      i. The first three are looking for each of the buttons and uses locators (button id's) to identify them
      ii. The fourth is doing a test on the header and we use the H1 tag as the locator as there should only ever be one H1 on a webpage (unique)
      iii. The last is a false test where we are use a locator that does not exist on the webpage and so we would expect our test to report a FALSE value as it cannot be located

We continue with isExisting in the next lecture (part 2)

Lecture 46 - isExisting Part 2

In this lecture we continue building the isExistingTest.js file. We currently have the skeleton test defined and have placed locators in each of the "it" test blocks. We now need to use the locators to test for expected values.

Lecture Resources on Udemy:

- http://v4.webdriver.io/api.html
- http://v4.webdriver.io/api/state/isExisting.html
- http://www.webdriveruniversity.com/Hidden-Elements/index.html

Key Points:

- The skeleton test isExistingTest.js file is now in place
- We have gone through the locators that enable us to locate the elements on the page
- We now need to complete the isExistingTest.js file to make use of the locators by using the isExisting function
- This will enable us to check whether the buttons exist on the DOM (Document Object Model) when our tests run

What is the HTML DOM?

The HTML DOM is a standard object model and programming interface for HTML. It defines:

- The HTML elements as objects
- The properties of all HTML elements
- The methods to access all HTML elements
- The events for all HTML elements
- Learn more here: https://www.youtube.com/watch?v=-0ZcldkGlt8

Instructions (continued from Part 1):

1. We amend the first "it" test block to look like this:

```
5  describe('Test whether hidden elements exist', function() {
6    it('Button display is set to non display but still exists in html dom therefore should return true', function(done) {
7      var isEnabled = browser.isExisting('#not-displayed');
8      console.log(isEnabled);
9      expect(isEnabled).to.equal(true);
10   });
```

   a. Here we are creating a new variable called isEnabled and assign it a value that uses browser.isExisting which uses a parameter value of the locator for the first test (id of #not-displayed which is associated to the first button)
   b. We then log the value of this variable to the console (so we can see it in the command line when our test runs – for demonstration purposes)
   c. We then add an assertion using *expect* and provide it with the variable value and expect the value to return 'true' as we know the button does exist. If it doesn't exist, we would expect this test to fail

2. We amend the second "it" test block to look like this:

```
12   it('Button display is set to visibility hidden but still exists in html dom therefore should return true', function(done) {
13     var isEnabled = browser.isExisting('#visibility-hidden');
14     console.log(isEnabled);
15     expect(isEnabled).to.equal(true);
16   });
```

   a. This does exactly the same as the first "it" test block except this time we use the #visibility-hidden locator for the second button

3. We amend the third “it” test block to look like this:

```
18  it('Button display is set to zero opacity but still exists in html dom therefore should return true', function(done) {
19    var isEnabled = browser.isExisting('#zero-opacity');
20    console.log(isEnabled);
21    expect(isEnabled).to.equal(true);
22  });
```

   a. This does exactly the same as the first “it” test block except this time we use the #zero-opacity locator for the third button

4. We amend the fourth “it” test block to look like this:

```
24  it('Header text is visible and exists in the html dom therefore should return true', function(done) {
25    var isEnabled = browser.isExisting('h1');
26    console.log(isEnabled);
27    expect(isEnabled).to.equal(true);
28  });
```

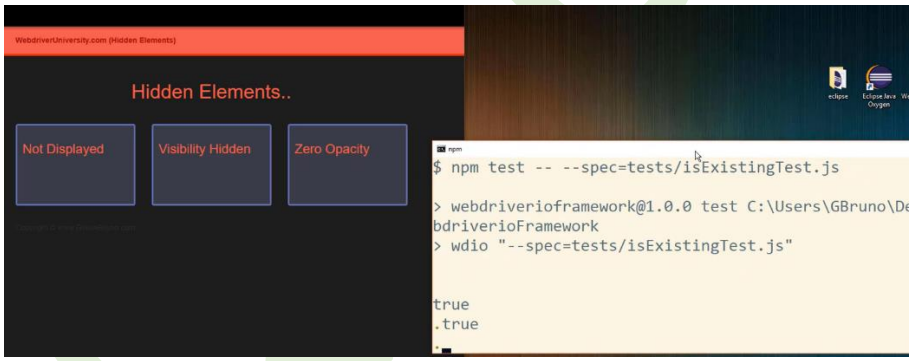
   a. This too does exactly the same as the first “it” test block except this time we use the h1 locator for the header section (remember, there is only one h1 element on the page, hence why we can use the h1 element instead of needing to use an id).

5. Lastly, we amend the last “it” test block to look like this:

```
30  it('There is no such element with the id of #no-such-element within the html dom therefore should return false', function(do
31    var isEnabled = browser.isExisting('#no-such-element');
32    console.log(isEnabled);
33    expect(isEnabled).to.equal(false);
34  });
35 });
```

   a. This too does the same as the first “it” code block, **but we expect a <u>false</u> value to be returned** because we know there isn’t an element with an id of #no-such-element on the webpage/ HTML DOM

6. Once you have made the changes above, save the file and close Sublime Text
7. Open up GitBash / iTerm2 and navigate to our webdriverioFramework directory
8. Run the following command:
9. **npm test -- -- specs=tests/isExistingTest.js** + press enter
10. You should see the tests run, as shown below:



11. Once the test has completed, take a look at the console output:

12. You should see that 5 tests have passed! And more impressively 5 tests were completed in 10 seconds!!
13. The tests passed because our test was able to locate the elements and returned true/false values that were identical to what we were expected (i.e. it found the h1 header element and returned true and we were expecting a true value, as we knew a h1 existed on the page).

Lecture 47 - isVisible

In this lecture we take a look at the isVisible function. This function will return true if the selected DOM-element is visible for the locator that we provide.

Lecture Resources on Udemy:

- http://v4.webdriver.io/api.html
- http://v4.webdriver.io/api/state/isVisible.html
- http://www.webdriveruniversity.com/Hidden-Elements/index.html

Key Points:

- We use the isVisible method that will return a true or false value depending on whether the element in scope is visible in the browser
- We again use the Hidden Elements page on webdriveruniversity.com and will conduct our tests using the buttons
- A skeleton file is again provided, for which you can download from the Udemy resource section associated to this lecture

Instructions:

How to set up and run an isVisible test

1. Download the skeleton test file from lecture resource section on Udemy. It will look like this:

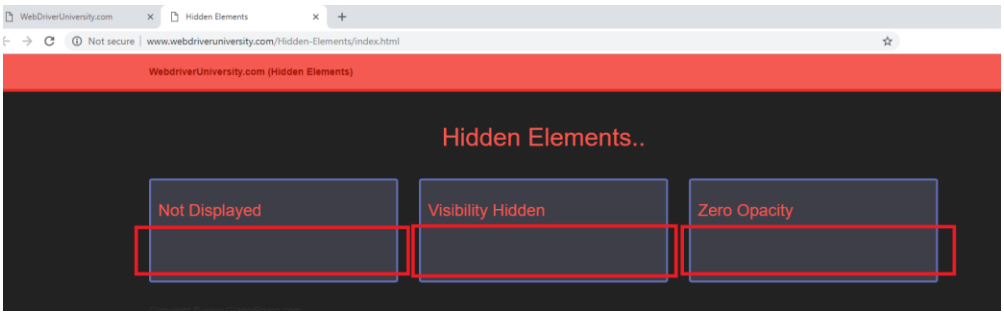
File Edit Selection Find View Goto Tools Project Preferences Help

isVisibleTest.js

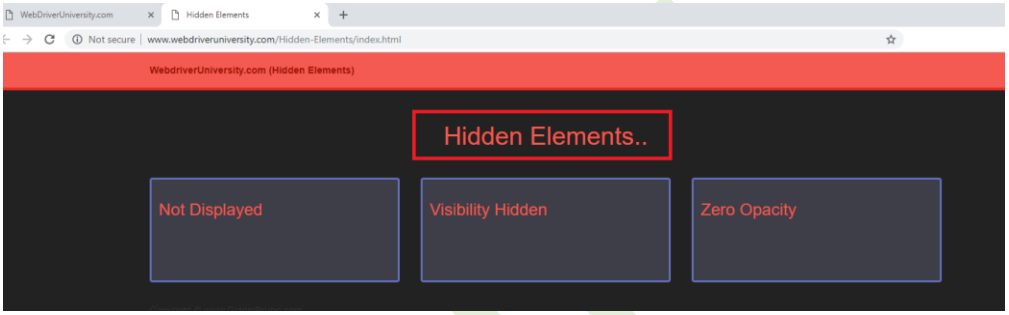
```
1  beforeEach(function() {
2    browser.url("/Hidden-Elements/index.html");
3  })
4
5  describe('Test whether hidden elements are visible', function() {
6    it('Button display is set to non display therefore should return false', function(done) {
7      var isVisible = browser.isVisible('#not-displayed');
8      console.log(isVisible);
9      expect(isVisible).to.equal(false);
10   });
11
12   it('Button display is set to visibility hidden therefore should return false', function(done) {
13     var isVisible = browser.isVisible('#visibility-hidden');
14     console.log(isVisible);
15     expect(isVisible).to.equal(false);
16   });
17
18   it('Button display is set to zero opacity therefore should return false', function(done) {
19     var isVisible = browser.isVisible('#zero-opacity');
20     console.log(isVisible);
21     expect(isVisible).to.equal(false);
22   });
23
24   it('Header text is visible therefore should return true', function(done) {
25     var isVisible = browser.isVisible('h1');
26     console.log(isVisible);
27     expect(isVisible).to.equal(true);
28   });
29 });
```

2. Save this file to your tests folder (webdrivenioFramework/tests) and call it isVisibleTest.js
3. Then, review the file – some key points are:
    a. Much of the file is similar to that from the previous two lectures where we are using the same locators but are instead using the isVisible function (instead of the isExisting function)
    b. If we look at the assertions on line 9, 15, 21 and 27 you will see we have defined expected values (false, false, false, true) and this is because we know that the first

three “it” tests are looking to see if the buttons are visible for which we know they aren’t visible due to their .CSS properties:

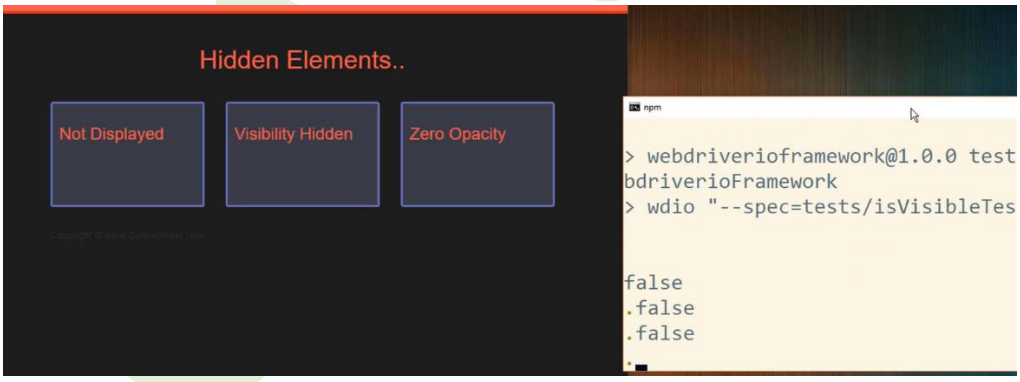


c. The last “it” test is instead looking for the h1 elements (header) which we know is visible, as shown:



i. As the header is visible, this test should return “true”.

4. Once, you’ve reviewed the file, save it and close Sublime Text
5. Open up GitBash / iTerm2 and navigate to our webdrivenioFramework directory
6. Run the following command:
7. **npm test -- -- specs=tests/isVisibleTest.js** + press enter
8. You should see the tests run, as shown below:



9. And if we review the console output, you should see:

a. Where four tests have passed
b. The values from the test results are false, false, false and true (as expected)

Lecture 48 – hasFocus – Part 1

In this lecture we take a look at the hasFocus function. This function allows us to test whether an element has focus (e.g. we have instructed our test to select a checkbox option and we want to prove the checkbox element is now in focus of the browser).

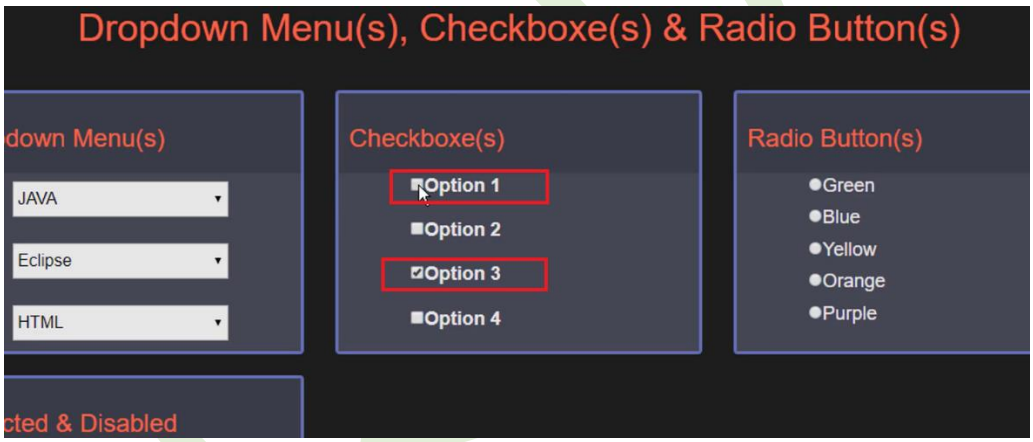
Lecture Resources on Udemy:

- http://v4.webdriver.io/api.html
- http://v4.webdriver.io/api/state/hasFocus.html
- http://www.webdriveruniversity.com/Dropdown-Checkboxes-RadioButtons/index.html

Key Points:

- We use the hasFocus function that returns a true or false value
- This function can be used to report whether the selected DOM-element currently has focus
- A typical example of where we would use this is on a webform. If we instruct our test to a webpage containing a form, a field from the form would not automatically have focus (unless an autofocus property has been defined for the html element).
    - The test at this point would return false, where as if we instructed out test to select the field (e.g. by using click()) then the test would return true
- A skeleton file is again provided, for which you can download from the Udemy resource section associated to this lecture

Details about this test:



In this test we are going to use the above webpage, specifically the checkbox area which can be found here: http://www.webdriveruniversity.com/Dropdown-Checkboxes-RadioButtons/index.html

- **The first test** will select checkbox Option 1 (which is currently unticked) so that it becomes ticked. We then check to see if this option has focus (which should return **true**)
- **The second test** will focus on checkbox Option 3 which is already ticked when visiting the page. This should report **false** because we have not selected this value (as it was pre-selected) and does not have focus.

Continued on next page

Instructions:

Setting up our base test file

1. Open up Sublime Text and create a new file
2. Call it hasFocusTest.js and save it in our “tests” folder
3. Open the file again and write the following code:

```
describe('Test Radio Buttons Page', function() {
  browser.url("Dropdown-Checkboxes-RadioButtons/index.html");

  it('Should be able to focus on radio button elements', function(done) {

  });
});
```

   a. This code should be familiar to you now. Here we are creating the base test case by giving the test a “describe” value which is to test radio buttons page
   b. We then set the browser URL using the base URL set in the WDIO.config file + the webpage URL defined above
   c. We have then created an “it” test which “should be able to focus on radio button elements” and we will then add the test code within this block

Instructions:

How to get locators for the checkbox selections & how to create the tests

- We need to retrieve locators in order to interact with the elements of interest
- This is so we can tell our test to select or to complete a function on the element provided

1. Go to the webdriveruniversity checkbox page using the following link:
   a. http://www.webdriveruniversity.com/Dropdown-Checkboxes-RadioButtons/index.html
2. Hover over the checkbox Option 1, right click and select ‘Selector Actions’ and then select Copy CSS
   a. This will save the locator for option 1 to your clipboard
3. Then paste this value inside the “it” block
4. Do the same for the third checkbox Option 3
5. Your code at this point should look like the following:

```
describe('Test Radio Buttons Page', function() {
  browser.url("Dropdown-Checkboxes-RadioButtons/index.html");

  it('Should be able to focus on radio button elements', function(done) {
    #checkboxes label:nth-of-type(1) [type]
    #checkboxes label:nth-child(5) [type]

  });
});
```

Continued next page

6. We now need to use these locators to formulate our tests. First, we focus on checkbox Option 1 and add the following code:

```
untitled
1  describe('Test Radio Buttons Page', function() {
2    browser.url("Dropdown-Checkboxes-RadioButtons/index.html");
3
4    it('Should be able to focus on checkbox button elements', function(done) {
5      browser.click('#checkboxes label:nth-of-type(1) [type]');
6      var checkboxButtonOne = browser.hasFocus('#checkboxes label:nth-of-type(1) [type]');
7      console.log('Checkbox button one has a value of: ' + checkboxButtonOne);
8      expect(checkboxButtonOne, 'The checkbox (One) should have focus!').to.be.true;
9
10     var radioButtonOne = browser.hasFocus('#checkboxes label:nth-of-type(1) [type]');
11     #checkboxes label:nth-child(5) [type]
12
13   });
14 });
```

a. **On line 5** – we are using the browser.click method to select option 1 of the checkbox and provide the locator css value
b. **On line 6** – we create a new variable called checkboxButtonOne and set it a value using the browser.hasFocus function. We then provide the locator value so the function knows what element it is checking.
c. **On line 7** – we are simply logging some text to the console which will display "Checkbox button one has a value of" + the variable value which will be true or false when the test reaches this point
d. **On line 8** – we are adding an assertion using the expect method and provide an argument of checkboxButtonOne, provide a message that's to be returned and then we use *.to.be.true* – as we expect the value to return true in this case

We continue with this test in the next lecture

Lecture 49 – hasFocus – Part 2

In this lecture we continue from Part 1 and continue adding to our test

Key Points:

- So far, we have created the skeleton test
- Have identified the locators needed to select the relevant checkbox selections
- We have written the code for the first "it" block where we are;
    - Selecting the first checkbox option
    - Checking to see if the checkbox has focus
    - Logging the value of this check to the console (true or false)
    - Using an assertion as we expect the value to be true (meaning the checkbox value 1 has focus at this point)

Instructions (continued):

1. We now need to use second locator to formulate the rest of our test. This will now focus on the third checkbox Option 3 (which is already ticked by default)
2. Add the following code:

```
1  describe('Test Checkboxes Buttons Page', function() {
2      browser.url("Dropdown-Checkboxes-RadioButtons/index.html");
3
4      it('Should be able to focus on checkbox button elements', function(done) {
5          browser.click('#checkboxes label:nth-of-type(1) [type]');
6          var checkboxButtonOne = browser.hasFocus('#checkboxes label:nth-of-type(1) [type]');
7          console.log('Checkbox button one has a value of: ' + checkboxButtonOne);
8          expect(checkboxButtonOne, 'The checkbox (One) should have focus!').to.be.true;
9
10         var checkboxButtonTwo = browser.hasFocus('#checkboxes label:nth-child(5) [type]');
11         console.log('Checkbox button two has a value of: ' + checkboxButtonTwo);
12         expect(checkboxButtonTwo, 'The checkbox (Two) should have focus!').to.be.false;
13     });
14 });
```

    a. **On line 10** – we are again creating a new variable called checkboxButtonTwo and set it a value using the browser.hasFocus function. We then provide the second locator value, so the function knows what element it is checking
    e. **On line 11** - we are simply logging some text to the console which will display "Checkbox button one has a value of" + the variable value which will be true or false when the test reaches this point
    b. **On line 12** - we are adding an assertion using the expect method and provide an argument of checkboxButtonTwo, provide a message that's to be returned and then we use *.to.be.false* – as we expect the value to return false in this case as we have not specifically focused on the checkbox option 3
3. We now need to make two more small changes to this file.
    a. We need to add the browser.setViewporSize() to define a viewpoint when our test runs (this is to overcome a problem where sometimes a test is unable to select a particular element).
    b. Add the following code:

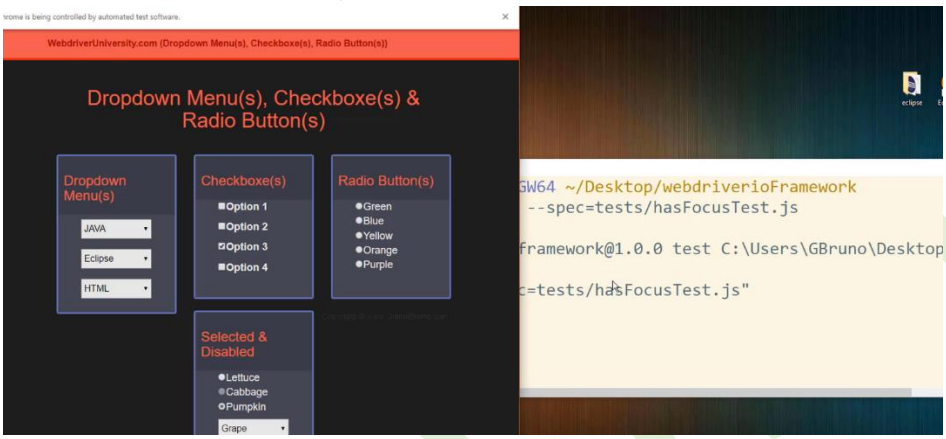
```
4      it('Should be able to focus on checkbox button elements', function(done) {
5          browser.setViewportSize({
6              width: 1200,
7              height: 800
8          })
9          browser.click('#checkboxes label:nth-of-type(1) [type]');
```

c. We then need to add a pause at the beginning of the "it" block so that there is sufficient time for the page to load before our test attempts to select the checkbox Option 1

d. Add the following code:

```
9       browser.pause(2000);
10      browser.click('#checkboxes label:nth-of-type(1) [type]');
```

4. Now save your file and close Sublime Text
5. Open up GitBash / iTerm2 and navigate to our webdriverioFramework directory
6. Run the following command:
7. **npm test -- -- specs=tests/hasFocusTest.js** + press enter
8. You should see the tests run, as shown below:



9. Then if we review the console output, we should see:

10. As you can see, the test has passed (don't forget we combined the two tests into one "it" block, hence why there is only 1 pass shown). Also take note of the output at the top, the values from the hasFocus test has returned true (for the first checkbox Option 1) and false (for the third checkbox option 3) which is what we were expecting!

Lecture 50 – isEnabled – Part 1

In this lecture we take a look at the isEnabled function. This function allows us to test whether an element is enabled and will return a true or false value. A typical example of when we would use this function is when we test web forms. Sometimes a form field will be disabled (e.g. a second email input field) until the first email input field has been entered.

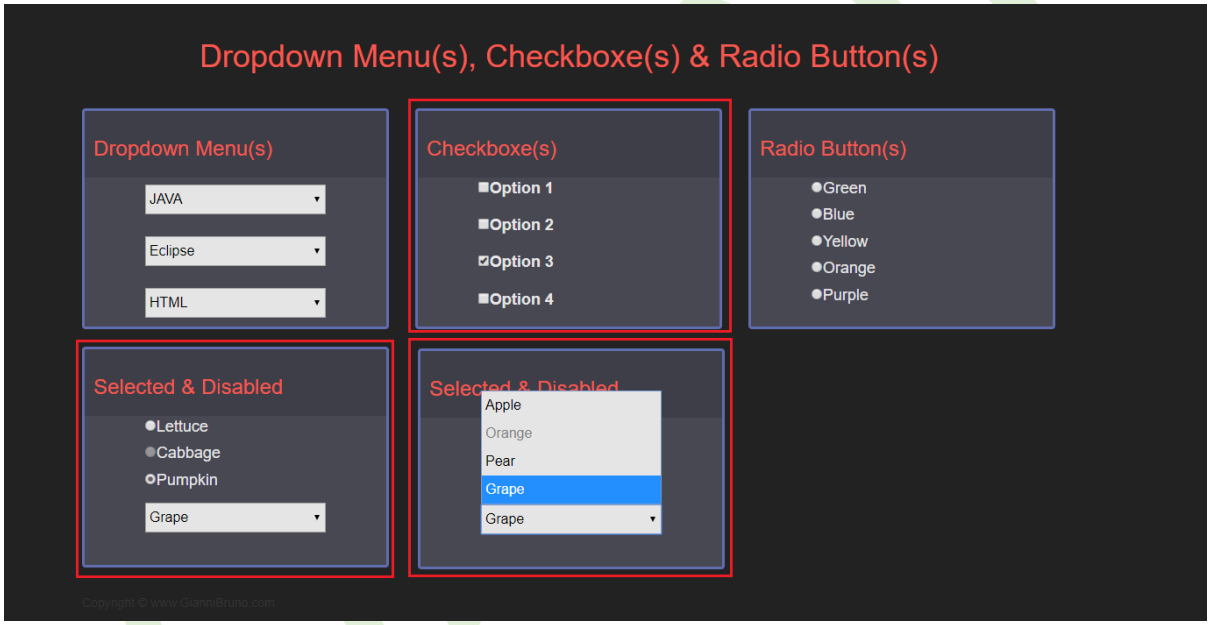
Lecture Resources on Udemy:

- http://v4.webdriver.io/api/state/isEnabled.html
- http://www.webdriveruniversity.com/Dropdown-Checkboxes-RadioButtons/index.html

Key Points:

- We use the isEnabled function that returns a true or false value
- We would use this function to see if an element if enabled or disabled

Details about this test:



In this test we are going to use the above webpage, specifically the Checkboxes and Selected & Disabled areas, which can be found here: http://www.webdriveruniversity.com/Dropdown-Checkboxes-RadioButtons/index.html

We will run multiple tests but take note of second checkbox option from the Selected & Disabled section and the second dropdown menu options (also from the Selected & Disabled section) which are disabled. This is the perfect opportunity to use the isEnabled function which should return false if we attempt to select these two values.

Instructions:

1. Download the code from the Udemy resources section for this lecture
2. Save this file in your tests folder and ensure it has the named isEnabledTest.js
3. The content of this file should look like the below (for which I explain):

Continued on next page

isVisibleTest.js | isEnabledTest.js

```
1  beforeEach(function() {
2    browser.url("/Dropdown-Checkboxes-RadioButtons/index.html");
3    browser.setViewportSize({
4      width: 1200,
5      height: 800
6    })
7    browser.pause(2000);
8  })
9
10 describe('Test Enabled Dropdown Menus, Checkboxes & Radio Buttons', function() {
11   it('Dropdown item orange is disabled therefore should return false', function(done) {
12       option[value='orange']
13   });
14
15   it('Dropdown item grape is enabled therefore should return true', function(done) {
16     option[value='grape']
17   });
18
19   it('Option2 is enabled therefore should return true', function(done) {
20     input[value='option-2']
21   });
22
23   it('Radio button pumpkin is enabled therefore should be true', function(done) {
24     input[value='pumpkin']
25   });
26
27   it('Radio button cabbage is disabled therefore should be false', function(done) {
28     input[value='cabbage']
29   });
30 });
```

4. Most of this code should be familiar with you by now, as much of it has been reused from past lectures
5. To give a brief explanation:
    a. We have set the browser URL and viewpoint size
    b. We have added a pause so there is sufficient time for the page to load
    c. We have created a describe code block and provided a description of what this test is expected to carry out
    d. We then have multiple “it” code blocks for which we will use to set our tests
6. Notice how the locators are already defined in the “it” blocks, where:
    a. Lines 12 & 16 – are testing the dropdown menu items
    b. Line 20 – is testing the Checkboxes section (Option 2 from the list)
    c. Lines 24 & 28 – are testing the Radio buttons from the Selected & Disabled section
    d. We use the value of these selections to locate the elements in our tests
        i. You can attempt to practise finding these locators on the webpage by using developer mode (F12 using Chrome) and selecting the elements with the selector tool and then by using Ranorex to find unique locators

We continue with this test in the next lecture

Lecture 51 – isEnabled – Part 2

In this lecture we continue developing our test (from Part 1) and will now add assertions into our "it" blocks.

Lecture Resources on Udemy:

- http://v4.webdriver.io/api/state/isEnabled.html
- http://www.webdriveruniversity.com/Dropdown-Checkboxes-RadioButtons/index.html

Key Points:

- We have used the skeleton code from the Udemy resources of lecture 48 to create the skeleton test file
- We have the locators needed to select the checkbox, radio buttons and dropdown elements and have placed this into multiple "it" code blocks
- We now need to add assertions and log the output of each test

Instructions:

Developing our tests further

1. Amend the first "it" block to look like:

```
11  it('Dropdown item orange is disabled therefore should return false', function(done) {
12    var isEnabled = browser.isEnabled("option[value='orange']");
13    console.log(isEnabled);
14    expect(isEnabled).to.equal(false);
15  });
```

   a. We have created a new variable and assigned it a value using browser.isEnabled and then provided the locator (option[value='orange']) to select it
   b. We then write the output of the variable to the console
   c. Finally, we add an assertion using expect where we expect the value to be false

2. Amend the second "it" block to look like:

```
17  it('Dropdown item grape is enabled therefore should return true', function(done) {
18    var isEnabled = browser.isEnabled("option[value='grape']");
19    console.log(isEnabled);
20    expect(isEnabled).to.equal(true);
21  });
```

   a. As above except we use a different locator and expect the value to be true because it's selectable

3. Amend the third "it" block to look like:

```
23  it('Option2 is enabled therefore should return true', function(done) {
24    var isEnabled = browser.isEnabled("input[value='option-2']");
25    console.log(isEnabled);
26    expect(isEnabled).to.equal(true);
27  });
28
```

   a. As above except we are using a different locator for the checkbox (option 2)

4. Amend the fourth "it" block to look like:

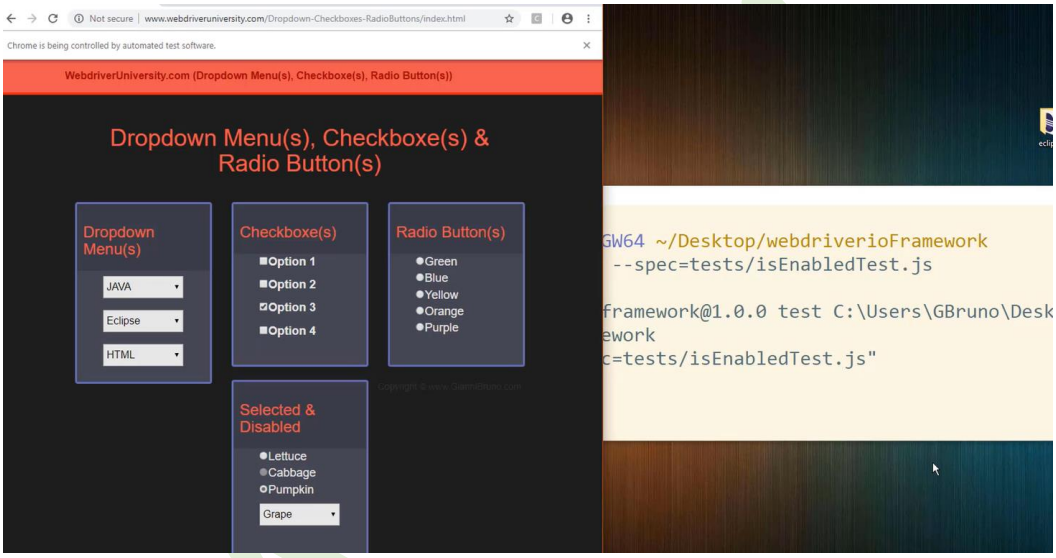
```
29  it('Radio button pumpkin is enabled therefore should be true', function(done) {
30    var isEnabled = browser.isEnabled("input[value='pumpkin']");
31    console.log(isEnabled);
32    expect(isEnabled).to.equal(true);
33  });
34
```

   a. As above except we are using a different locator

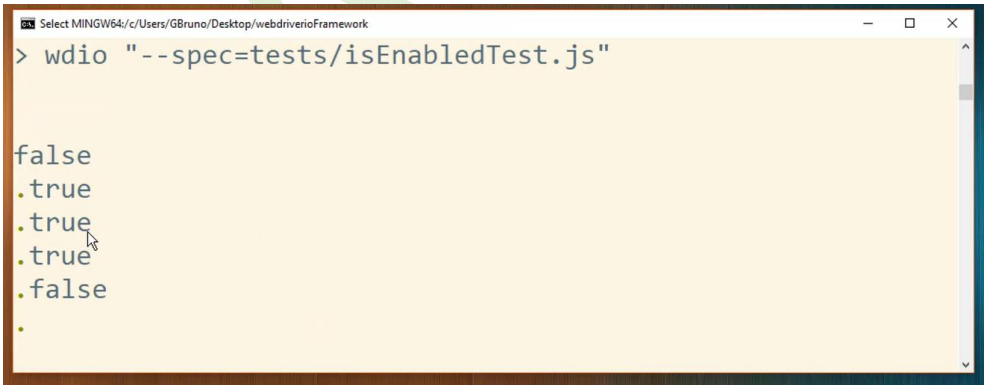
5. Finally, we amend the last “it” block to look like:

```
35  it('Radio button cabbage is disabled therefore should be false', function(done) {
36    var isEnabled = browser.isEnabled("input[value='cabbage']");
37    console.log(isEnabled);
38    expect(isEnabled).to.equal(false);
39  });
```

   a. Again, this is the same as above except for the different locator and that we are expecting a false value to be returned, because the “cabbage” selection on the radio button is disabled
6. Now save your file and close Sublime Text
7. Open up GitBash / iTerm2 and navigate to our webdriverioFramework directory
8. Run the following command:
9. **npm test -- -- specs=tests/isEnabledTest.js** + press enter
10. You should see the tests run, as shown below:



11. Then if we review the console output, we should see:



12. As you can see, all tests have passed, and the values outputted are identical to the values that we were expecting. This is because:
   a. The first test was based on selecting the dropdown menu “orange” which is disabled
   b. The second was based on the dropdown menu “grape” which is selectable
   c. The third was based on option 2 (the second option from the Checkboxes section) which returned true, as we can select it
   d. The fourth was to select the “pumpkin” radio button which returned true as it’s selectable
   e. The last was to select the “Cabbage” radio button which returned false because it is disabled and cannot be selected

Lecture 52 – isSelected – Part 1

In this lecture take a look at the isSelected function which can be used to determine if a radio button option has been selected or not. It will return a true or false value.

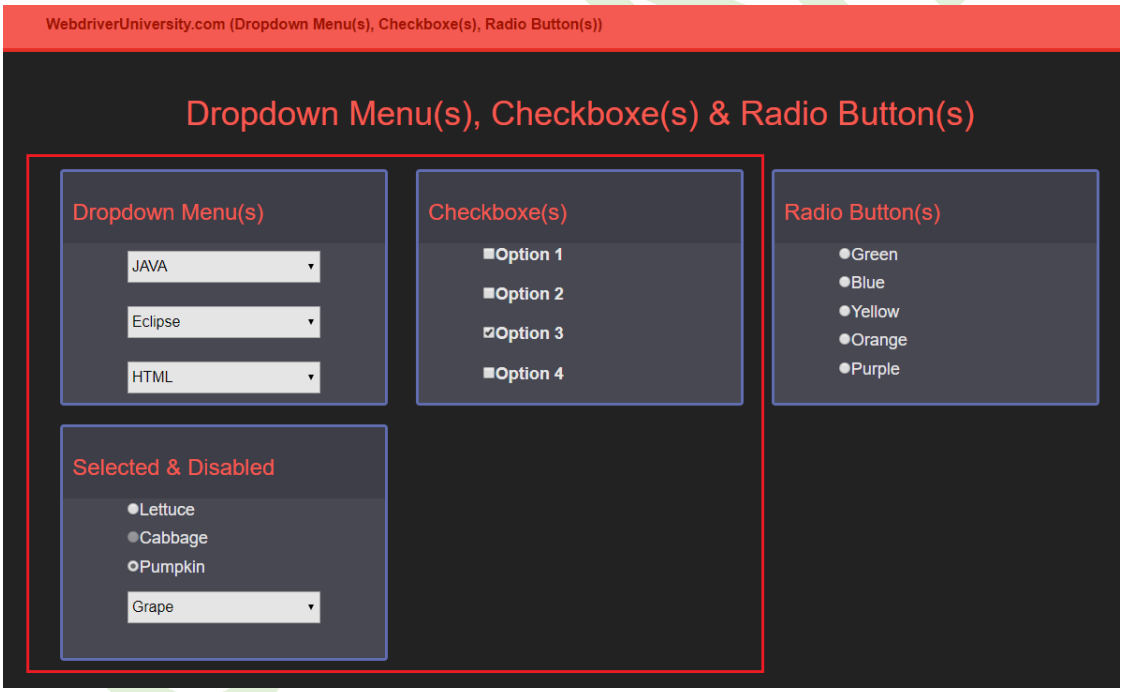
Lecture Resources on Udemy:

- http://v4.webdriver.io/api/state/isSelected.html
- http://www.webdriveruniversity.com/Dropdown-Checkboxes-RadioButtons/index.html

Key Points:

- We will again be using the Dropdown-Checkboxes-RadioButtons webpage from webdriveruniversity.com to create this test
- We want to focus on the Checkboxe(s) and Selected & Disabled sections and report whether these options are selected or not
- A skeleton code file can be downloaded from the Udemy resources section

Details about this test:



- A skeleton code file is already provided to you, which you can download from the Udemy resources section of this lecture
- We will be using the Dropdown Menu(s), Checkboxes, and Selected & Disabled sections from this webpage to conduct our tests
- We want to see if particular values from these sections are selected using the isSelected function
- When a user visits this page, you will see that some values are already selected;
    - We intend to test these pre-selected values and the isSelected function should return “true”
    - We will also test some unselected values where we would expect the isSelected function to return false

Instructions:

Setting up a test file to use the isSelected function

1. Download the lecture resources from Udemy associated to this lecture
2. Save the file in your "tests" folder and ensure it has the name isSelectedTest.js
3. Open the file using Sublime Text and your code should look like this:

isSelectedTest.js

```
1  beforeEach(function() {
2    browser.url("/Dropdown-Checkboxes-RadioButtons/index.html");
3  })
4
5  describe('Test Selected Dropdown Menus, Checkboxes & Radio Buttons', function() {
6    it('Dropdown item Java is selected therefore should return true', function(done) {
7      option[value='java']
8    });
9
10   it('Dropdown item maven is not selected therefore should return false', function(done) {
11     option[value='maven']
12   });
13
14   it('Option2 is not selected therefore should return false', function(done) {
15     input[value='option-2']
16   });
17
18   it('Option3 is selected therefore should return true', function(done) {
19     input[value='option-3']
20   });
21
22   it('Radio button pumpkin is selected therefore should be true', function(done) {
23     input[value='pumpkin']
24   });
25 });
```

   a. Lines 1-3 simply use a beforeEach hook so that each one of our "it" tests use the URL defined in line 2 (remember this uses the base URL defined in the WDIO.config file + the one defined in the file above)
   b. Line 5 has a describe block defined, which states the overall test we are performing
   c. We then have a number of "it" code blocks that explain what we are trying to test
   d. Finally, we have the locators from the webpage in order to test the elements described in the "it" code block descriptions (e.g. for the first "it" block, we want to test the dropdown menu that has the value of "Java" to see if it isSelected)

> If you need to recap on how to find the locators, please refer to the lecture video. As this has been explained a number of times previously. I have not repeated it in these lecture notes and have instead provided the locators for you.

We continue developing this file in the next lecture video

Lecture 53 – isSelected – Part 2

In this lecture we continue to build our isSelected test from Part 1.

Lecture Resources on Udemy:

- http://v4.webdriver.io/api/state/isSelected.html
- http://www.webdriveruniversity.com/Dropdown-Checkboxes-RadioButtons/index.html

Key Points:

- We now have the skeleton file in place that has a describe block and multiple "it" test blocks for various test scenarios
- We have the locators defined in each of the "it" blocks, so we can interact with the elements that are in scope for the tests
- We now need to build the "it" block tests to use the locators, add assertions and log the output to the console

Instructions (continued from Part 1):

1. Amend the first "it" block to look like this:

```
6  it('Dropdown item Java is selected therefore should return true', function(done) {
7    var isSelected = browser.isSelected("option[value='java']");
8    console.log(isSelected);
9  });
10
```

   a. Here we have added a new variable (isSelected) and assigned it a value using the browser.isSelected function. We have then provided the locator as the parameter.
   b. We then log the value of the variable to the console.

2. Amend the second "it" block to look like this:

```
11 it('Dropdown item maven is not selected therefore should return false', function(done) {
12   var isSelected = browser.isSelected("option[value='maven']");
13   console.log(isSelected);
14 });
15
```

   a. This is the same as the first "it" block but this time we have provided the locator for the second test.

3. Amend the third "it" block to look like this:

```
16 it('Option2 is not selected therefore should return false', function(done) {
17   var isSelected = browser.isSelected("input[value='option-2']");
18   console.log(isSelected);
19 });
20
```

   a. This is the same as the first "it" block but this time we have provided the locator for the third test.

4. Amend the fourth "it" block to look like this:

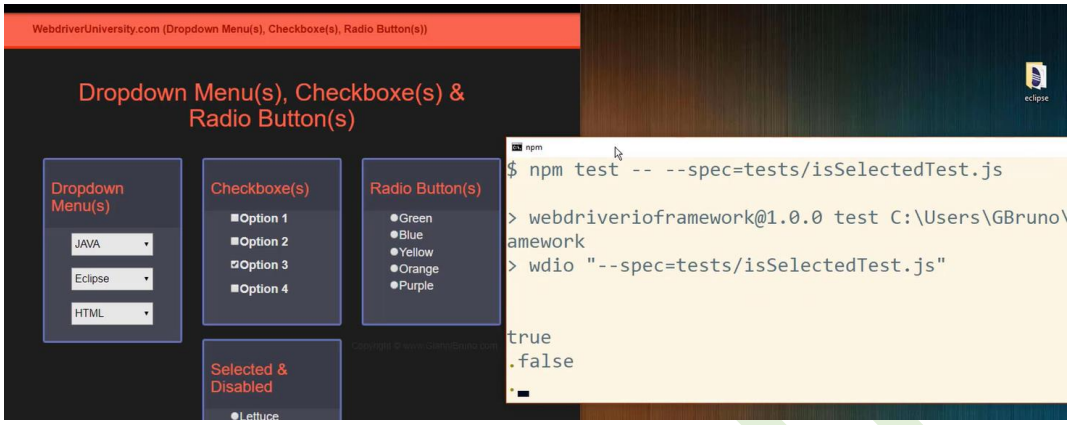
```
21 it('Option3 is selected therefore should return true', function(done) {
22   var isSelected = browser.isSelected("input[value='option-3']");
23   console.log(isSelected);
24 });
25
```

   a. This is the same as the first "it" block but this time we have provided the locator for the fourth test.

5. Finally, amend the last "it" block to look like this:

```
26 it('Radio button pumpkin is selected therefore should be true', function(done) {
27   var isSelected = browser.isSelected("input[value='pumpkin']");
28   console.log(isSelected);
29 });
```

   a. This is the same as the first "it" test but uses the fifth locator

6. Once you have made the changes above, save your file and close Sublime Text
7. Open up GitBash / iTerm2 and navigate to our webdriverioFramework directory
8. Run the following command:
9. **npm test -- -- specs=tests/isEnabledTest.js** + press enter
10. You should see the tests run, as shown below:
11. 

12. Then if we review the console output, we should see:

13. Where 5 tests have run and Passed and also the values outputted are:
    a. True – Because the dropdown menu option “Java” **is** selected
    b. False – Because the dropdown menu option “Maven” **is not** selected
    c. False – Because option 2 of the checkbox **is not** selected
    d. True – Because option 3 of the checkbox **is** selected
    e. True – Because the radio button “Pumpkin” **is** selected

Lecture 54 – isVisibleWithinViewport - Part 1

In this lecture we take a look at the isVisibleWithinViewpoint function, which will return true if the selected DOM-element is found by given selector and is visible within the viewport.

Lecture Resources on Udemy:

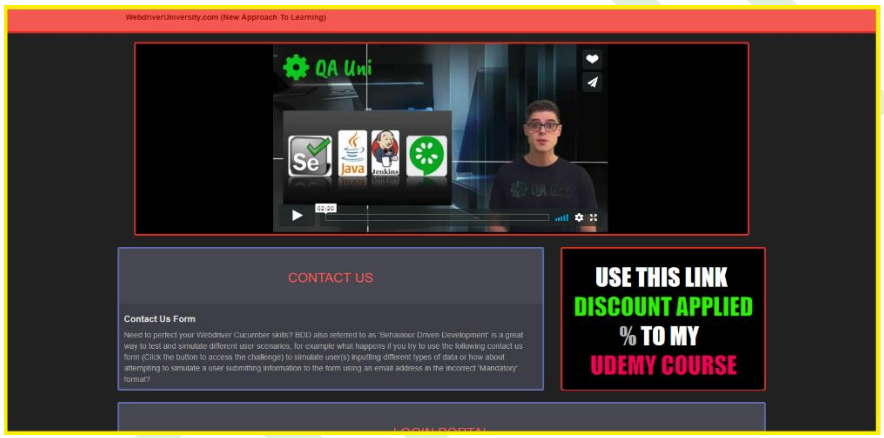
- http://v4.webdriver.io/api/state/isVisibleWithinViewport.html
- http://www.webdriveruniversity.com/Hidden-Elements/index.html

Key Points:

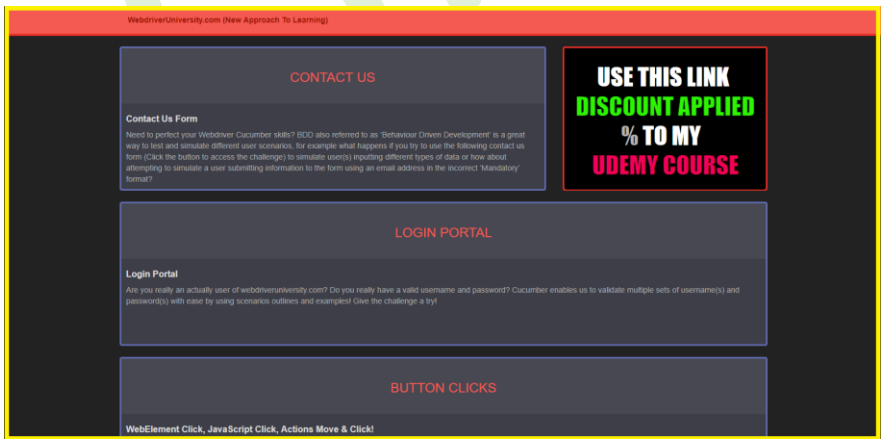
- We construct a test using the isVisibleWithinViewpoint function and do this by using the hidden elements page from webdriveruniversity.com
- The buttons on this page are not visible to the user and fall outside of the viewpoint view, so should return false when using isVisibleWithinViewpoint

What is a viewport?

The viewport is the part of the webpage that the user can currently see. The scrollbars move the viewport to show other parts of the page. So, for example, when a user goes to webdriveruniversity.com, the initial viewpoint is highlighted by the yellow border below:



If the user was to scroll down, then the viewpoint will change:

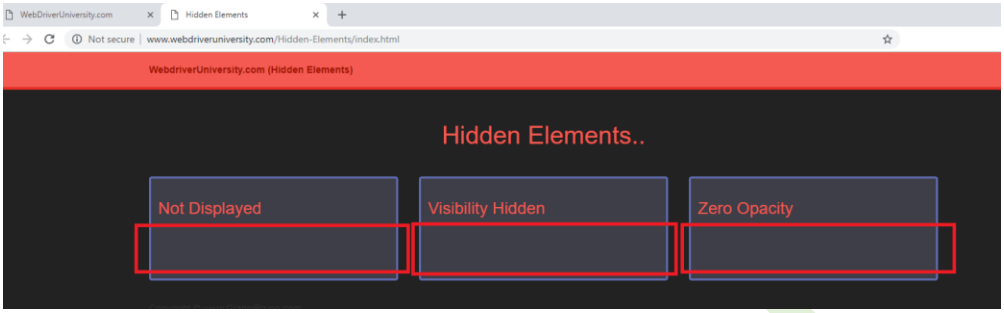


It's basically the view displayed to the user within the browser. It's the part of the page that is visible to them.

Instructions:

How to use isVisibleWithinViewpoint and steps to construct out test

1. We are going to use the hidden-elements webpage from webdriveruniversity.com, specifically the hidden buttons section, as shown:



2. Download the lecture code from the Udemy resource section
3. Save this file to your "tests" folder and make sure it's named viewportTest.js
4. Open the file is Sublime Text, it should look like this:

isSelectedTest.js | viewportTest.js

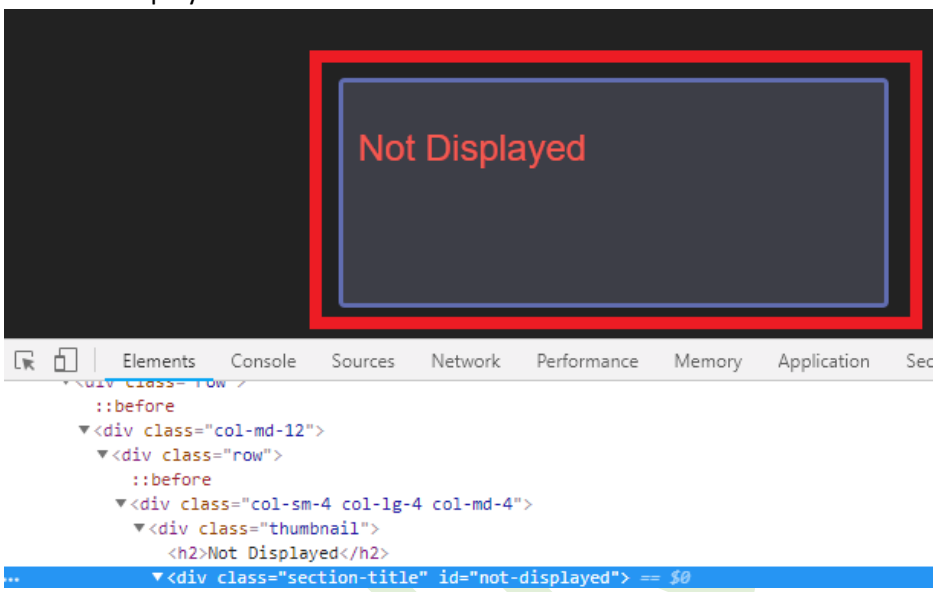
```
1  beforeEach(function() {
2      browser.url("Hidden-Elements/index.html");
3  })
4
5  describe('Test whether specific elements are visible within viewport', function() {
6
7      it('should resize the current viewport', function () {
8          browser.setViewportSize({
9              width: 1200,
10             height: 800
11         })
12         browser.pause(2000);
13     });
14
15     it('should detect if an element is visible', function () {
16         var isVisibleWithinViewport = browser.isVisibleWithinViewport("#not-displayed");
17         console.log(isVisibleWithinViewport); //false
18
19         var isVisibleWithinViewport = browser.isVisibleWithinViewport("#visibility-hidden");
20         console.log(isVisibleWithinViewport); //false
21
22         var isVisibleWithinViewport = browser.isVisibleWithinViewport("#zero-opacity");
23         console.log(isVisibleWithinViewport); //false
24
25         var isVisibleWithinViewport = browser.isVisibleWithinViewport("h1");
26         console.log(isVisibleWithinViewport); //true
27     });
28 });
```

5. Explaining the code above:
    a. **Lines 1 to 3 –** We simply use a beforeEach hook so that each one of our "it" tests use the URL defined in line 2 (remember this uses the base URL defined in the WDIO.config file + the one defined in the file above)
    b. **Line 5 -** has a describe block defined, which states the overall test we are performing
    c. **Lines 7 to 13** -is our first "it" test block and we change the size of the viewport of the browser by setting a specific width and height. We also add a pause to give the webpage sufficient time to load
    d. **Line 15** – is our second "it" test block and we provide a description of what we intend to test ("should detect if an element is visible")

Continued on next page

e. **Lines 16-18** – is our first viewport test. Here we have created a new variable and assigned it a value using browser.isVisivleWithinViewport and then provide the first locator for the first element that we will be testing.

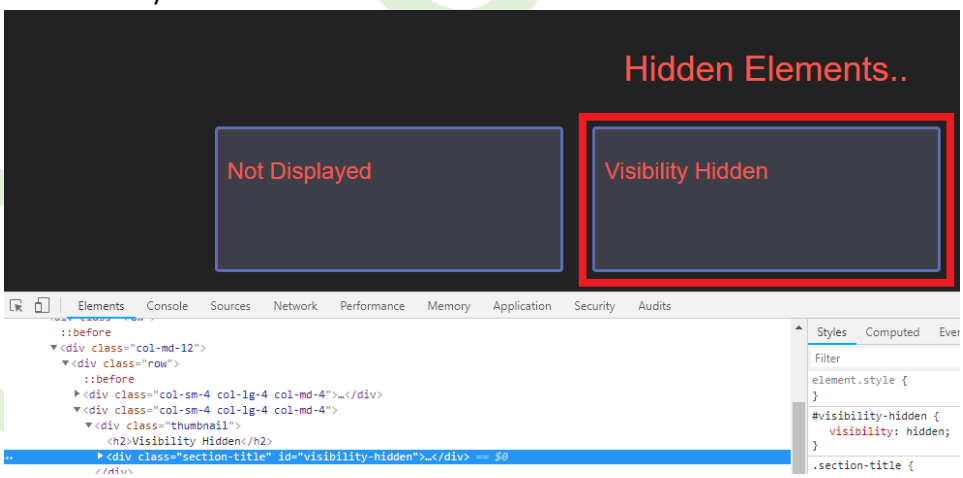
1. #not-displayed is the id if the first button



2. As this button is not visible when the page loads, we expect this test to return **false**

f. **Lines 19-21** – is our second viewport test. Here we have created a new variable and assigned it a value using browser.isVisivleWithinViewport and then provide the second locator for the second element that we will be testing.

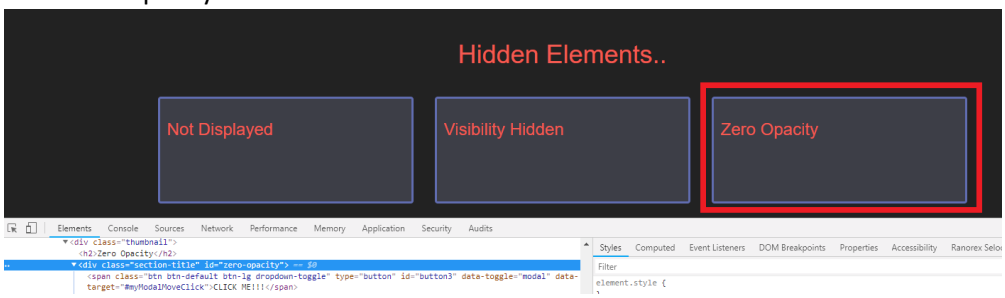
1. #visibility-hidden is the id if the second button



2. As this button is not visible when the page loads, we expect this test to return **false**

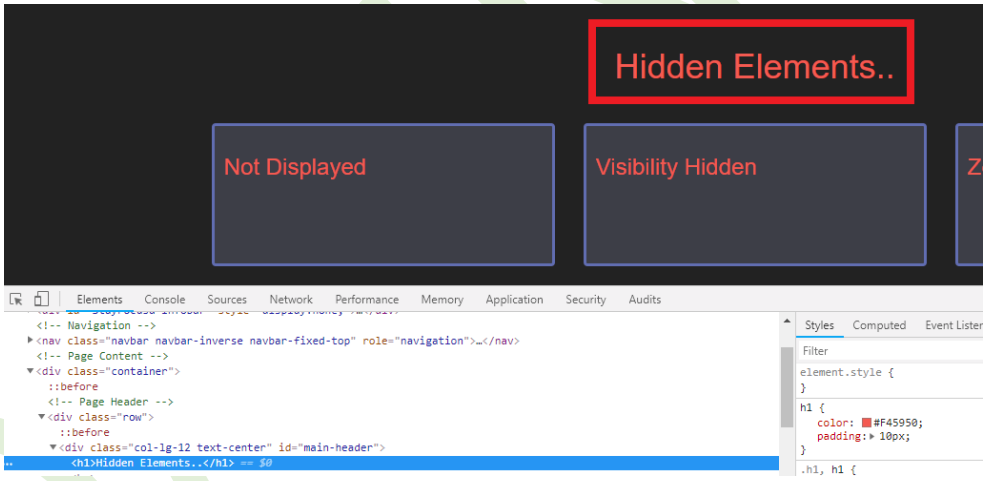
Continued on next page

g. **Lines 22-24** – is our third viewport test. Here we have created a new variable and assigned it a value using browser.isVisivleWithinViewport and then provide the third locator for the third element that we will be testing.
    1. #zero-opacity is the id of the third button



    2. As this button is not visible when the page loads, we expect this test to return **false**

h. **Lines 25-28** – is our fourth viewport test. Here we have created a new variable and assigned it a value using browser.isVisivleWithinViewport and then provide the fourth locator for the fourth element that we will be testing (the h1 header)
    1. We can use the element tag h1 in this test because there is only one h1 defined on the webpage (we do not need to use an id locator)



    2. As the h1 is visible when the page loads, we expect this test to return **true**

We continue with the isVisibleWithinViewport

test in the next lecture (Part 2)

Lecture 55 – isVisibleWithinViewport - Part 2

In this lecture we continue to build our isVisibleWithinViewpoint test from Part 1.

Lecture Resources on Udemy:

- http://v4.webdriver.io/api/state/isVisibleWithinViewport.html
- http://www.webdriveruniversity.com/Hidden-Elements/index.html
- http://v4.webdriver.io/api/protocol/windowHandleSize.html

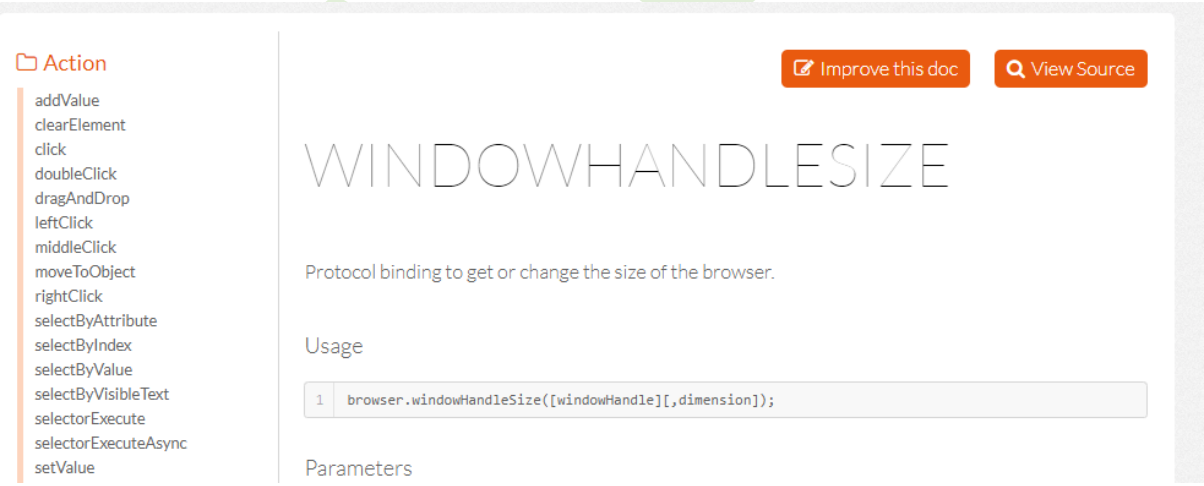
Key Points:

- We now have most of our test code in place
- We intend to test four different elements which are contained within one "it" block
- We have used locators to focus on the elements that we intend to test:
    - #not-displayed – for the first hidden button
    - #visibility-hidden – for the second hidden button
    - #zero-opacity – for the third hidden button
    - h1 – for the header element (for which there is only one h1 element

Explaining the windowHandleSize function

The webdriverio documentation has a number of state functions, one of these being windowHandleSize. We can use this function to change the size of the browser, but we can also use it to output the size of the browser window.

Find out more here: http://v4.webdriver.io/api/protocol/windowHandleSize.html



Instructions (continued from Part 1):

1. Add the following code on Line 14 and 15:

```
14          var windowSize = browser.windowHandleSize();
15          console.log(windowSize.value);
```

    a. Here we are creating a new variable (windowSize) and assigning it a value using browser.windowHandleSize() which will get the window size of the browser.
    b. We then log this output to the console

2. Next add the following lines of code to lines 31-35:

```
31          var width = browser.getViewportSize("width");
32          console.log(width);
33
34          var height = browser.getViewportSize("height");
35          console.log(height);
```

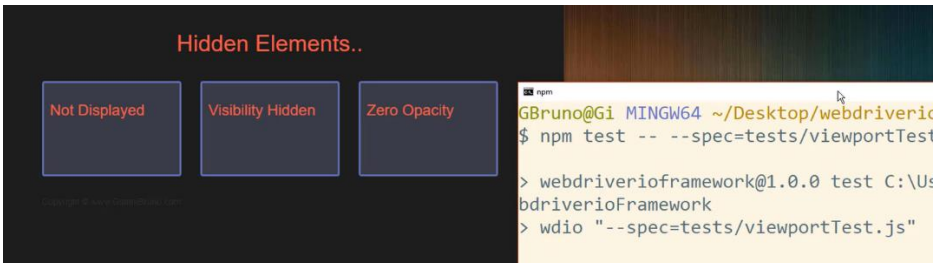
   a. Here we have created two new variables (width and height) and have used the browser.getViewportSize("width") which will return the viewpoint width and browser.getViewportSize("height") which will return the viewpoint height
   b. We then output these values to the console

3. Once you have made the changes above, your overall code should look like:

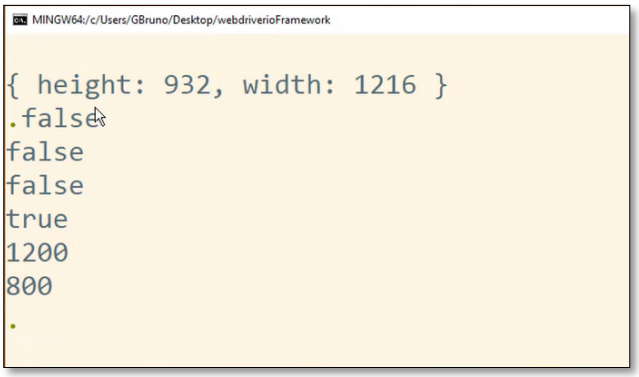
isSelectedTest.js | viewportTest.js

```
1  beforeEach(function() {
2      browser.url("Hidden-Elements/index.html");
3  })
4
5  describe('Test whether specific elements are visible within viewport', function() {
6
7      it('should resize the current viewport', function () {
8          browser.setViewportSize({
9              width: 1200,
10             height: 800
11         })
12         browser.pause(2000);
13
14         var windowSize = browser.windowHandleSize();
15         console.log(windowSize.value);
16     });
17
18     it('should detect if an element is visible', function () {
19         var isVisibleWithinViewport = browser.isVisibleWithinViewport("#not-displayed");
20         console.log(isVisibleWithinViewport); //false
21
22         var isVisibleWithinViewport = browser.isVisibleWithinViewport("#visibility-hidden");
23         console.log(isVisibleWithinViewport); //false
24
25         var isVisibleWithinViewport = browser.isVisibleWithinViewport("#zero-opacity");
26         console.log(isVisibleWithinViewport); //false
27
28         var isVisibleWithinViewport = browser.isVisibleWithinViewport("h1");
29         console.log(isVisibleWithinViewport); //true
30
31         var width = browser.getViewportSize("width");
32         console.log(width);
33
34         var height = browser.getViewportSize("height");
35         console.log(height);
36     });
37 });
```

4. Save the file and close Sublime Text
5. Open up GitBash / iTerm2 and navigate to our webdriverioFramework directory
6. Run the following command:
7. **npm test -- -- specs=tests/ viewportTest.js** + press enter
8. You should see the tests run, as shown below:



9. Then if we review the console output, we should see:



10. Where 2 tests have run and Passed and also the values outputted are:
    a. Height: 932 – This is the window height size of the browser
    b. Width: 1216 – This is the window width size of the browser
    c. False – Because button one is not visible in the viewport
    d. False – Because button two is not visible in the viewport
    e. False – Because button three is not visible in the viewport
    f. True – Because header is visible on the viewport
    g. 1200 – Because this is the width of the viewport (as set earlier in the code)
    h. 800 – Because this is the height of the viewport (as set earlier in the code)

Lecture 56 – getText, isVisible, isExisting - Part 1

In this lecture we go back to our contactUsTest.js test and look to enhance the current file by introducing getText, isVisible and isExisting functions.

Lecture Resources on Udemy:

- http://v4.webdriver.io/api/property/getText.html
- http://v4.webdriver.io/api/state/isVisible.html
- http://v4.webdriver.io/api/state/isExisting.html
- http://www.webdriveruniversity.com/Contact-Us/contactus.html

Our current contactUsTest.js file at this stage:

contactUsTest.js

```
beforeEach(function() {
  browser.url('/Contact-Us/contactus.html');
})

describe('Test Contact Us form WebdriverUni', function() {
  beforeEach(function() {
    console.log('Inside the describe block!');
  })

  it('Should be able to submit a successful submission via contact us form', function(done) {
    browser.setValue("[name='first_name']",'Joe');
    browser.setValue("[name='last_name']",'Blogs');
    browser.setValue("[name='email']", 'joe_blogs@mail.com');
    browser.setValue("[name='message']", 'How much does product x cost?');
    browser.click("[type='submit']");

    #contact_reply h1
    });

  it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
    browser.setValue("[name='first_name']",'Mike');
    browser.setValue("[name='last_name']",'Woods');
    browser.setValue("[name='email']", 'mike_woods@mail.com');
    browser.click("[type='submit']");

    body
    });


  it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
    browser.setValue("[name='first_name']",'Sarah');
    browser.setValue("[name='email']", 'sarah_woods@mail.com');
    browser.click("[type='submit']");

    body
    });

  it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
    browser.setValue("[name='first_name']",'Jim');
    browser.setValue("[name='last_name']",'Jomes');
    browser.click("[type='submit']");

    body
    });
});
```

To briefly explain what we have done above:

- We are testing the contact us form from webdriveruniversity.com by testing a number of form combinations
    - First “it” test – provide all fields to the contact us form
    - Second “it” test – provides all but one (message) field to the form
    - Third “it” test – provides all but two (message and last name) fields to the form
    - Fourth “it” test – only provides the first name and last name to the form

Explaining the functions we will be using to improve our test:

getText:

```
1  browser.getText(selector);
```

- This function allows you to retrieve the text from the locator provided
- For example, say we have an a example text element defined in our webpage.
    - We can use getText using the locator of the id of this element to retrieve the text "example text"

isExisting:

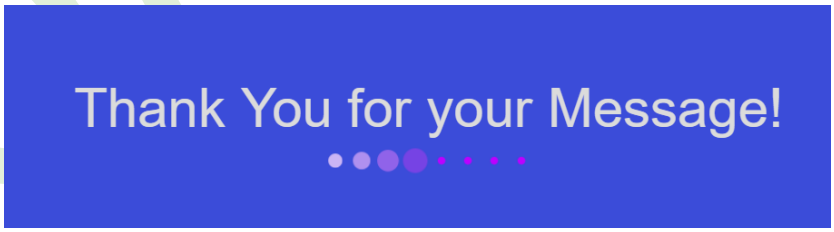
- Please refer to lecture 43 & 44 for a recap (since this has already been covered)

isVisible:

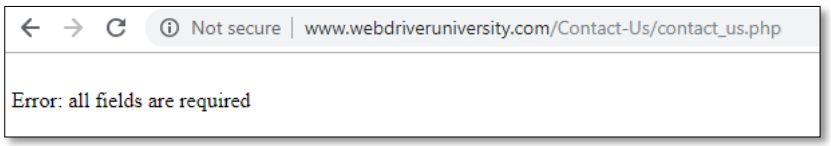
- Please refer to lecture 45 for a recap (since this has already been covered)

What we intend to do:

- We intend to use a combination of getText, isExisting and isVisible inside or test file so enhance it.
- We are going to test a successful and unsuccessful submission using the contact form located here:
    - http://www.webdriveruniversity.com/Contact-Us/contactus.html
    - A successful submission will be when all fields are provided and submitted (since all fields of the form are mandatory)
    - An unsuccessful submission is when a mandatory field is not provided when the form is submitted
- The form displays the following output based on the scenarios above:
    - **Successful**:



    - **Unsuccessful**:



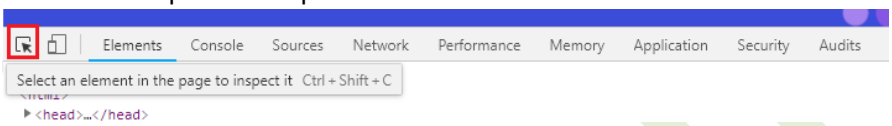
    - The error message can vary depending on what field was not provided

Retrieving the locators for the Successful and Unsuccessful scenarios above:

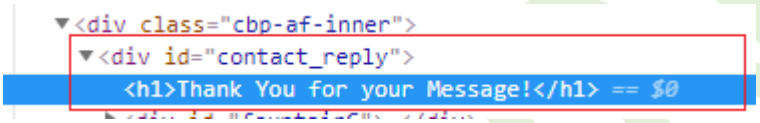
Before we proceed, we need to get locators for a successful submission and an unsuccessful submission. This is because we are going to use this output in our tests (to know whether a test was a successful submission or an unsuccessful submission).

We can retrieve the locators by following these steps:

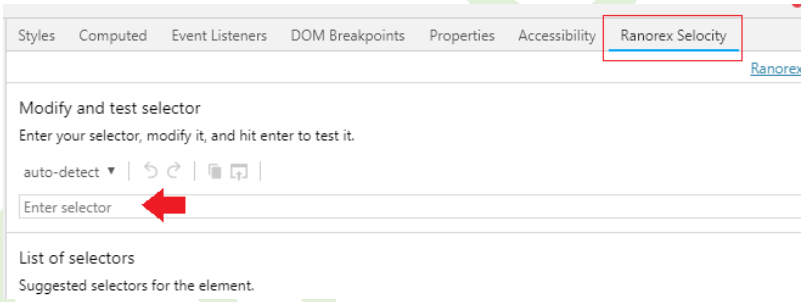
1. Go to the contact us form on webdriveruniversity.com by using the following link:
    a. http://www.webdriveruniversity.com/Contact-Us/contactus.html
2. Fill in the form manually by entering all fields and press the submit button
3. You will be shown the "thank you for your message" page
4. Press F12 to open developer mode and select the selector tool as shown:



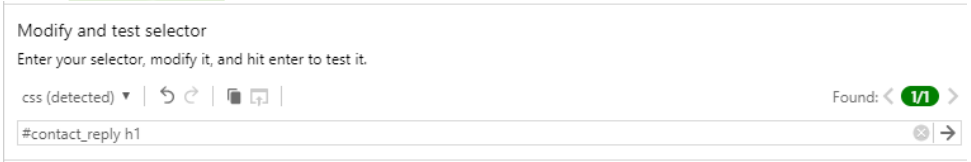
5. Now, hover over the "Thank you for your message" and then left click it
6. The inspector will show you something similar to:



    a. Notice that the message is embedded in a h1 element and the div outside of the h1 element has an id of "contact_reply"
    b. We can use the id and h1 element to form the locator
7. Copy the id "contact_reply" to your clipboard
8. Move to the right of the inspector tool and locate the Ranorex Selocity tab:



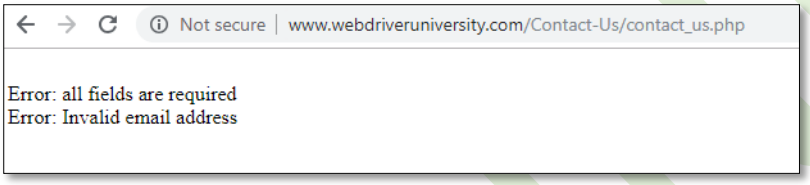
9. Type # and then paste the ID into the area shown by the red arrow above
10. Then add a space and then write h1 and press enter
11. The field should look like this:



    a. Notice how the Found (on the right) has found 1 element
    b. We have told Ranorex to use the div id and then the h1 nested within the div to create a locator
    c. We can now use this to form part of our test
12. Copy the **#contact_reply h1** to your clipboard
13. Open up the contactUsTest.js file from our "tests" folder using Sublime Text
14. Paste the locator in your test file, as shown:

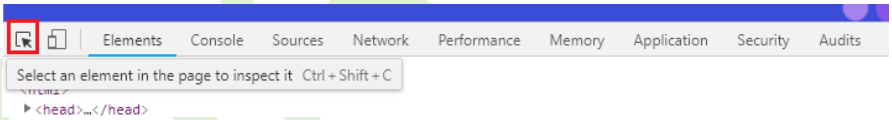
```
10  it('Should be able to submit a successful subm
11    browser.setValue("[name='first_name']",'Joe'
12    browser.setValue("[name='last_name']",'Blogs
13    browser.setValue("[name='email']", 'joe_blog
14    browser.setValue("[name='message']", 'How mu
15    browser.click("[type='submit']");
16
17    #contact_reply h1
18    });
19
```

15. Next, go back to the contact us form on webdriveruniversity.com and fill in the following fields:
    a. First Name
    b. Last Name
    c. Message
    d. Make sure to leave the email address field blank!
    e. And press the submit button
16. You will be shown the following page:

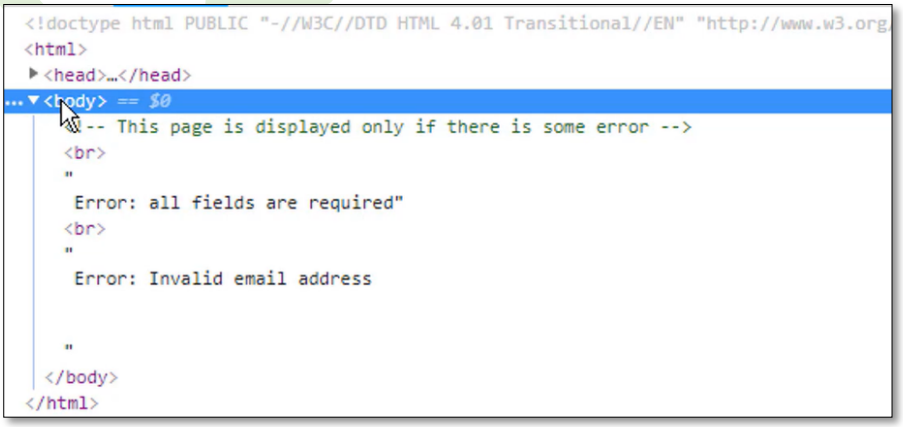


17. Highlight the text, as shown:

18. Then right click and click Inspect
19. Click the selector tool

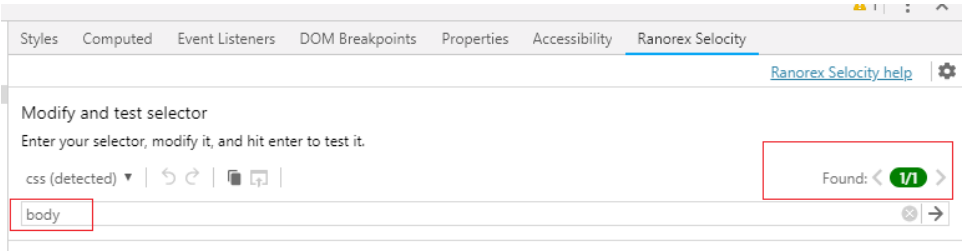


20. Hover over the message and left mouse click it
21. You will see that only the following elements exist on the page:



22. Since only the error message are displayed on this page, we can simply use the <body> tag as the locator

23. We can confirm this by using Ranorex Selocity as shown:



24. Copy the “body” tag to your clipboard
25. Go back to your test code in Sublime Text and paste the locator as shown to all the negative test scenarios (all tests that we expect to fail due to being an unsuccessful submission) as shown:

```
20  it('Should not be able to submit a successful submission via contact us form as all fields are requi
21    browser.setValue("[name='first_name']",'Mike');
22    browser.setValue("[name='last_name']",'Woods');
23    browser.setValue("[name='email']", 'mike_woods@mail.com');
24    browser.click("[type='submit']");
25
26    body
27    });
28
29
30  it('Should not be able to submit a successful submission via contact us form as all fields are requi
31    browser.setValue("[name='first_name']",'Sarah');
32    browser.setValue("[name='email']", 'sarah_woods@mail.com');
33    browser.click("[type='submit']");
34
35    body
36    });
37
38  it('Should not be able to submit a successful submission via contact us form as all fields are requi
39    browser.setValue("[name='first_name']",'Jim');
40    browser.setValue("[name='last_name']",'Jomes');
41    browser.click("[type='submit']");
42
43    body
44    });
45 });
```

26. We now have all the locators we need!

We will continue improving this test in the next lecture

Lecture 57 – getText, isVisible, isExisting - Part 2

In this lecture we continue improving our contactUsTest.js file and will now use the locators to build methods round them to validate the output. We are now going to test to see if a successful form submission has taken place.

Lecture Resources on Udemy:

- http://v4.webdriver.io/api/property/getText.html
- http://v4.webdriver.io/api/state/isVisible.html
- http://v4.webdriver.io/api/state/isExisting.html
- http://www.webdriveruniversity.com/Contact-Us/contactus.html

Key Points:

- We now have relevant locators in place to interact with the contact us webform from webdriveruniversity.com
- We have placed these locators in our contactUsTest.js file (in each of the "it" code blocks)
- We now need to use these locators to build tests for each of the scenarios

Instructions:

Amending the first "it" code block. This is a successful scenario where we expect the form to submit correctly since we are providing all required fields

1. You code for the first "it" block should look like this:

```
10  it('Should be able to submit a successful submission via contact us form', function(done) {
11    browser.setValue("[name='first_name']",'Joe');
12    browser.setValue("[name='last_name']",'Blogs');
13    browser.setValue("[name='email']", 'joe_blogs@mail.com');
14    browser.setValue("[name='message']", 'How much does product x cost?');
15    browser.click("[type='submit']");
16
17    #contact_reply h1
18
19
20    });
21
```

2. Create a new variable called successfulContactConfirmation and the value should equal the browser.isExisting('#contact_reply h1');
3. Then, add an assertion by adding: expect(successfulContactConfirmation, 'Successful submission Message does not exist').to.be.true;
    a. What we are doing here is using isExisting to prove the h1 element exists and the assertion compares what we expect to see (true indicates we think it does exist, so this should return a true value)
4. We then create a new variable called successfulSubmission that has a value of = browser.getText('#contact_reply h1');
5. Then we again create an assertion using expect(successfulSubmission).to.equal('Thank You for your Message!');
    a. What we are doing here is creating a new variable that gets the text from the h1 element (which contains the successful submission message) and we then use an assertion to check the content of the message to what we are expecting.
6. Once you have made the changes above, remove the locator line (#contact_reply h1) as we have not embedded this into our test.

7. You code should now look like the following:

```
it('Should be able to submit a successful submission via contact us form', function(done) {
  browser.setValue("[name='first_name']",'Joe');
  browser.setValue("[name='last_name']",'Blogs');
  browser.setValue("[name='email']", 'joe_blogs@mail.com');
  browser.setValue("[name='message']", 'How much does product x cost?');
  browser.click("[type='submit']");

  var successfulContactConfirmation = browser.isExisting('#contact_reply h1');
  expect(successfulContactConfirmation, 'Successful submission Message does not exist').to.be.true;

  var successfulSubmission = browser.getText('#contact_reply h1');
  expect(successfulSubmission).to.equal('Thank You for your Message!');
  });
```

8. We have now completed the code for the first "it" block.

Amending the second "it" code block. This is a negative scenario where we do not expect a successful form submission because we are missing mandatory fields

1. Your code for your second "it" code block should currently look like this:

```
it('Should not be able to submit a successful submission via contact us form as all fields are requi
  browser.setValue("[name='first_name']",'Mike');
  browser.setValue("[name='last_name']",'Woods');
  browser.setValue("[name='email']", 'mike_woods@mail.com');
  browser.click("[type='submit']");

  body
  });
```

2. Another way to test an unsuccessful submission is to check to see if a success message is **not** returned. So, for example, if we only enter one field on the contact us form and try to submit it, then we wouldn't expect the success message (from the first "it" code block) to return.
3. Copy the following lines from your first "it" test:

```
  var successfulContactConfirmation = browser.isExisting('#contact_reply h1');
  expect(successfulContactConfirmation, 'Successful submission Message does not exist').to.be.false;
```

   a. Notice we change the .to.be.true to to.be.**false**
   b. This is because we do not expect the successful message to be shown if we attempt to submit a form without all mandatory fields
4. These means we do not need to use the "body" locator after all, so delete it.
5. You code for the second "it" code block should look like this:

```
it('Should not be able to submit a successful submission via contact us form as all fields are requi
  browser.setValue("[name='first_name']",'Mike');
  browser.setValue("[name='last_name']",'Woods');
  browser.setValue("[name='email']", 'mike_woods@mail.com');
  browser.click("[type='submit']");

  var successfulContactConfirmation = browser.isExisting('#contact_reply h1');
  expect(successfulContactConfirmation, 'Successful submission Message does not exist').to.be.false;
  });
```

6.
7. We have now completed the code for the second "it" block.

Continued on next page

Amending the third "it" code block. This is a negative scenario where we do not expect a successful form submission because we are missing mandatory fields

1. You code for the third "it" block should look like this:

```
35  it('Should not be able to submit a successful submission via contact us form as all fields are requi
36    browser.setValue("[name='first_name']",'Sarah');
37    browser.setValue("[name='email']", 'sarah_woods@mail.com');
38    browser.click("[type='submit']");
39
40    body
41    });
```

2. As we are again testing a negative scenario (i.e. we do not expect the success message to be shown) it means we can copy the same code from the second "it" test.
    a. We again don't need to use the 'body' locator, since we have thought of a better way to test an unsuccessful submission – so delete it
3. Make the same changes as you've made from the second "it" code block above, so your code now looks like:

```
35  it('Should not be able to submit a successful submission via contact us form as all fields are requi
36    browser.setValue("[name='first_name']",'Sarah');
37    browser.setValue("[name='email']", 'sarah_woods@mail.com');
38    browser.click("[type='submit']");
39
40    var successfulContactConfirmation = browser.isExisting('#contact_reply h1');
41    expect(successfulContactConfirmation, 'Successful submission Message does not exist').to.be.false;
42    });
43
```

4. We have now completed the code for the third "it" block.

Amending the fourth "it" code block. This is a negative scenario where we do not expect a successful form submission because we are missing mandatory fields

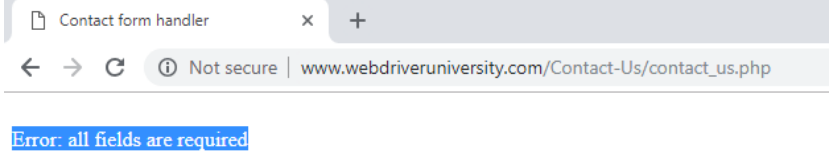
1. You code for the fourth "it" block should look like this:

```
44  it('Should not be able to submit a successful submission via contact us form as all fields are requi
45    browser.setValue("[name='first_name']",'Jim');
46    browser.setValue("[name='last_name']",'Jomes');
47    browser.click("[type='submit']");
48
49    body
50    });
```

2. This time (for good practice) we will be using a different type of assertion so that we do use the 'body' locator
3. Add the following lines of code to your forth "it" block:

```
50    var errorText = browser.getText('body');
51    expect(errorText).to.include('Error: all fields are required');
```

    a. What we are doing here is creating a new variable that uses the 'body' locator from the error page
    b. We then add an assertion that expects the error message from here:



    c. So, if this message is outputted, it means the form failed to submit successfully, because we have ended up on the errors page with the above message showing
4. We then go a step further and add another assertion:

```
53    var errorText = browser.isVisible('body');
54    expect(errorText, 'Error message is not visible').to.be.true;
```

    a. This creates a new variable that uses browser.isVisible where we expect a true of false value to be returned depending if the body locator is visible

b. We then add an assertion where we expect the value to be true because if we are on the error page, then there should be a body element present

5. Now delete the 'body' locator from
6. You fourth "it" test block should now look like this:

```
44  it('Should not be able to submit a successful submission via contact us form as all fields are required',
45    browser.setValue("[name='first_name']",'Jim');
46    browser.setValue("[name='last_name']",'Jomes');
47    browser.click("[type='submit']");
48
49    var errorText = browser.getText('body');
50    expect(errorText).to.include('Error: all fields are required');
51
52    var errorText = browser.isVisible('body');
53    expect(errorText, 'Error message is not visible').to.be.true;
54    });
55 });
```

7. We have now completed the code for the third "it" block.

We will continue improving this test in the next lecture

Lecture 58 – getText, isVisible, isExisting - Part 3

In this lecture we now review our code from the changes made in lecture 54 and 55 and attempt to execute our test.

Lecture Resources on Udemy:

- http://v4.webdriver.io/api/property/getText.html
- http://v4.webdriver.io/api/state/isVisible.html
- http://v4.webdriver.io/api/state/isExisting.html
- http://www.webdriveruniversity.com/Contact-Us/contactus.html

Key Points:

- We have now amended our code to test four scenarios (one successful form submission and three unsuccessful form submissions
- We have used the locators from Lecture 54 to build assertions that check the output with values we are expecting
- We are now ready to review our code before running the test

Instructions:

As we have already gone through most of the changes from the two previous lectures, please refer to the lecture video for a code explanation and recap of the changes made.

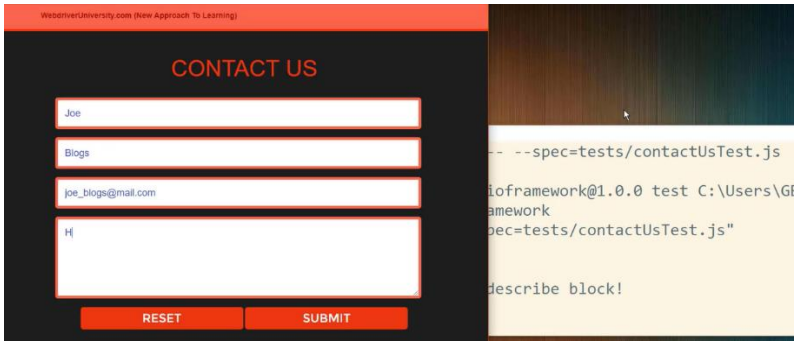
1. We need to make one small change to our contactUsTest.js test file, as shown:

```
48
49      body
50      var errorText = browser.getText('body');
51      expect(errorText).to.include('Error: all fields are required');
52
53      var errorText = browser.isVisble('body');
54      expect(errorText, 'Error message is not visible').to.be.true;
55      });
56  });
```

2. As we have already made use of this locator in our test, we can now remove this from line 49
3. We also misspelt isVisible on line 53, change this to the correct spelling, isVisible
4. Save your file once you've made the changes above
5. Close your file and open GitBash/ iTerm2 and navigate into the webdriverioFramework directory
6. Type the following command:

```
GBruno@Gi MINGW64 ~/Desktop/webdriverioFramework
$ npm test -- --spec=tests/contactUsTest.js
```

7. This will start our test, as shown:



8. You should see that all four tests have passed, as shown:

```
4 passing (14.10s)
```

9. Now to confirm our assertions work, lets make an additional change
10. Open up your contactUsTest.js file using Sublime Text and review the following code:

```
19
20      var successfulSubmission = browser.getText('#contact_reply h1');
21      expect(successfulSubmission).to.equal('Thank You for your Message!');
22      });
23
```

11. We are going to purposely place a fault in our code so that it gets picked up as a failure when our test runs. Change the above code to:

```
20      var successfulSubmission = browser.getText('#contact_reply h1');
21      expect(successfulSubmission).to.equal('Thanks');
22      });
```

12. This will error because the expected text from the locator #contact_reply h1 should equal "Thank You for your Message!" not "Thanks".
13. Save your file and rerun your test in GitBash/ iTerm2 as shown:

```
GBruno@Gi MINGW64 ~/Desktop/webdriverioFramework
$ npm test -- --spec=tests/contactUsTest.js
```

14. You should see the test run and the following output:

```
1 failing

1) Test Contact Us form WebdriverUni Should be able to subm
t a successful submission via contact us form:
expected 'Thank You for your Message!' to equal 'Thanks'
running chrome
AssertionError: expected 'Thank You for your Message!' to e
ual 'Thanks'
```

15. As you can see, a failure has been reported and the reason is due to the expected text being different to that which was returned from the test

Make sure to change your file back once you've attempted the failure above!

Lecture 59 – waitForText Part 1

In this lecture we take a look at the waitForText function. We can use this function to wait for an element (when given a CSS selector) for a given amount of time in milliseconds to have text or content.

Lecture Resources on Udemy:

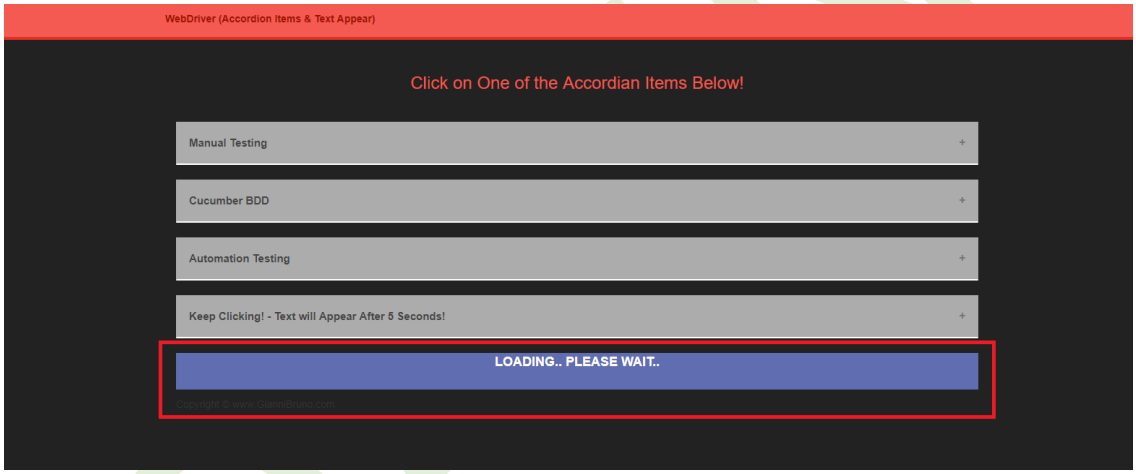
- http://v4.webdriver.io/api/utility/waitForText.html
- http://www.webdriveruniversity.com/Accordion/index.html
- http://v4.webdriver.io/api/utility/$.html#Usage

Key Points:

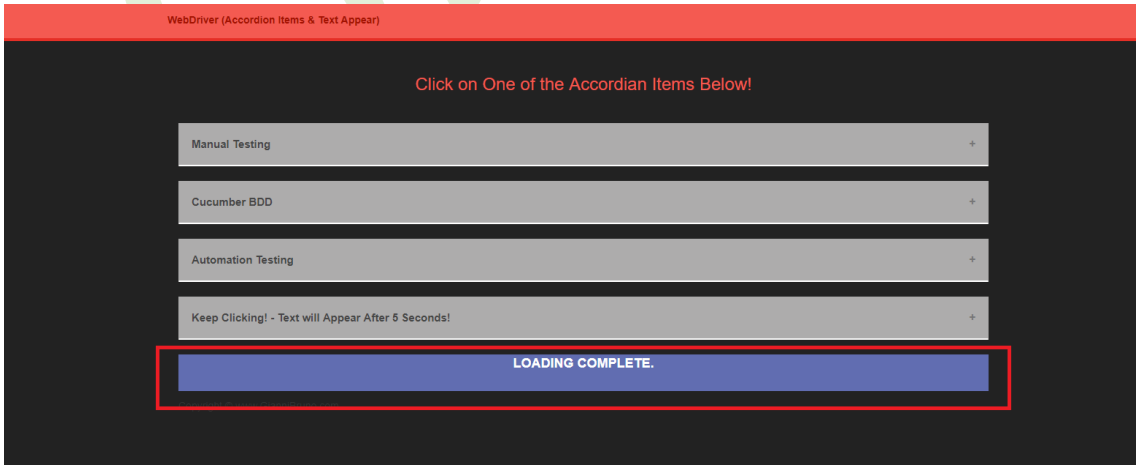
- We use the waitForText function in this test
- We use this in combination with the accordion page from webdriveruniversity.com

Details about this test:

We shall be using the accordion webpage from webdriveruniversity.com (see link above and image below):



Notice how after a period of time, the "Loading… Please Wait" section changes to the below:



Once loading is complete, the wording changes to "Loading Complete."

This is a perfect scenario to use waitForText as we can look for the "Loading Complete" wording.

Instructions:

Creating a waitForText test

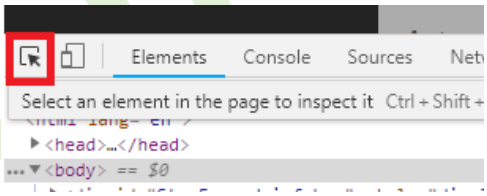
1. Download the skeleton test file from the Udemy resources section of this lecture
2. The file should look like:

```
waitForText.js
1  beforeEach(function(){
2      browser.url("/Accordion/index.html");
3  })
4  describe('Verify that the correct text appears on the accordion page', function() {
5      it('Verify loading complete text is visible after a set duration of time', function() {
6
7      });
8  });
9
10
11
```

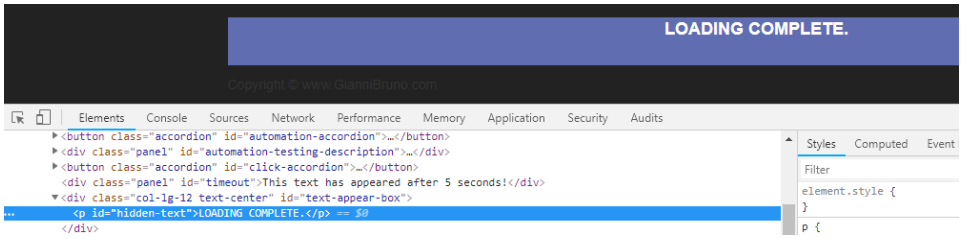
   a. We have used a beforeEach hook so that our browser directs to the accordion page (don't forget this uses the base url from the WDIO.config file + the URL defined above)
   b. We have then created a describe that explains what we are trying to do in the overall test
   c. We then have an "it" test block with a description of "verify loading complete text is…"
3. Save the file with a file name of waitForText.js and save in into your "tests" folder
4. If not already open, open the file using Sublime Text
5. The first thing we need to do is increase the timeout time that's set in the WDIO.config page. We can do this by adding the following code which overrides it to 20 seconds:

```
6      this.timeout(20000);
```

6. We now need to identify a locator that we can use to test the "Loading Complete." text from the accordion webpage
   a. Go to http://www.webdriveruniversity.com/Accordion/index.html
   b. Press F12 on your keyboard to open the Inspector tool
   c. Click on the element selector tool:



   d. Then hover over and select the "Loading Complete." Element, as shown:



   e. Notice that the element has a unique id of "hidden-text"
      i. We can use this as the locator to test this element
   f. Copy "hidden-text" to your clipboard
   g. Then go back to your test file in Sublime Text

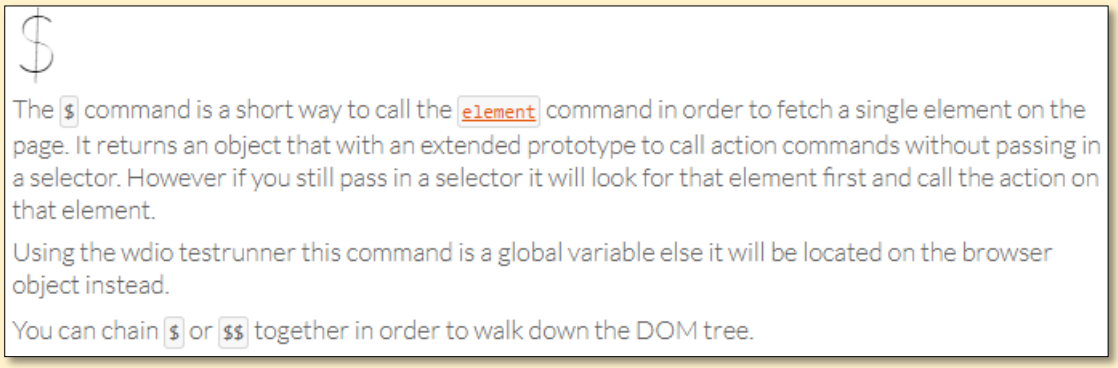
7. Paste the locator into your file like shown:

```
1  beforeEach(function() {
2      browser.url("/Accordion/index.html");
3  })
4  describe('Verify the that correct text appears on the accordion page', function() {
5    it('Verify loading complete text is visible after a set duration of time', function() {
6      this.timeout(20000);
7      hidden-text
8
9    });
10 });
```

8. Add a "#" in front of the hidden-text wording because it's an id and these are identified by a hash symbol at the front

TIP!

There is a quicker way to refer to selectors in a test file. We can use the $ symbol and $$.



$

The `$` command is a short way to call the `element` command in order to fetch a single element on the page. It returns an object that with an extended prototype to call action commands without passing in a selector. However if you still pass in a selector it will look for that element first and call the action on that element.

Using the wdio testrunner this command is a global variable else it will be located on the browser object instead.

You can chain `$` or `$$` together in order to walk down the DOM tree.

```
1  $(selector);
```

Parameters

Param	Type	Details
selector	String	selector to fetch a certain element

Example

index.html

```
1  <ul id="menu">
2      <li><a href="/">Home</a></li>
3      <li><a href="/">Developer Guide</a></li>
4      <li><a href="/">API</a></li>
5      <li><a href="/">Contribute</a></li>
6  </ul>
```

$.js

```
1  it('should get text from a menu link', function () {
2      var text = $('#menu');
3
4      console.log(text.$$('li')[2].$('a').getText()); // outputs: "API"
5      // same as
6      console.log(text.$$('li')[2].getText('a'));
7  });
```

9. Now we are going to use the shortcut above by using a single $
10. Create a new variable called elem and give is a value of = $('#hidden-text');

waitForText.js

```
1  beforeEach(function(){
2      browser.url("/Accordion/index.html");
3  })
4  describe('Verify that the correct text appears on the accordion page', function() {
5      it('Verify loading complete text is visible after a set duration of time', function() {
6          this.timeout(20000);
7          var elem = $('#hidden-text');
8      });
9  });
10
11
12
```

We will continue building this test in the next lecture

Lecture 60 – waitForText Part 2

In this lecture we continue building our waitForText test from Part1.

Lecture Resources on Udemy:

- http://v4.webdriver.io/api/utility/waitForText.html
- http://www.webdriveruniversity.com/Accordion/index.html
- http://v4.webdriver.io/api/utility/$.html#Usage

Key Points:

- We now have the basic test file in place
- We have gone through the $ and $$ shortcuts that can be used to select locators more efficiently
- We have identified the selector needed to identify the "Loading Complete." text

Details about the next stage:

We are going to use a while loop in the following few steps. For those of you unaware of a while loop, it can be explained as a looping condition that checks to see if a condition is met before proceeding.

So, for example, say we have a variable called number that has a value of 1. We can use a while loop to see if the variable is less than 10. If it's less than 10 then we add 1 to the variable after each iteration. Once 10 is reached the condition will move onto the next instruction.

In our test, we are going to use a while loop to keep checking the accordion text until the value "LOADING COMPLETE." appears on the webpage. Once this text appears, it means loading has completed and our condition has been met.

Instructions:

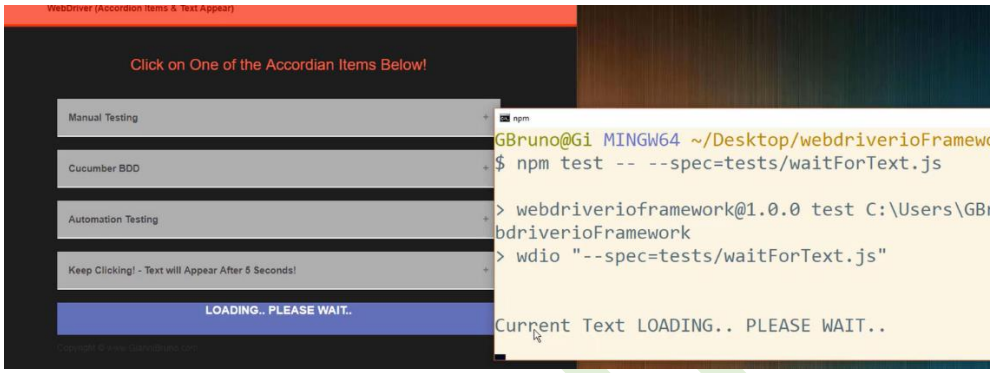
1. Amend your test file to have the following code:

waitForText.js

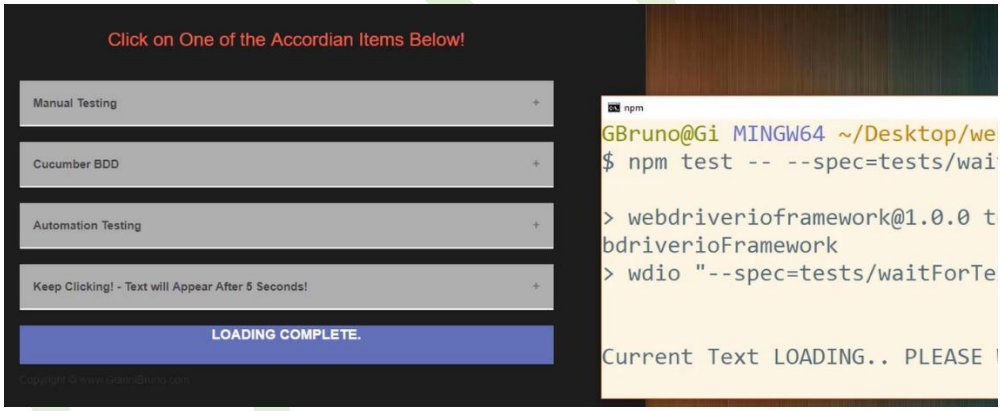
```
1  beforeEach(function(){
2      browser.url("/Accordion/index.html");
3  })
4  describe('Verify that the correct text appears on the accordion page', function() {
5      it('Verify loading complete text is visible after a set duration of time', function() {
6          this.timeout(20000);
7          var elem = $('#hidden-text');
8          console.log("Current Text " + elem.getText());
9          elem.waitForText(1000);
10
11         while(elem.getText() != 'LOADING COMPLETE.') {
12             browser.pause(1000);
13         }
14         console.log(elem.getText);
15     });
16 });
17
18
19
```

2. On Line 8 – we are writing the text of the #hidden-text element to the console
3. On Line 9 – we are instructing the waitForText function to wait 1 second (otherwise it will continuously check)

4. On line 11-15 – we are using a while loop to check the variable (elem) to getText and if the text from the variable doesn't equal "LOADING COMPLETE." It means that the element hasn't finished loading
    a. We then command the loop to wait for a second before checking again
5. Once the text does equal "LOADING COMPLETE.", we then log the elem text (using the elem.getText() function) and print this to the console
6. Once you have made the changes above, save your file
7. Open up GitBash / iTerm2 and navigate to our webdriverioFramework directory
8. Run the following command:
9. **npm test -- -- specs=tests/ waitForText.js** + press enter
10. You should see the tests run, as shown below:



11. Look at the console and notice the LOADING… PLEASE WAIT… has been printed
12. After a little longer, notice the button change to "Loading Complete."



13. Then the console updates almost instantly to:

14. This shows our test has passed and the last log to the console is the wording "LOADING COMPLETE.". This proves we can use the waitForText command to wait for text to appear before proceeding!

Lecture 61 – waitForExist, waitForVisible Part 1

In this lecture we take a look at the waitforExist and waitForVisible functions from the webdriverio documentation.

waitForExist can be used to wait for an element (using a locator) for a provided amount of time in milliseconds for the element to be present within the DOM. The value returned will be true if the found. There is also a reverse flag that can be used that will instead return true if the element does not exist.

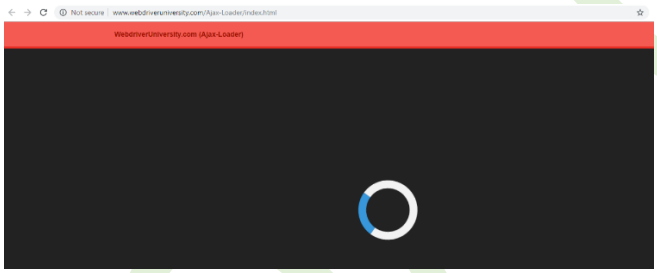
waitForVisible can be used to wait for an element (using a selector) for an amount of time in milliseconds for an element to be visible on the page. The value returned will be true. There is also a reverse flag that can be used return true if the element does not become visible in the given amount of time.

Lecture Resources on Udemy:

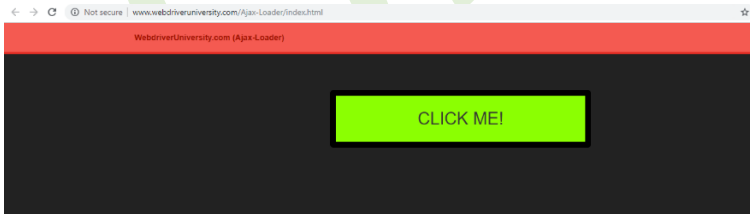
- http://v4.webdriver.io/api/utility/waitForExist.html
- http://v4.webdriver.io/api/utility/waitForVisible.html
- http://www.webdriveruniversity.com/Ajax-Loader/index.html

Details about the waitforExist Test:

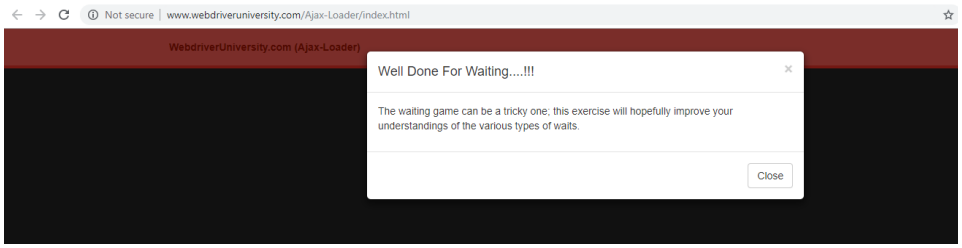
We will be using the Ajax loader exercise from webdriveruniversity in this test (found here: http://www.webdriveruniversity.com/Ajax-Loader/index.html)



When you visit this page, you will notice there is a loader element that circulates for a few seconds.



Then a button eventually appears.



Then, once you click the button, a prompt message is shown to the user.

We can test this scenario using the waitForExist function. We can wait for the loader to complete loading before checking to see if the “click me” button appears.

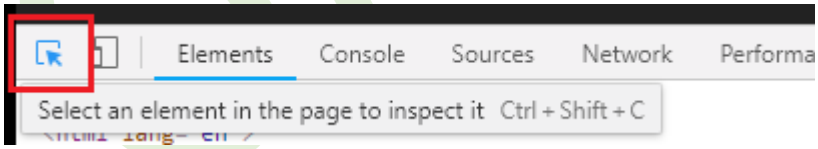
Instructions:

Creating the waitForExist test using the Ajax page

1. Download the skeleton code from the Udemy resource section
2. The code should look like this:

```
waitForExist_waitForVisible_PRE.js
1  beforeEach(function() {
2    browser.url("/Ajax-Loader/index.html");
3  })
4
5  describe('Validate whether the click me button is visible', function() {
6    it('Validate whether the click me button code is visible within the DOM', function(done) {
7
8    });
9
10   it('Validate whether the click me button appears once the Ajax loader fully loads', function(done) {
11   });
12
13 });
14
15
16
```

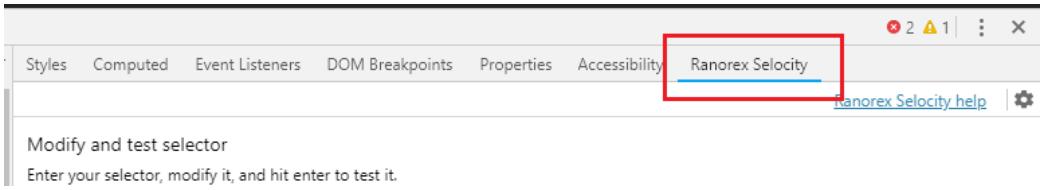
   a. Here we have created a forEach hook that directs each of our tests the Ajax loader page on webdriveruniversity.com
   b. Then, we have created a “describe” block that provides a description of the overall test we are trying to perform
   c. We then have two “it” blocks with a short description of what we are looking to achieve in each test
3. Now we need to identify a locator that we can use for this test
4. Go to http://www.webdriveruniversity.com/Ajax-Loader/index.html
5. Wait for the “CLICK ME!” button to appear
6. Then go into inspector mode (F12)
7. Click the selector tool



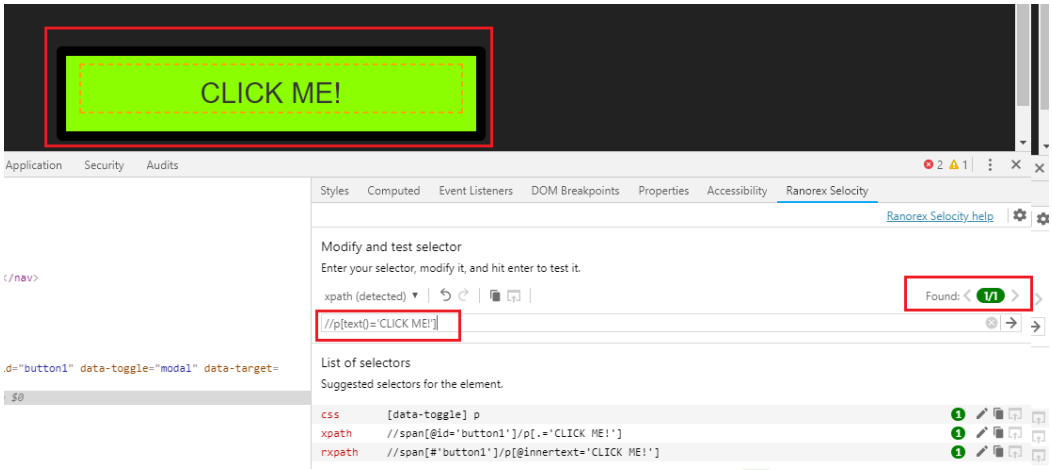
8. Then select the “CLICK ME!” button
9. You’ll see the text is positioned inside a <p> tag

```
<p>CLICK ME!</p> == $0
```

10. Now go to the right-hand side of the inspector panel and click the Ranorex Selocity tab:



11. We need to create a unique X-path by typing the following:



12. This does the following:
    a. It looks for any P tag that has text of “CLICK ME!” in the DOM and when we press enter, it creates an X-Path locator which is confirmed by the dotted orange line around the element.
    b. Also notice on the right-hand side that we have 1/1 reported. This means there is only one element that meets the criteria
13. Copy the locator to your clipboard (//p[text()='CLICK ME!'])
14. And place it in your test file, like so:

```
1  beforeEach(function() {
2    browser.url("/Ajax-Loader/index.html");
3  })
4
5  describe('Validate whether the click me button is visible', function() {
6    it('Validate whether the click me button code is visible within the DOM', function(done) {
7      //p[text()='CLICK ME!']
8    });
9
10   it('Validate whether the click me button appears once the Ajax loader fully loads', function(done) {
11   });
12
13 });
```

15. Now add the following into your code:

```
1  beforeEach(function() {
2    browser.url("/Ajax-Loader/index.html");
3  })
4
5  describe('Validate whether the click me button is visible', function() {
6    it('Validate whether the click me button code is visible within the DOM', function(done) {
7      //p[text()='CLICK ME!']
8      this.timeout(20000);
9      var clickMeButton = "//p[text()='CLICK ME!']";
10     browser.waitForExist(clickMeButton, 8000, false);
11   });
12
13   it('Validate whether the click me button appears once the Ajax loader fully loads', function(done) {
14   });
15
16 });
```

16. We are doing the following
    a. We are increasing the default timeout to 20 seconds
    b. We then create a new variable that has a value of the locator
    c. We then use the waitForExist method and pass the variable as the first argument, then we provide a time of 8000 (8 seconds) where we want to wait 8 seconds for the element to appear, then we provide a *false* value as the last argument because we want to wait until the element appears on the DOM (if we wait for a *true* value, it will wait for 8 seconds until it is not visible)

Lecture 62 – waitForExist, waitForVisible Part 2

In this lecture we continue from the last lecture and further develop the waitForExist and waitForVisible test file.

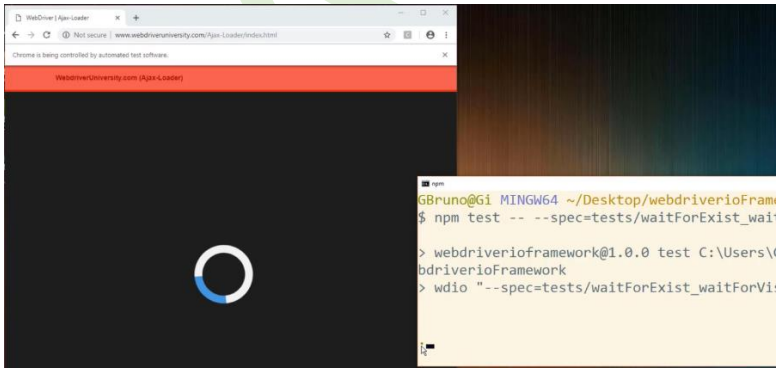
Lecture Resources on Udemy:

- http://v4.webdriver.io/api/utility/waitForExist.html
- http://v4.webdriver.io/api/utility/waitForVisible.html
- http://www.webdriveruniversity.com/Ajax-Loader/index.html

1. Now we are going to start adding to the second "it" block.
2. Here we are going to validate the click me button appears once the Ajax loader fully loads
3. Add the following code to your second "it" block:

```
12  it('Validate whether the click me button appears once the Ajax loader fully loads', function(done) {
13      this.timeout(20000);
14      var clickMeButton = "//p[text()='CLICK ME!']";
15      browser.waitForVisible(clickMeButton, 8000, false);
16  });
```

   a. Here we are overriding the default timeout to 20 seconds
   b. We have created a new variable and have assigned in a value of the locator
   c. Then we use waitForVisible to check to see if the element is visible after 8 seconds (shown as 8000). Finally, we set the flag to false (if we set this to true it means we do not want to button to be visible in the 8 seconds set).
4. Once you have made the changes above, save you file to your "tests" folder and give the file a name of waitForExist_waitForVisible.js and then close Sublime Text
5. Open up GitBash / iTerm2 and navigate to our webdriverioFramework directory
6. Run the following command:
7. **npm test -- -- specs=tests/waitForExist_waitForVisible.js** + press enter
8. You should see the tests run, as shown below:



9. Our test is now waiting for the "LOADING COMPLETE!" text to appear
   a. Notice how a single green dot appears. This indicates the first "it" test has passed

10. After a few more seconds, the second test finishes and reports both tests have passed
    a. Notice there are now two green dots, indicated both tests completed.

Lecture 63 – waitUntil

In this lecture we look at the waitUntil command. The waitUntil command allows you to wait on something. It expects a condition and waits until that condition is fulfilled with a truthy value.

Lecture Resources on Udemy:

- http://v4.webdriver.io/api/utility/waitUntil.html
- http://www.webdriveruniversity.com/Accordion/index.html

Key Points:

- This command uses the following structure:
  - browser.waitUntil(condition[,timeout][,timeoutMsg][,interval]);
  - Where:
    - condition = The condition to wait on
    - timeout = timeout in ms (default: 500)
    - timeoutMsg = error message to throw when waitUntil times out
    - interval = interval between condition checks (default: 500)
- We are going to test the accordion page from webdriveruniversity.com by waiting until the "LOADING.. PLEASE WAIT.." message turns to "LOADING COMPLETE."

Instructions:

1. Download the code example from the Udemy resources section associated to this lecture
2. You will see a skeleton code file that contains the following:

```
waitUntilTest_first.js
1  beforeEach(function() {
2      browser.url("/Accordion/index.html");
3  })
4
5  describe('Validate the loading functionality works correctly', function() {
6      it('Verify relevant text LOADING COMPLETE appears after a period of time', function () {
7
8      });
9  });
10
11
```

   a. Here we are using a forEach hook to direct us to the correct URL (don't forget this uses the base page from the WDIO.config file and extends the URL with that written above)
   b. We have then created a describe block which explains to overall test
   c. We then have a "it" code block which explains what we are testing in this individual test case

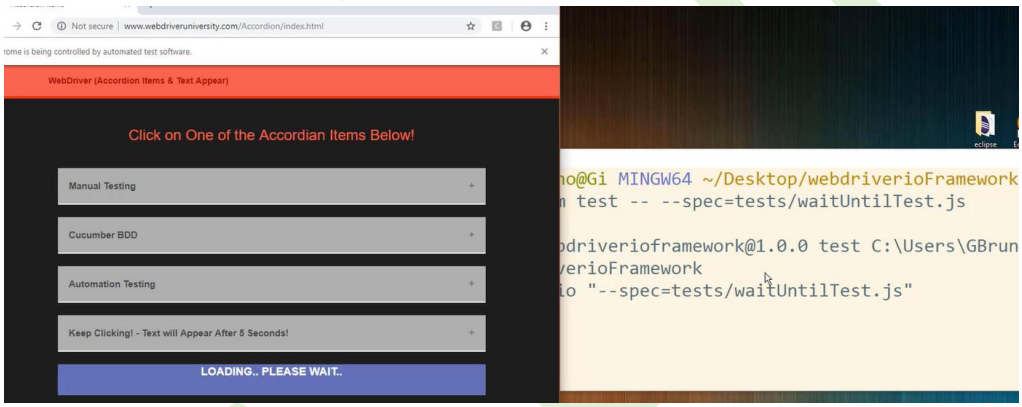
3. Add the following code (I explain what it does below):

```
1  beforeEach(function() {
2      browser.url("/Accordion/index.html");
3  })
4
5  describe('Validate the loading functionality works correctly', function() {
6      it('Verify relevant text LOADING COMPLETE appears after a period of time', function () {
7          this.timeout(20000);
8          browser.waitUntil(function () {
9              return browser.getText('#hidden-text') === 'LOADING COMPLETE.'
10         console.log(browser.getText('#hidden-text'));
11         }, 12000, 'expected text to be different!');
12     });
13 });
```

   a. First, we have added a timeout statement which overrides the default timeout set in the WDIO.config file

b. We then use the waitUntil command using **browser.waitUntil(function() {** which then has code within this function that contains the actual logic that we are waiting to meet
c. We then use *return* where we are using browser.getText and then providing the locator reference and then check to see if the equals “LOADING COMPLETE”
d. We then log the text to the console
e. We wait 12 seconds (shown as 12000) and provide an error message of “expected text to be different” meaning, if the doesn’t equal “LOADING COMPLETE” after 8 seconds then output the error message.

4. Once you have made the changes above, save you file to your “tests” folder and give the file a name of waitUntilTest.js and then close Sublime Text
5. Open up GitBash / iTerm2 and navigate to our webdriverioFramework directory
6. Run the following command:
7. **npm test -- -- specs=tests/ waitUntilTest.js** + press enter
8. You should see the tests run, as shown below:



9. Then when the text changes, you should see the following in the console:

10. Where it shows one test has passed and the output above is only written to the console when the text from webdriveruniversity.com changes to “LOADING COMPLETE!”.

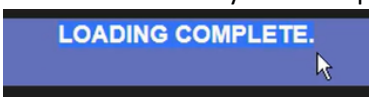
Lecture 64 – waitForValue

In this lecture we look at the waitForValue command. WaitForValue waits for an element (using a locator) to have a value within the amount of time set in milliseconds.

Lecture Resources on Udemy:

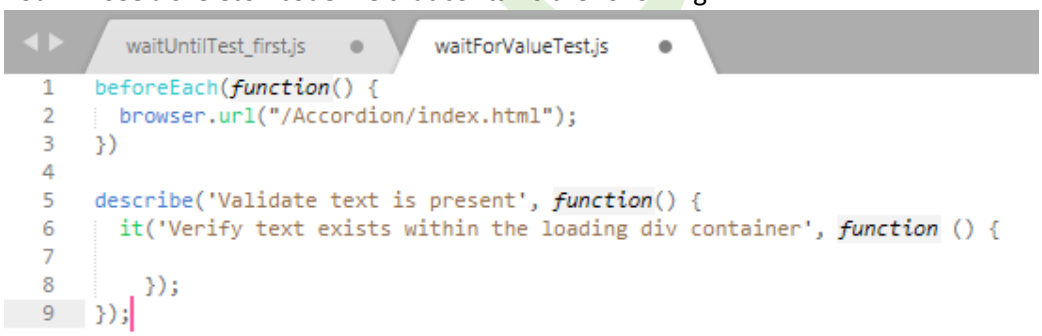
- http://v4.webdriver.io/api/utility/waitForValue.html
- http://www.webdriveruniversity.com/Accordion/index.html

Key Points:

- This command uses the following structure:
  - browser.waitForValue(selector[,ms][,reverse]);
    - where selector is the element that we are waiting on
    - ms is the time in milliseconds
    - reverse is if we want to set a reverse flag
- We are going to check the following div container (from the accordion webdriveruniversity.com webpage) to see if it contains any text:
- 

Instructions:

1. Download the code example from the Udemy resources section associated to this lecture
2. You will see a skeleton code file that contains the following:



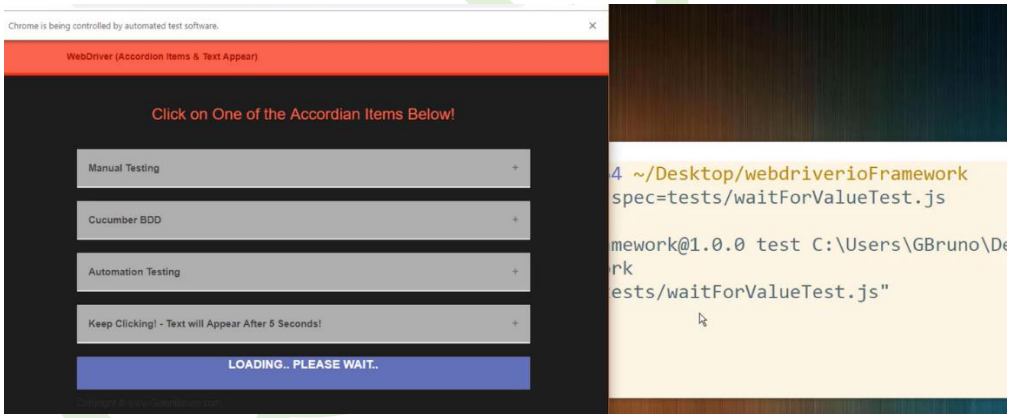
   a. Here we are using a forEach hook to direct us to the correct URL (don't forget this uses the base page from the WDIO.config file and extends the URL with that written above)
   b. We have then created a describe block which explains to overall test
   c. We then have a "it" code block which explains what we are testing in this individual test case

3. Add the following code (I explain what it does below):

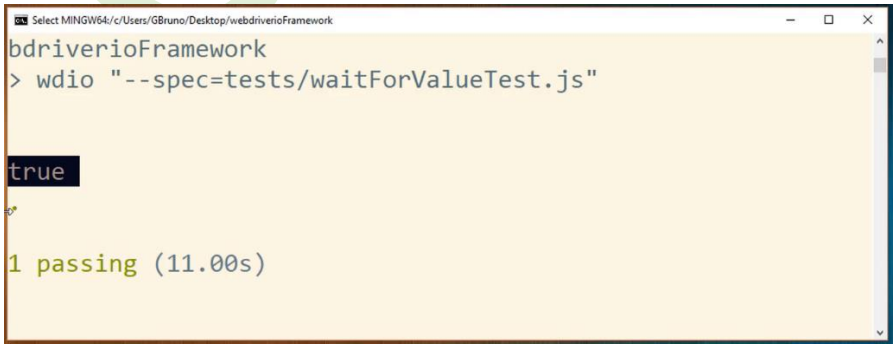
```
beforeEach(function() {
  browser.url("/Accordion/index.html");
})

describe('Validate text is present', function() {
  it('Verify text exists within the loading div container', function () {
    this.timeout(20000);

    var text = browser.waitForValue('#hidden-text', 2000);
    console.log(text);
    });
});
```

   a. First, we have added a timeout statement which overrides the default timeout set in the WDIO.config file
   b. We then create a new variable called text and assign it a value
   c. The value consists of browser.waitForValue and we pass the following arguments, #hidden-text which is the locator (id of the LOADING COMPLETE div) and we set a 2000 (two second) timer to wait for the hidden-text element to load
   d. We then log the value of the variable to the console

4. Once you have made the changes above, save you file to your "tests" folder and give the file a name of waitForValue.js and then close Sublime Text
5. Open up GitBash / iTerm2 and navigate to our webdriverioFramework directory
6. Run the following command:
7. **npm test -- -- specs=tests/waitForValue.js** + press enter
8. You should see the tests run, as shown below:



11. Then when the text changes, you should see the following in the console:



12. As you can see, the test has passed, and the value returned back is "true" indicating that the value of the div element does contain text.

Module 18 – Using External Data (Sync Data Mode)

In this module, we look at different ways we can supply our tests data.

Lecture 65 - Using External Data (Sync Data Mode) - Part 1

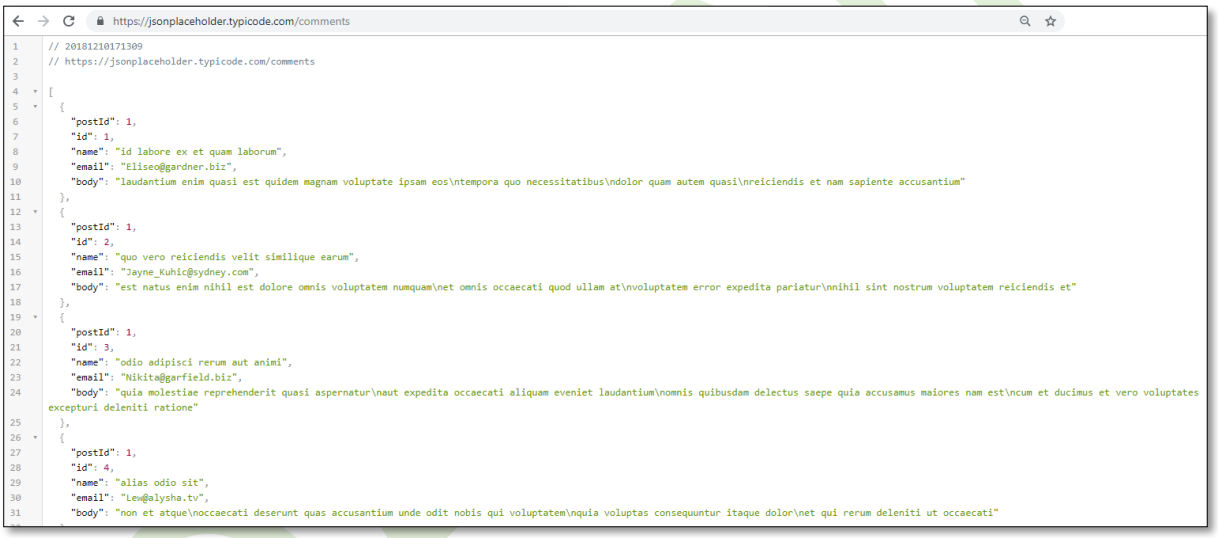
In this lecture we take a look at Sync-Request which is a npm package than can be used to GET data from a given URL to retrieve data that we can then pull into our tests, rather than having to define specific values (such as testEmail@email.com).

Lecture Resources on Udemy:

- https://www.npmjs.com/package/sync-request
- https://jsonplaceholder.typicode.com/comments

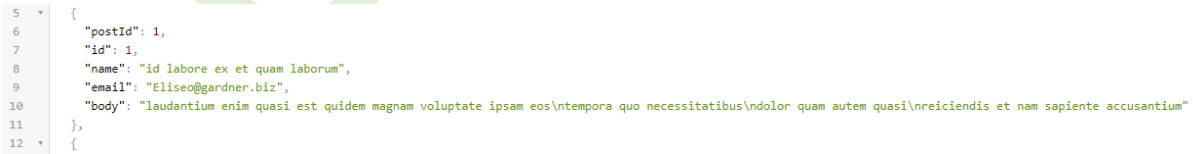
Details about this test:

Take a look at the following JSON file (which can be found here: https://jsonplaceholder.typicode.com/comments)



Notice that each object (shown by each {} block above) has an attribute : value pair. For example, the id attribute has a numeric value of 1, 2, 3 based on the object it's associated to.

If we take a closer look at the first object:



You can see there are five attributes associated to this object and each one of these attributes has a value set (e.g. the first object has an attribute of "name" and a value of "id labore ex et quam laborum".

We can use data housed within a JSON file to feed into our test cases. So, for example, if we were testing the contact us form on the webdriveruniversity.com website, we could attempt to test the form using values parsed from the JSON file rather than having to define values for each individual field within the code itself.

If we take a look back at our contactUs.js file, it looks like this:

contactUsTest.js

```
1  var request = require('sync-request');
2
3  beforeEach(function() {
4      browser.url('/Contact-Us/contactus.html');
5  })
6
7  describe('Test Contact Us form WebdriverUni', function() {
8      var res = request('GET', 'http://jsonplaceholder.typicode.com/posts/1/comments');
9
10     var contactusDetails = JSON.parse(res.getBody().toString('utf8'));
11
12     beforeEach(function() {
13         console.log('Inside the describe block!');
14     })
15
16   it('Should be able to submit a successful submission via contact us form', function(done) {
17     browser.setValue("[name='first_name']",'Joe');
18     browser.setValue("[name='last_name']",'Blogs');
19     browser.setValue("[name='email']", 'joe_blogs@mail.com');
20     browser.setValue("[name='message']", 'How much does product x cost?');
21     browser.click("[type='submit']");
22
23     var successfulContactConfirmation = browser.isExisting('#contact_reply h1');
24     expect(successfulContactConfirmation, 'Successful submission Message does not exist').to.be.true;
25
26     var successfulSubmission = browser.getText('#contact_reply h1');
27     expect(successfulSubmission).to.equal('Thank You for your Message!');
28     });
29
```

Look at line 19:

```
19      browser.setValue("[name='email']", 'joe_blogs@mail.com');
```

Here we are using the setValue function and supplying a locator of ("[name='email']" and then we set a value to be used ('joe_blogs@mail.com').

What we can do instead is pass a value from a JSON file, meaning we don't have to specifically write joe_blogs@mail.com within the file. In other words, we reference a field from the JSON file and the data is retrieved and used as part of the test.

Why is this useful? Because, say we wanted to test a variety of different values when running our tests. The old method would require us amending each and every value before we execute the test. Now all we would need to do is supply a different JSON file that contains different data or amend the data in the JSON file itself, meaning the test .js file doesn't have to be changed.

Instructions:

How to install the sync-request package and how we can use the JSON file to parse data to our test

1. Open up GitBash / iTerm2 and navigate to our webdriverioFramework directory
2. To install the sync-request package, type the following command:

```
GBruno@Gi MINGW64 ~/Desktop/webdriverioFramework
$ npm install --save-dev sync-request
```

3. Then press enter
4. The package should begin to install and once complete, it should look like this:

```
MINGW64:/c/Users/GBruno/Desktop/webdriverioFramework
npm WARN webdriverioframework@1.0.0 No repository field.

+ sync-request@6.0.0
added 17 packages from 39 contributors and audited 744 packa
ges in 4.697s
found 0 vulnerabilities


GBruno@Gi MINGW64 ~/Desktop/webdriverioFramework
$
```

5. Now we need to modify our contactUsTest.js file. Go to the tests folder using windows explorer/ finder or download the file from the Udemy lecture resources section and place a copy in the tests folder
6. We need to create a new variable that links to sync-request package. Type the following on line 1 of your test file:

```
contactUsTest.js
1  var request = require('sync-request');
```

7. Then add the following lines to your code:

```
7  describe('Test Contact Us form WebdriverUni', function() {
8      var res = request('GET', 'http://jsonplaceholder.typicode.com/posts/1/comments');
9
10     var contactusDetails = JSON.parse(res.getBody().toString('utf8'));
11
12     beforeEach(function() {
13         console.log('Inside the describe block!');
14     })
```

   a. Arrow 1 – we are creating a new variable and setting it a value as shown on the sync-request documentation that links to a JSON file based on the URL we provide
   b. Arrow 2 – we are formatting the data so that we can use it within our tests.
      i. JSON – indicates we are using retrieving the data from a JSON format
      ii. .parse – indicates we want to parse the data from the file to our test
      iii. res.getBody – is the result from the GET response and providing the body of the data retrieved
      iv. .toString('utf8) – instructs that we want to data as a String format using utf8 encoding (a widely accepted uni-code encoding that supports many languages)

We continue making changes to our file in the next lecture

Lecture 66 - Using External Data (Sync Data Mode) - Part 2

In this lecture we continue on from Part 1 and finish making changes to our test file to use data from a JSON file. So far, we have installed the sync-request package, have referenced the package in our test file and have created two variables that link to the JSON file URL and have set the format of the data retrieved.

Lecture Resources on Udemy:

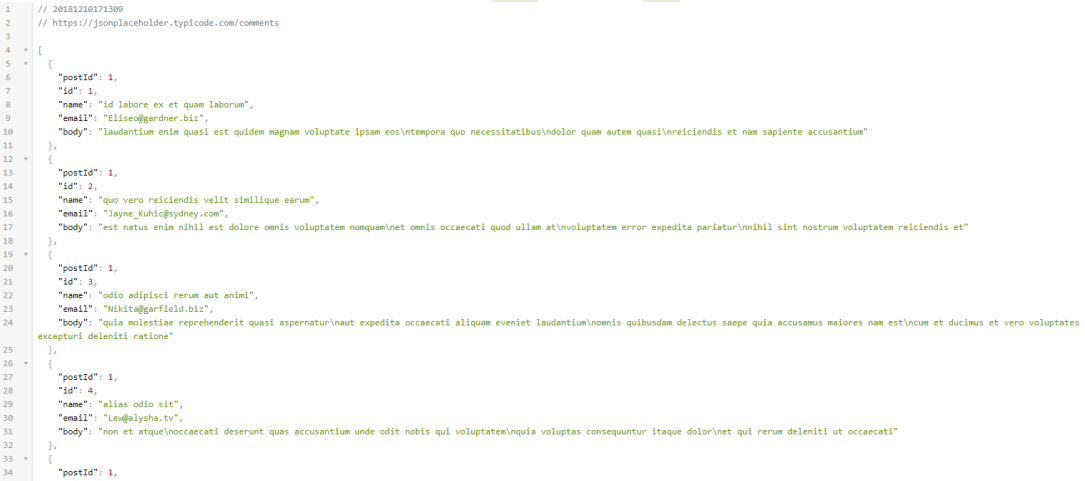
- https://www.npmjs.com/package/sync-request
- https://jsonplaceholder.typicode.com/comments

Instructions (continued from Part 1)

1. Now we need to alter our first "it" test to use the JSON data
2. If not already open, use Sublime Text and open the contactUs.js test from your "tests" folder
3. The code for the first "it" block should look like this:

```
16  it('Should be able to submit a successful submission via contact us form', function(done) {
17    browser.setValue("[name='first_name']",'Joe');
18    browser.setValue("[name='last_name']",'Blogs');
19    browser.setValue("[name='email']", 'joe_blogs@mail.com');
20    browser.setValue("[name='message']", 'How much does product x cost?');
21    browser.click("[type='submit']");
22
23    var successfulContactConfirmation = browser.isExisting('#contact_reply h1');
24    expect(successfulContactConfirmation, 'Successful submission Message does not exist').to.be.true;
25
26    var successfulSubmission = browser.getText('#contact_reply h1');
27    expect(successfulSubmission).to.equal('Thank You for your Message!');
28    });
29
```

4. If we quickly review the JSON data (see URL above) it currently looks like this:



```
// 20181210171309
// https://jsonplaceholder.typicode.com/comments

[
  {
    "postId": 1,
    "id": 1,
    "name": "id labore ex et quam laborum",
    "email": "Eliseo@gardner.biz",
    "body": "laudantium enim quasi est quidem magnam voluptate ipsam eos\ntempora quo necessitatibus\ndolor quam autem quasi\nreiciendis et nam sapiente accusantium"
  },
  {
    "postId": 1,
    "id": 2,
    "name": "quo vero reiciendis velit similique earum",
    "email": "Jayne_Kuhic@sydney.com",
    "body": "est natus enim nihil est dolore omnis voluptatem numquam\net omnis occaecati quod ullam at\nvoluptatem error expedita pariatur\nnihil sint nostrum voluptatem reiciendis et"
  },
  {
    "postId": 1,
    "id": 3,
    "name": "odio adipisci rerum aut animi",
    "email": "Nikita@garfield.biz",
    "body": "quia molestiae reprehenderit quasi aspernatur\naut expedita occaecati aliquam eveniet laudantium\nomnis quibusdam delectus saepe quia accusamus maiores nam est\ncum et ducimus et vero voluptates excepturi deleniti ratione"
  },
  {
    "postId": 1,
    "id": 4,
    "name": "alias odio sit",
    "email": "Lew@alysha.tv",
    "body": "non et atque\noccaecati deserunt quas accusantium unde odit nobis qui voluptatem\nquia voluptas consequuntur itaque dolor\net qui rerum deleniti ut occaecati"
  },
  {
    "postId": 1,
```

5. If we amend our first "it" code block to use this data, then the test will run five times as there are five objects within the JSON file
6. The first change we need to make is use a forEach hook that will run for each test. I have added the following to line 16:

```
16  contactusDetails.forEach(function (contactDetail) {
```

   a. Notice we have created an object called contactDetail for which we will use within the "it" test (shown in a moment)

7. I have also added closing brackets on line 30:
8.

```
30      });
```

9. I then replace the following (before I changed it):

```
20    browser.setValue("[name='email']", 'joe_blogs@mail.com');
21    browser.setValue("[name='message']", 'How much does product x cost?');
22    browser.click("[type='submit']");
```

10. To (updated code below):

```
20    browser.setValue("[name='email']", contactDetail.email);
21    browser.setValue("[name='message']", contactDetail.body);
22    browser.click("[type='submit']");
```

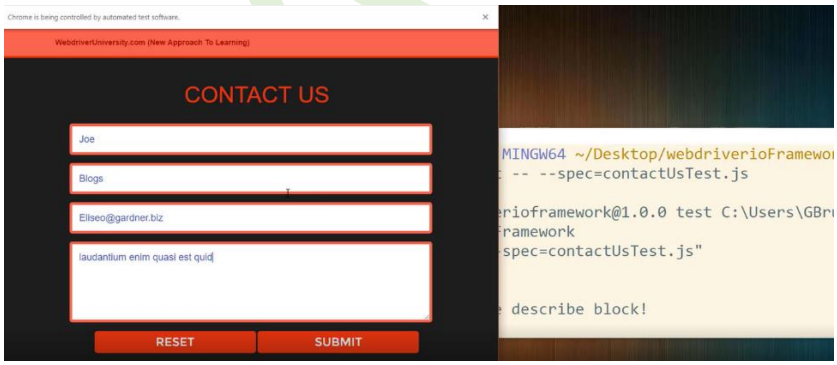
11. What I have done here is used the object (contactDetail) to call the .email and .body data from the JSON file (look at the JSON file and notice there are email and body attributes):

```
6     "postId": 1,
7     "id": 1,
8     "name": "id labore ex et quam laborum",
9     "email": "Eliseo@gardner.biz",
10    "body": "laudantium enim quasi est quidem magnam voluptate ipsam eos\ntempora quo necessit
11  },
12  {
13    "postId": 1,
14    "id": 2,
15    "name": "quo vero reiciendis velit similique earum",
16    "email": "Jayne_Kuhic@sydney.com",
17    "body": "est natus enim nihil est dolore omnis voluptatem numquam\net omnis occaecati quod
18  },
```

12. This means that for each test that runs, go and retrieve the email and body input from the JSON file and parse it to the test when running (instead of using a set value that was previously defined in the .js file before)
13. Since we only want to run the first "it" test (since for the purpose of this demonstration, I just want to show how we can use JSON data on the first "it" test block) make the following change to the first "it" code block

```
it.only('Should be able to submit a
```

    a. This will mean only the first "it" test block is initiated
14. Once you have made the changes above, save the file
15. Open up GitBash / iTerm2 and navigate to our webdriverioFramework directory
16. Run the following command:
17. **npm test -- -- specs=tests/contactUsTest.js** + press enter
18. You should see the tests run, as shown below:



19. Notice the data in the email and message fields. Look familiar? This is data from the JSON file that's being parsed to our test
20. You should see the test run multiple times (for each JSON object)
21. Finally, you should see 5 test passes:

22. This confirms how we can use data from an external source (JSON file).
23. Make sure to remove the following (.only) from your file before proceeding to the next lecture:

```
it.only('Should be able to submit a
```

Module 19 – Custom Commands (addCommand)

In this module, we look at custom commands using the addCommand function from the webdriverio documentation. Custom commands give you the opportunity to bundle a specific sequence of commands that are used frequently in a handy single command call.

Lecture 67 - Custom Commands (addCommand) - Part 1

If you want to extend the browser instance with your own set of commands, there is a method called addCommand available from the browser object.

Lecture Resources on Udemy:

- http://v4.webdriver.io/api/utility/addCommand.html
- http://v4.webdriver.io/guide/usage/customcommands.html
- http://jsonplaceholder.typicode.com/posts/1/comments

Details about this test:

If we review our current contactUsTest.js file, we can see that we have repetitive code, as shown:

contactUsTest.js

```
var request = require('sync-request');

beforeEach(function() {
    browser.url('/Contact-Us/contactus.html');
})

describe('Test Contact Us form WebdriverUni', function() {
    var res = request('GET', 'http://jsonplaceholder.typicode.com/posts/1/comments');

    var contactusDetails = JSON.parse(res.getBody().toString('utf8'));

    beforeEach(function() {
        console.log('Inside the describe block!');
    })

  it('Should be able to submit a successful submission via contact us form', function(done) {
    browser.setValue("[name='first_name']",'Joe');
    browser.setValue("[name='last_name']",'Blogs');
    browser.setValue("[name='email']", 'joe_blogs@mail.com');
    browser.setValue("[name='message']", 'How much does product x cost?');
    browser.click("[type='submit']");

    var successfulContactConfirmation = browser.isExisting('#contact_reply h1');
    expect(successfulContactConfirmation, 'Successful submission Message does not exist').to.be.true;

    var successfulSubmission = browser.getText('#contact_reply h1');
    expect(successfulSubmission).to.equal('Thank You for your Message!');
    });

  it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
    browser.setValue("[name='first_name']",'Mike');
    browser.setValue("[name='last_name']",'Woods');
    browser.setValue("[name='email']", 'mike_woods@mail.com');
    browser.click("[type='submit']");

    var successfulContactConfirmation = browser.isExisting('#contact_reply h1');
    expect(successfulContactConfirmation, 'Successful submission Message does not exist').to.be.false;
    });


  it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
    browser.setValue("[name='first_name']",'Sarah');
    browser.setValue("[name='email']", 'sarah_woods@mail.com');
    browser.click("[type='submit']");

```

This is bad practice, as we should always look to abide by the DRY (don't repeat yourself) concept. What we intend to do in this lecture (and next) is to create and use custom commands so that we can combine all first_name, last_name, email, message and submit all into a single custom command that we can call and use.

It doesn't matter if in some "it" tests we only use some of the locators. We develop our custom commands to handle various scenarios (which I show you next).

Instructions:

1. Download the code example from the Udemy resources section associated to this lecture or open the contactUsTest.js file from your "tests" folder using Sublime Text
2. We need to add a custom command using the *addCommand* function and we can do this by adding the following code to our test file:

```
var request = require('sync-request');

browser.addCommand("submitDataViaContactUsForm", function (firstName, lastName, emailAddress, comments) {
    if(firstName) {
        browser.setValue("[name='first_name']", firstName);
    }
    if(lastName) {
        browser.setValue("[name='last_name']", lastName);
    }
    if(emailAddress) {
        browser.setValue("[name='email']", emailAddress);
    }
    if(comments) {
        browser.setValue("[name='message']", comments);
    }
    browser.click("[type='submit']");
}

beforeEach(function() {
    browser.url('/Contact-Us/contactus.html');
})
```

3. The code above does the following:
    a. We use the addCommand and the first parameter is the name we give the new command ("submitDataViaContactUsForm")
    b. We then provide a number of parameters (firstName, lastName, emailAddress etc.)
    c. We then use if statements to see if a parameter value has been provided, so for the first example we are checking to see IF there is a firstName value, and if so, we use the locator for the first_name and provide it a value passed by the parameter
    d. We repeat this for each of the contact us fields
    e. Lastly, we have some code at the end that selects the submit button. Notice this is not in an IF statement and will submit regardless when this command is used
4. Once you have made the changes above, save the file

We continue with custom commands in the next lecture

Lecture 68 - Custom Commands (addCommand) - Part 2

In this lecture, we continue on from Part 1 and start making use of the new custom command that we've now created. At the moment, our test still references locators in each of the "it" blocks. We need to update this so that we now call the custom command.

Lecture Resources on Udemy:

- http://v4.webdriver.io/api/utility/addCommand.html
- http://v4.webdriver.io/guide/usage/customcommands.html
- http://jsonplaceholder.typicode.com/posts/1/comments

Instructions (continued from Part 1):

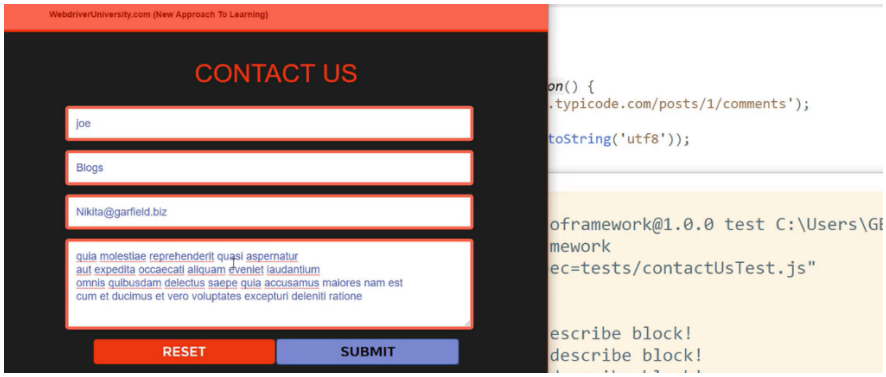
1. Let's recap and be clear on the changes we are about to make to the first "it" code block
2. Currently, the first "it" block looks like this:

```
15
16  it('Should be able to submit a successful submission via contact us form', function(done) {
17    browser.setValue("[name='first_name']",'Joe');
18    browser.setValue("[name='last_name']",'Blogs');
19    browser.setValue("[name='email']", 'joe_blogs@mail.com');
20    browser.setValue("[name='message']", 'How much does product x cost?');
21    browser.click("[type='submit']");
22
23    var successfulContactConfirmation = browser.isExisting('#contact_reply h1');
24    expect(successfulContactConfirmation, 'Successful submission Message does not exist').to.be.true;
25
26    var successfulSubmission = browser.getText('#contact_reply h1');
27    expect(successfulSubmission).to.equal('Thank You for your Message!');
28    });
29
```

3. We no longer the lines 17-21, so remove them
4. Then, amend your code to look like this:

```
33  it('Should be able to submit a successful submission via contact us form', function(done) {
34    browser.submitDataViaContactUsForm('joe', 'Blogs', contactDetail.email, contactDetail.body);
35
36    var successfulContactConfirmation = browser.isExisting('#contact_reply h1');
37    expect(successfulContactConfirmation, 'Successful submission Message does not exist').to.be.true;
38
39    var successfulSubmission = browser.getText('#contact_reply h1');
40    expect(successfulSubmission).to.equal('Thank You for your Message!');
41      })
42    });
```

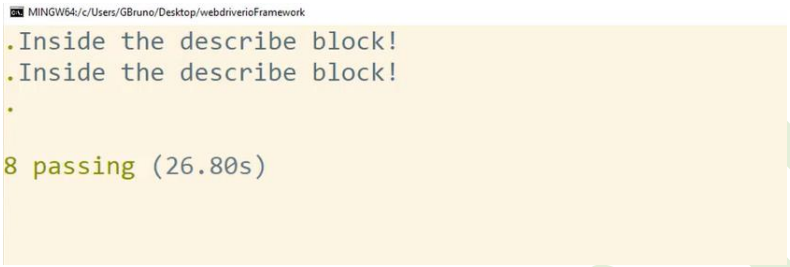
5. What we have done here is called the browser and referenced our new custom command (submitDataViaContactUsForm) and then provided parameter values ('joe', 'blogs', contactDetail.email, contactDetail.body ← the last two uses the JSON file data)
6. Once you have made the change above, save your file
7. Open up GitBash / iTerm2 and navigate to our webdriverioFramework directory
8. Run the following command:
9. **npm test -- -- specs=tests/contactUsTest.js** + press enter
10. You should see the tests run, as shown below:



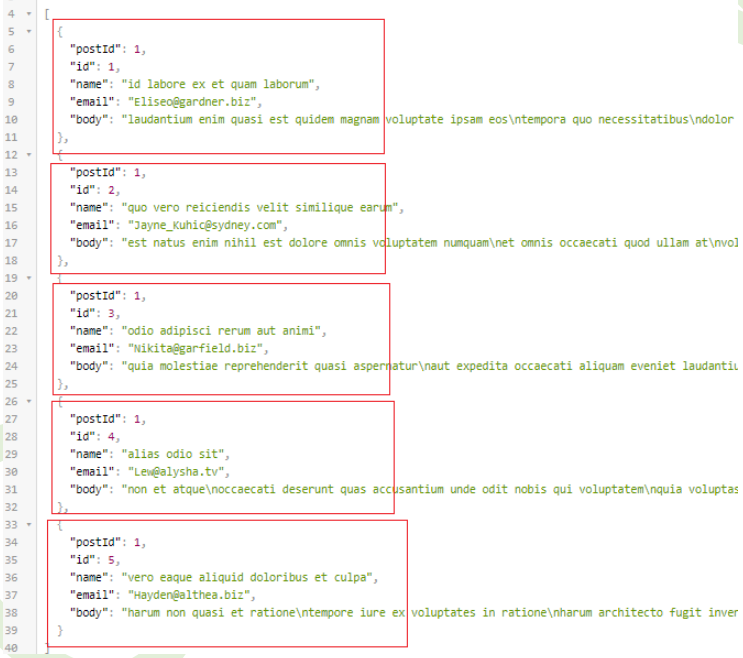
11. Notice how our test now runs similar to before but requires much less code!
12. Joe and Blogs are values we have defined as parameter values
13. The email and body are values parsed from the JSON file
14. The first “it” block will run multiple times based on the number of objects present in the JSON file (recap using lecture 63 & 64 if needed) but each initiation would use “Joe” and “blogs” since we have explicitly set these value as shown:

```
32  contactusDetails.forEach(function (contactDetail) {
33    it('Should be able to submit a successful submission via contact us form', function(done) {
34      browser.submitDataViaContactUsForm('joe', 'Blogs', contactDetail.email, contactDetail.body);
35
```

15. The console should show that 8 tests have passed:



16. 8 tests have run because:
    a. The first “it” block runs five times as there are five objects in the JSON file



    b. Then the second, third and fourth “it” blocks run once each
    c. This equates to 8 tests in total (which we were expecting)

Module 20 – Injecting JavaScript Code (Execute Command)

In this module, we look at injecting Javascript to a webpage when our tests run using the Execute command. This is useful because is provides flexibility.

Say there was a video player on a webpage and we want to press start the video, then pause it and then start it again. We can do this using JavaScript. We can tell our test to run Javascript code when needed to carry out certain actions.

Lecture 69 - Injecting JavaScript Code (Execute Command) - Part 1

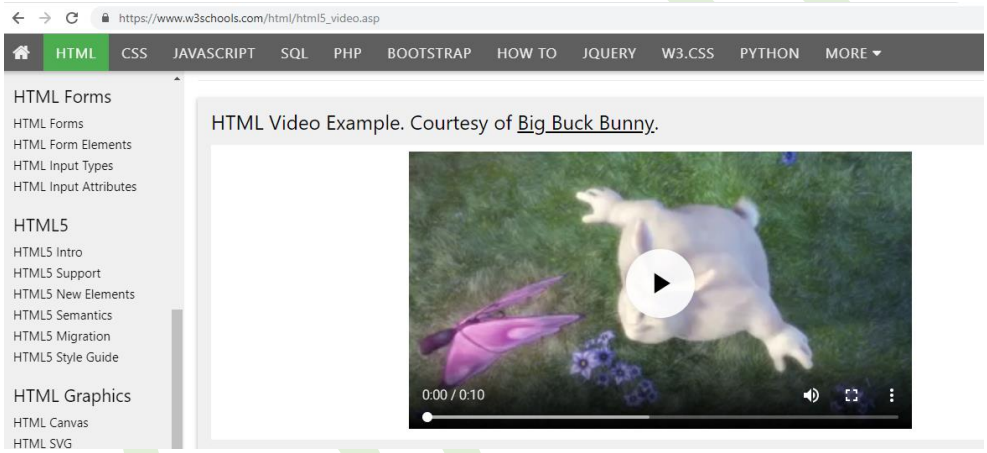
In this lecture, we look at interacting with a video on a webpage. The video player has a play button that transitions to a pause when pressed. We are going to use Javascript to interact with the video player by writing Javascript code within our test file by using the *execute* command.

Lecture Resources on Udemy:

- http://v4.webdriver.io/api/protocol/execute.html#Usage
- https://www.w3schools.com/html/html5_video.asp

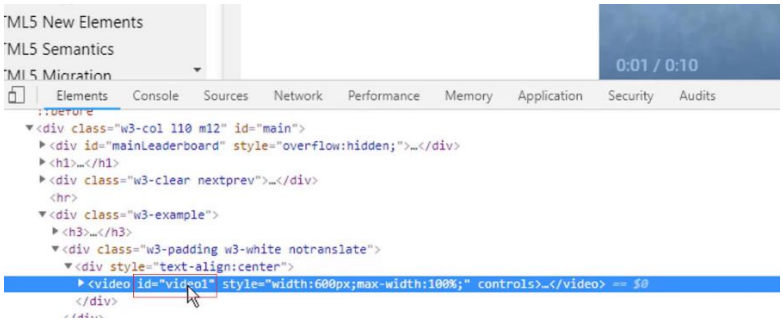
Details about this test:

If we head over to the following URL https://www.w3schools.com/html/html5_video.asp you will see a video that has a duration of 10 seconds

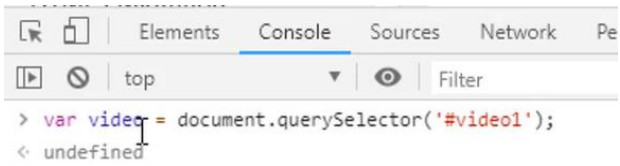


As a user, you can see that you can start the video by selecting the play button. However, think about how you could instruct your test to start the video? There isn't a specific button (like the submit button from past tests) to use. What we can do is use JavaScript to start the video and we can embed the Javascript into our test file.

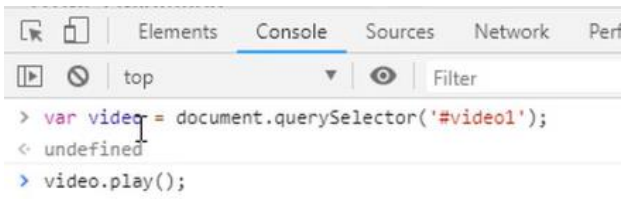
For example, if we use the inspector tool and focus on the video, we find that the video itself has a unique id:



We can use this id as a locator. We can also use the console tab to demonstrate how we can write Javascript to start the video:



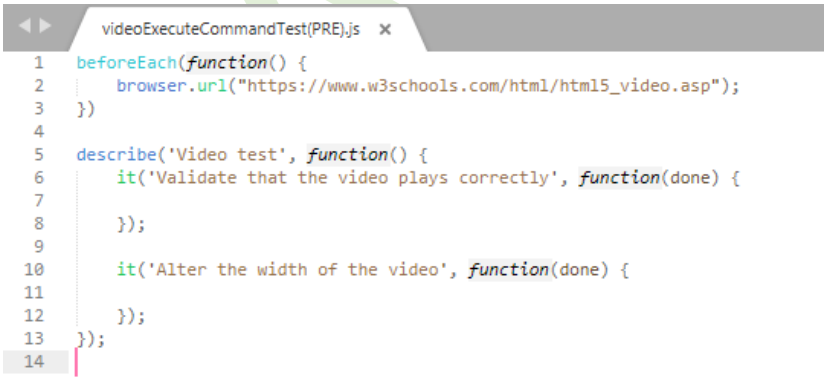
Here we define a variable called video and use a Javascript method to identify the selector. We can then write the following that will start the video:



This demonstrates how we can use Javascript in general. We are able to manipulate the DOM on the front-end. We will now look at how we can use Javascript inside one of our tests.

Instructions:

1. I have provided a Mocha template that you can download from the Udemy resources section associated to this lecture
2. Download it and save it to you tests folder and give it a name of videoExecuteCommandTest.js
3. Open the file using Sublime Text
4. It should look like this:



   a. We have used a beforeEach hook that navigates to the w3schools.com video webpage
   b. We have created a describe block that explains the overall test case
   c. We have created two “it” test blocks with descriptions of what we are wanting to achieve

5. The next few steps will focus around preparing our tests for the next couple of lectures. The main points that we need to complete in this lecture are the following:

```
beforeEach(function() {
    browser.url("https://www.w3schools.com/html/html5_video.asp");
})

//Injects a snippet of JavaScript into the page for execution in the context of the currently selected frame.
browser.addCommand('isVideoPaused', function(){

    //anything outside the script cant access inside vise versa
    var isPaused = browser.execute(function(){
        console.log('Outputted inside the console window');

        //cant use browser command inside the script
        var video = document.querySelector('#video1');
        return video.paused;
    })
    return isPaused.value; //verify that the video is indeed paused
});

describe('Video test', function() {
    it('Validate that the video plays correctly', function(done) {

    });

    it('Alter the width of the video', function(done) {

    });
});

```

   a. Create a new variable *isPaused* that has a value derived by using the execute command
   b. The inner code of the execute function will log a message to the console
   c. We create a second variable *video* that uses the JavaScript queryselector method, that takes an id value as a parameter
   d. We then return variable.paused which will return true or false depending if the video is paused at the time the code runs, which is then assigned to the *video* variable
   e. Lastly, we return isPaused.value which will return true or false as this variable will inherit the value assigned to the *video* variable
6. Once you have made the changes above, save your file.

We continue with this subject in the next lecture

Lecture 70 - Injecting JavaScript Code (Execute Command) - Part 2

In this lecture, we continue from Part 1 and continue to develop our test to use Javascript code injection within our test.

Lecture Resources on Udemy:

- http://v4.webdriver.io/api/protocol/execute.html#Usage
- https://www.w3schools.com/html/html5_video.asp

Instructions:

We are now ready to start adding to our "it" test blocks:

FINAL_videoExecuteCommandTest.js | SKELETON_videoExecuteCommandTest.js

```
1  beforeEach(function() {
2      browser.url("https://www.w3schools.com/html/html5_video.asp");
3  })
4
5  browser.addCommand('isVideoPaused', function(){
6      var isPaused = browser.execute(function(){
7          console.log('Outputted inside the console window');
8
9          var video = document.querySelector('#video1');
10         return video.paused;
11     })
12     return isPaused.value;
13 });
14
15 describe('Video test', function() {
16     it('Validate that the video is paused upon accessing the page', function(done) {
17
18     });
19
20     it('Alter the width of the video', function(done) {
21
22     });
23 });
24
```

1. We start with the first "it" block
2. We override the default timeout using "this.timout(20000);"
3. We set a browser.pause(6000); so, we can see the console output when the test is running
    a. In the real world, you would not need to add a pause here
4. We create a new variable called *isPaused* and set a value browser.isPaused();
    a. **This uses the method defined on lines 5-13 (notice the usage of isVideoPaused() in the below image**
5. We then create an assertion using the *expect* command:
    a. expect(isPaused).to.be.true;
    b. here we are expecting a true value to be returned as the video should be paused when visiting the webpage
6. We then add a pause command (browser.pause(6000);
    a. This is so we can see the outcome when reviewing the test
    b. In the real world, you would not need to add a pause here
7. You first "it" block should now look like this:

```
16      it('Validate that the video is paused upon accessing the page', function(done) {
17          this.timeout(20000);
18          var isPaused = browser.isVideoPaused();
19          expect(isPaused).to.be.true;
20          browser.pause(6000);
21      });
```

8. Now we are ready to develop the second “it” block
9. We create a new variable called *videoWidth* and give is a value as shown:

```
var videoWidth = browser.execute(function() {
    var video = document.querySelector('#video1');
    return video.style.width = "300px";
```

    a. Here we are using the *execute* command using a function that uses a new variable, *video*
    b. We then assign the *video* variable a value of document.querySelector which is a Javascript method (read more here: https://developer.mozilla.org/en-US/docs/Web/API/Document/querySelector) and pass is a parameter value of the locator of the video (‘#video1’)
    c. This create an object of the element (the video in this case) and assigns it to the variable, *video*
    d. We have then added a return statement that uses the *video* object and returns the style.width = “300px”;
        i. We are amending the width of the video1 element by overriding the default width
10. Finally, we add another pause, using browser.pause(6000);
11. Your second “it” block should now look like this:

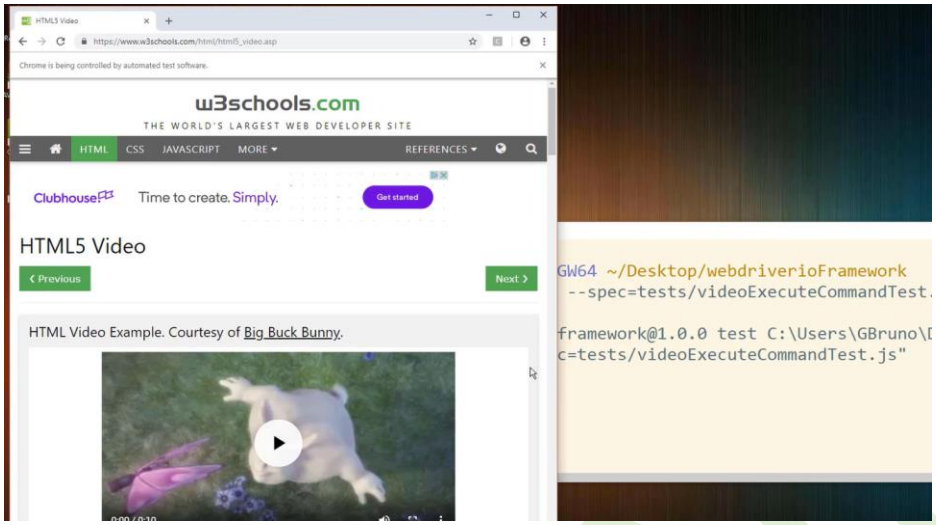
```
24      it('Alter the width of the video', function(done) {
25          var videoWidth = browser.execute(function() {
26              var video = document.querySelector('#video1');
27              return video.style.width = "300px";
28          });
29          browser.pause(6000);
30      });
```

12. Your overall file should now look like this:

```
1   beforeEach(function() {
2       browser.url("https://www.w3schools.com/html/html5_video.asp");
3   })
4
5   browser.addCommand('isVideoPaused', function(){
6       var isPaused = browser.execute(function(){
7           console.log('Outputted inside the console window');
8
9           var video = document.querySelector('#video1');
10          return video.paused;
11      })
12      return isPaused.value;
13  });
14
15  describe('Video test', function() {
16      it('Validate that the video is paused upon accessing the page', function(done) {
17          this.timeout(20000);
18          var isPaused = browser.isVideoPaused();
19          expect(isPaused).to.be.true;
20          browser.pause(6000);
21      });
22
23      it('Alter the width of the video', function(done) {
24          var videoWidth = browser.execute(function() {
25              var video = document.querySelector('#video1');
26              return video.style.width = "300px";
27          });
28          browser.pause(6000);
29      });
30  });
```

13. Once you have made the above changes, save your file and close Sublime Text
14. Open up GitBash / iTerm2 and navigate to our webdriverioFramework directory

15. Run the following command:
16. **npm test -- -- specs=tests/videoExecuteCommandTest.js** + press enter
17. You should see the tests run, as shown below:



18. You should see the width of the video change when the second “it” test completes
19. You should also see the following console output when the test completes:

20. Notice that both tests have passed. This means out first “it” test is paused which was the value we were expecting based on our assertion:

```
16      it('Validate that the video is paused upon accessing the page', function(done) {
17          this.timeout(20000);
18          var isPaused = browser.isVideoPaused();
19          expect(isPaused).to.be.true;
20          browser.pause(6000);
21      });
```

21. The second “it” test didn’t contain an assertion, but we observed the width of the video change. We could confirm this by adding an assertion and providing an expected width if we wanted.

We continue with this subject in the next lecture

Lecture 71 - Injecting JavaScript Code (Execute Command) - Part 3

In this lecture, we make a small change to slow down our test so that we can see the output that's logged to the console when the first "it" test block runs.

Lecture Resources on Udemy:

- http://v4.webdriver.io/api/protocol/execute.html#Usage
- https://www.w3schools.com/html/html5_video.asp

Details about the change:

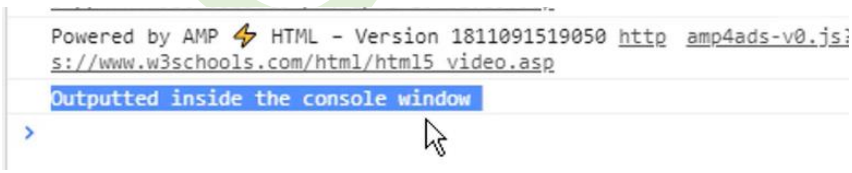
You may have noticed that the following line did not show in the console when we ran out test:

```
1   beforeEach(function() {
2       browser.url("https://www.w3schools.com/html/html5_video.asp");
3   })
4
5   browser.addCommand('isVideoPaused', function(){
6       var isPaused = browser.execute(function(){
7           console.log('Outputted inside the console window');
8
9           var video = document.querySelector('#video1');
10          return video.paused;
11      })
12      return isPaused.value;
13  });
```

This is because the script runs so fast, that we are unable to see the message output. We can make the following change to slow the **first** "it" test down so that you can see the output message:

```
15  describe('Video test', function() {
16      it('Validate that the video is paused upon accessing the page', function(done) {
17          this.timeout(20000);
18          browser.pause(6000);
19          var isPaused = browser.isVideoPaused();
20          expect(isPaused).to.be.true;
21          browser.pause(6000);
22      });
23
```

If we run our test again and look at the console using the inspector tool (F12) you should see the following:



This confirms that our first "it" block uses the method defined on lines 5-13 of our test file. The message above is outputted because we defined this on line 7.

The objective of these lectures was to show you the benefits and how easy it is to embed Javascript code into your tests using the webdriverio *execute* function. This is powerful and means we can interact with elements on a webpage during our tests!

Module 21 – Page Object Model (POM)

In this module, we look at Page Object Modelling (POM) and look at how we can further improve our tests using abstraction.

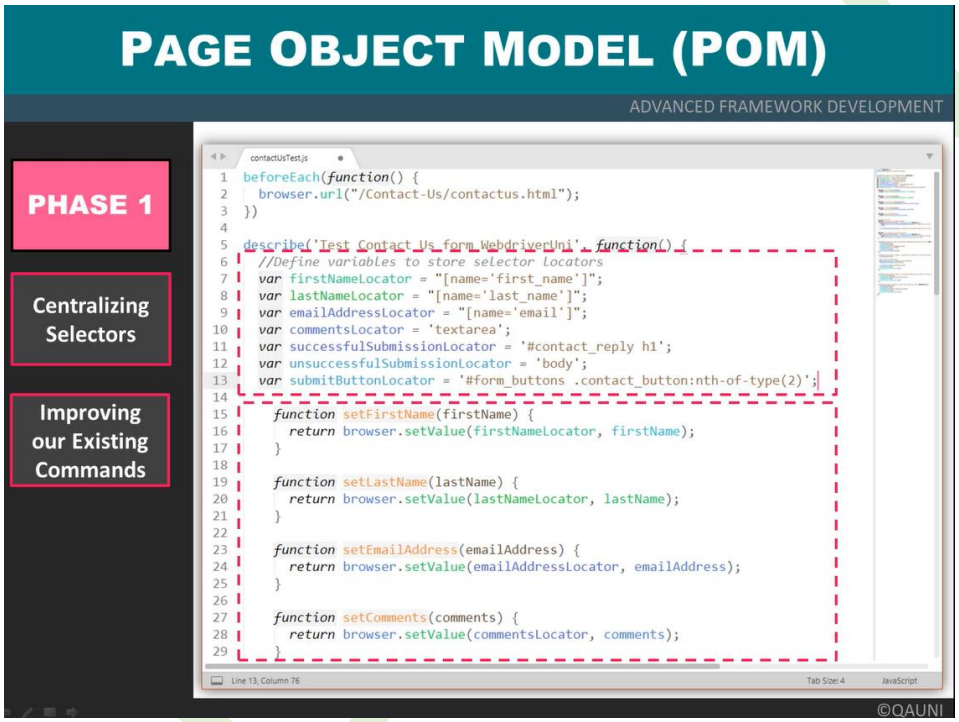
This module is split into three phases. Phase one provides an overview and lays the groundwork. Phase two builds upon phase one. Phase three goes further and builds on phase two.

Note that we will be using our contactUsTest.js file during the POM section. You can download the lecture code from each of the Udemy resources section or you can use your local copy of the test file.

Lecture 72 - Page Object Model (POM) – Intro

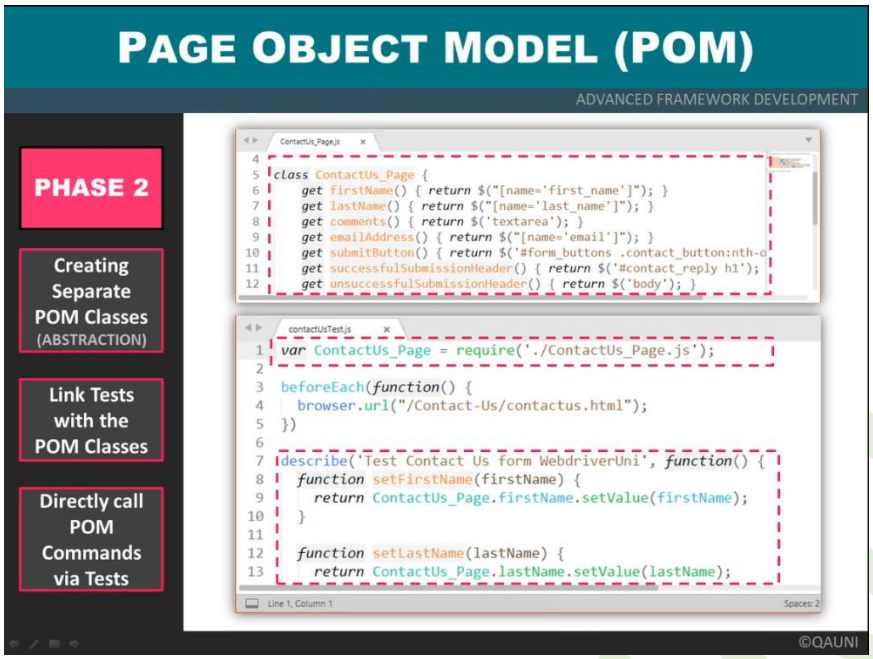
In this lecture, we cover what we are going to do in phase one, two and three. This lecture provides an overview of each phase and highlights some of the key improvements that we will cover.

Phase One



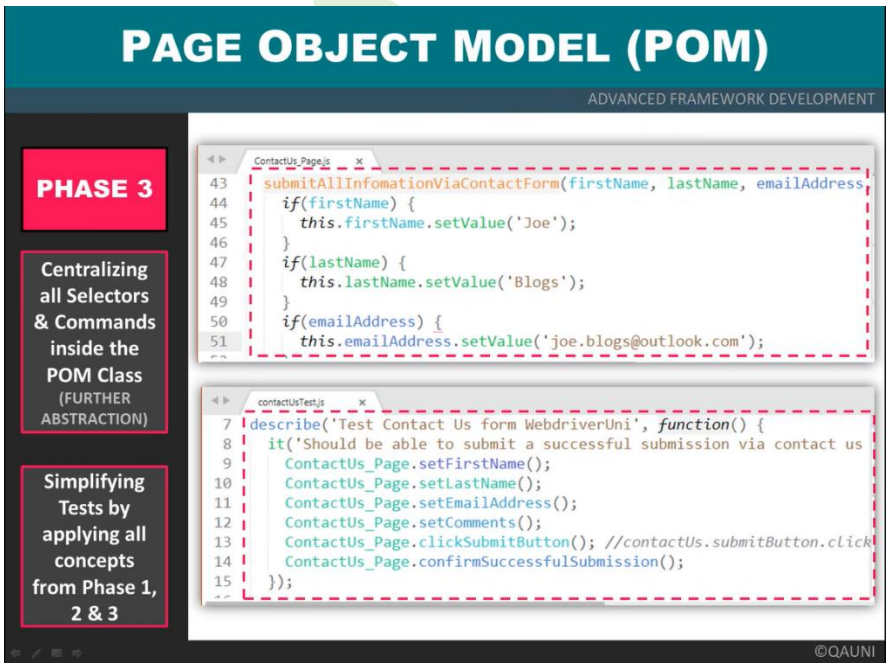
- In this phase, we look at centralising our selectors. This means we no longer need to explicitly define selector values inside our "it" tests. The benefit of this is maintainability.
    - Say you have a number of test files that use various locators and at some stage, the name of those locator's change. It would mean we have to update the one variable instead of each individual test case.
- We also take a look at improving our *existing* commands so that they are efficient, intelligent and cleaner.

Phase Two



- In phase two, we look at we look at creating separate Page Object Model classes that will use abstraction to further improve our tests
- We will look at linking our tests to the POM classes
- Then we look at ways of how we can directly call POM commands inside our test blocks

Phase Three



- In phase three, we go even further and start centralizing all selectors and commands into our POM class
- We also look at simplifying our tests even further by applying all concepts from phase one and two

The POM

Mohsin from medium.com does a great job explaining the POM and why we need it. You can read his full article here:

https://medium.com/tech-tajawal/page-object-model-pom-design-pattern-f9588630800b

The key points are listed below:

What is the Page Object Model?

Page Object Model is a design pattern which is very popular in test automation for improving test maintenance and lessening code duplication. A page object is an object-oriented class that fills in as an interface to a page of your system under test.

The tests then use the methods of this page object class at whatever point they have to connect with the UI of the page, the advantage is that if the UI changes for the page, the tests themselves don't need to be changed, only the code within the page object needs to change.

Subsequently all changes to support that new UI are located in one place.

Why we need POM?

Increasing automation test coverage can result in unmaintainable project structure, if locators are not managed in right way. This can happen due to duplication of code or mainly due to duplicated usage of locators.

For Example, in home page of web application we have menu bar which leads to different modules with different features.

Many automation test cases would be clicking through these menu buttons to execute specific tests. Imagine that the UI is changed/revamped and menu buttons are relocated to different position in home page, this will result automation tests to fail. Automated test cases will fail as scripts will not be able to find particular element-locators to perform action.

Now, QA Engineer need to walk through whole code to update locators where necessary. Updating element-locators in duplicated code will consume a lot of time to only adjust locators, while this time can be consumed to increase test coverage. We can save this time by using Page Object Model in our test automation framework.

Lecture 73 - Page Object Model (POM) - Phase 1 - Part 1

In this lecture, we start phase one and look at centralizing our selectors and improving our existing commands.

Instructions:

1. Download the contactUsTest.js file from the Udemy resource section associated to this lecture or use your local version of the file.
2. If you decide to download, ensure to take a backup of your existing contactUsTest.js file in the "tests" directory, and then place your downloaded version there
3. Open up the file using Sublime Text
4. You file at this stage should look like:

ORIG_contactUsTest.js

```
var request = require('sync-request');

browser.addCommand("submitDataViaContactUsForm", function (firstName, lastName, emailAddress, comments) {
    if(firstName) {
        browser.setValue("[name='first_name']", firstName);
    }
    if(lastName) {
        browser.setValue("[name='last_name']", lastName);
    }
    if(emailAddress) {
        browser.setValue("[name='email']", emailAddress);
    }
    if(comments) {
        browser.setValue("[name='message']", comments);
    }
    browser.click("[type='submit']");
})

beforeEach(function() {
    browser.url('/Contact-Us/contactus.html');
})

describe('Test Contact Us form WebdriverUni', function() {
    var res = request('GET', 'http://jsonplaceholder.typicode.com/posts/1/comments');

    var contactusDetails = JSON.parse(res.getBody().toString('utf8'));

    beforeEach(function() {
        console.log('Inside the describe block!');
    })

contactusDetails.forEach(function (contactDetail) {
  it('Should be able to submit a successful submission via contact us form', function(done) {
    browser.submitDataViaContactUsForm('joe', 'Blogs', contactDetail.email, contactDetail.body);

    var successfulContactConfirmation = browser.isExisting('#contact_reply h1');
    expect(successfulContactConfirmation, 'Successful submission Message does not exist').to.be.true;

    var successfulSubmission = browser.getText('#contact_reply h1');
    expect(successfulSubmission).to.equal('Thank You for your Message!');
        })
    });

  it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
    browser.setValue("[name='first_name']",'Mike');
    browser.setValue("[name='last_name']",'Woods');
    browser.setValue("[name='email']", 'mike_woods@mail.com');
    browser.click("[type='submit']");

    var successfulContactConfirmation = browser.isExisting('#contact_reply h1');
    expect(successfulContactConfirmation, 'Successful submission Message does not exist').to.be.false;
    });


  it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
    browser.setValue("[name='first_name']",'Sarah');
    browser.setValue("[name='email']", 'sarah_woods@mail.com');
    browser.click("[type='submit']");

    var successfulContactConfirmation = browser.isExisting('#contact_reply h1');
    expect(successfulContactConfirmation, 'Successful submission Message does not exist').to.be.false;
    });

  it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
    browser.setValue("[name='first_name']",'Jim');
    browser.setValue("[name='last_name']",'Jomes');
    browser.click("[type='submit']");

    var errorText = browser.getText('body');
    expect(errorText).to.include('Error: all fields are required');

    var errorText = browser.isVisible('body');
    expect(errorText, 'Error message is not visible').to.be.true;
    });
});
```

5. We then look at removing parts of the code that we are looking to improve
6. This is because we are going to be improving the way our test functions by adding intelligence that will make a code easier to maintain going forward
7. Remove the following sections, highlighted with the red border below:

ORIG_contactUsTest.js

```
var request = require('sync-request');

browser.addCommand("submitDataViaContactUsForm", function (firstName, lastName, emailAddress, comments) {
    if(firstName) {
        browser.setValue("[name='first_name']", firstName);
    }
    if(lastName) {
        browser.setValue("[name='last_name']", lastName);
    }
    if(emailAddress) {
        browser.setValue("[name='email']", emailAddress);
    }
    if(comments) {
        browser.setValue("[name='message']", comments);
    }
    browser.click("[type='submit']");
})

beforeEach(function() {
    browser.url('/Contact-Us/contactus.html');
})

describe('Test Contact Us form WebdriverUni', function() {
    var res = request('GET', 'http://jsonplaceholder.typicode.com/posts/1/comments');

    var contactusDetails = JSON.parse(res.getBody().toString('utf8'));

    beforeEach(function() {
        console.log('Inside the describe block!');
    })

contactusDetails.forEach(function (contactDetail) {
  it('Should be able to submit a successful submission via contact us form', function(done) {
    browser.submitDataViaContactUsForm('joe', 'Blogs', contactDetail.email, contactDetail.body);

    var successfulContactConfirmation = browser.isExisting('#contact_reply h1');
    expect(successfulContactConfirmation, 'Successful submission Message does not exist').to.be.true;

    var successfulSubmission = browser.getText('#contact_reply h1');
    expect(successfulSubmission).to.equal('Thank You for your Message!');
        })
    });

  it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
    browser.setValue("[name='first_name']",'Mike');
    browser.setValue("[name='last_name']",'Woods');
    browser.setValue("[name='email']", 'mike_woods@mail.com');
    browser.click("[type='submit']");

    var successfulContactConfirmation = browser.isExisting('#contact_reply h1');
    expect(successfulContactConfirmation, 'Successful submission Message does not exist').to.be.false;
    });


  it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
    browser.setValue("[name='first_name']",'Sarah');
    browser.setValue("[name='email']", 'sarah_woods@mail.com');
    browser.click("[type='submit']");

    var successfulContactConfirmation = browser.isExisting('#contact_reply h1');
    expect(successfulContactConfirmation, 'Successful submission Message does not exist').to.be.false;
    });

  it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
    browser.setValue("[name='first_name']",'Jim');
    browser.setValue("[name='last_name']",'Jomes');
    browser.click("[type='submit']");

    var errorText = browser.getText('body');
    expect(errorText).to.include('Error: all fields are required');

    var errorText = browser.isVisible('body');
    expect(errorText, 'Error message is not visible').to.be.true;
    });
});
```

8. Now we are ready to centralize our selectors and we can do this by first focusing on a selector that is used in various places within our code:
    a. Take the following example:
    b. `browser.setValue("[name='first_name']",'Mike');`
    c. This is defined multiple times throughout our code
9. This means that if the locator name changes, then currently we have to update our file in various places (very bad practice, especially if we have a number of test files!)
10. We create the following variables:

```
describe('Test Contact Us form WebdriverUni', function() {
  var res = request('GET', 'http://jsonplaceholder.typicode.com/posts/1/comments');
  var contactusDetails = JSON.parse(res.getBody().toString('utf8'));

  var firstNameSelector = "[name='first_name']";
  var lastNameSelector = "[name='last_name']";
  var emailAddressSelector = "[name='email']";
  var commentsSelector = "textarea";
  var successfulSubmissionSelector = "#contact_reply h1";
  var unsuccessfulSubmissionSelector = "body";
  var submitButtonSelector = "[type='submit']";

contactusDetails.forEach(function (contactDetail) {
  it('Should be able to submit a successful submission via contact us form', function(done) {
    browser.submitDataViaContactUsForm('joe', 'Blogs', contactDetail.email, contactDetail.body);
```

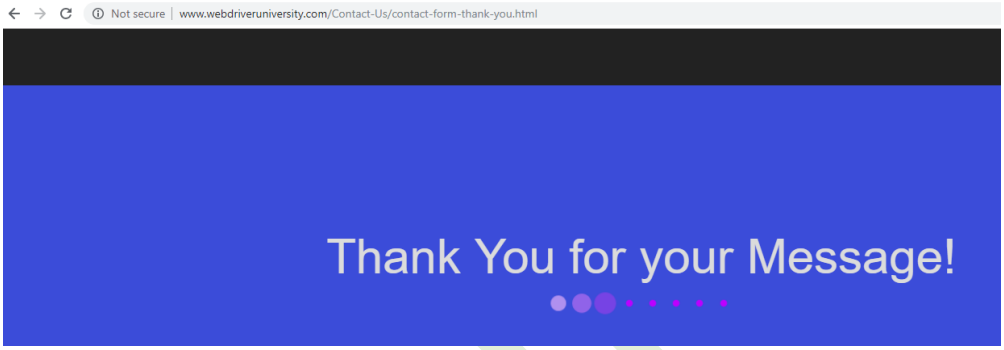
11. What we have done here is created variables that stores the locator values. This means that instead of defining each selector in each of the “it” tests, we can start looking at referencing the variables instead.
12. So, to explain further, take a look at the following variable:

```
15      var successfulSubmissionSelector = "#contact_reply h1";
```

    a. We are using a variable that will store what we expect to see when a successful submission is returned
    b. We know that from the following lines of code:

```
26      var successfulSubmission = browser.getText('#contact_reply h1');
27      expect(successfulSubmission).to.equal('Thank You for your Message!');
```

    c. We expect the #contact_reply h1 element will be returned because we are sent to the following page where this element resides:



We continue with phase one in the next lecture

Lecture 74 - Page Object Model (POM) - Phase 1 - Part 2

In this lecture, we continue improving our contactUsTest.js file from the work we have completed from Part 1.

Key Points:

- We have now removed redundant code in Part 1 that we are looking to improve
- We have also created the variables that stores out locators
- We are now going to use the new variables using functions so that we can parse values that are then returned that we can then use in our "it" test blocks

Instructions:

1. You code at this point should look like the following:

PRE_contactUsTest.js ● FINAL_contactUsTest.js ×

```
var request = require('sync-request');

beforeEach(function() {
    browser.url('/Contact-Us/contactus.html');
})

describe('Test Contact Us form WebdriverUni', function() {
    var res = request('GET', 'http://jsonplaceholder.typicode.com/posts/1/comments');
    var contactusDetails = JSON.parse(res.getBody().toString('utf8'));

    var firstNameSelector = "[name='first_name']";
    var lastNameSelector = "[name='last_name']";
    var emailAddresssSelector = "[name='email']";
    var commentsSelector = "textarea";
    var successfulSubmissionSelector = "#contact_reply h1";
    var unsuccessfulSubmissionSelector = "body";
    var submitButtonSelector = "[type='submit']";

contactusDetails.forEach(function (contactDetail) {
  it('Should be able to submit a successful submission via contact us form', function(done) {
    browser.submitDataViaContactUsForm('joe', 'Blogs', contactDetail.email, contactDetail.body);

    var successfulContactConfirmation = browser.isExisting('#contact_reply h1');
    expect(successfulContactConfirmation, 'Successful submission Message does not exist').to.be.true;

    var successfulSubmission = browser.getText('#contact_reply h1');
    expect(successfulSubmission).to.equal('Thank You for your Message!');
        })
    });

  it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
    browser.setValue("[name='first_name']",'Mike');
    browser.setValue("[name='last_name']",'Woods');
    browser.setValue("[name='email']", 'mike_woods@mail.com');
    browser.click("[type='submit']");

    var successfulContactConfirmation = browser.isExisting('#contact_reply h1');
    expect(successfulContactConfirmation, 'Successful submission Message does not exist').to.be.false;
    });


  it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
    browser.setValue("[name='first_name']",'Sarah');
    browser.setValue("[name='email']", 'sarah_woods@mail.com');
    browser.click("[type='submit']");

    var successfulContactConfirmation = browser.isExisting('#contact_reply h1');
    expect(successfulContactConfirmation, 'Successful submission Message does not exist').to.be.false;
    });

  it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
    browser.setValue("[name='first_name']",'Jim');
    browser.setValue("[name='last_name']",'Jomes');
    browser.click("[type='submit']");

    var errorText = browser.getText('body');
    expect(errorText).to.include('Error: all fields are required');

    var errorText = browser.isVisible('body');
    expect(errorText, 'Error message is not visible').to.be.true;
    });
});
```

2. We now need to create functions that will use our variables/selectors to interact with the contact us page

3. I will explain how one of these functions work before proceeding:

```
11      var firstNameSelector = "[name='first_name']";
12      var lastNameSelector = "[name='last_name']";
13      var emailAddressSelector = "[name='email']";
14      var commentsSelector = "textarea";
15      var successfulSubmissionSelector = "#contact_reply h1";
16      var unsuccessfulSubmissionSelector = "body";
17      var submitButtonSelector = "[type='submit']";
18
19      function setFirstName(firstName) {
20          return browser.setValue(firstNameSelector, firstName);
21      }
22
```

   a. Here I have created a new function called setFirstName which has a parameter named firstName
   b. We then return browser.setValue that takes a first argument of the variable from line 11 and then a second argument of firstName from the function parameter
   c. This means this function will set the value of the firstNameSelector variable to the value we provide for firstName

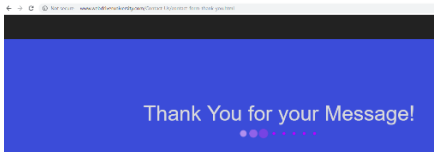
4. We add the following functions for the remaining variables:

```
19  function setFirstName(firstName) {
20      return browser.setValue(firstNameSelector, firstName);
21  }
22
23  function setLastName(lastName) {
24      return browser.setValue(lastNameSelector, lastName);
25  }
26
27  function setEmailAddress(emailAddress) {
28      return browser.setValue(emailAddressSelector, emailAddress);
29  }
30
31  function setComments(comments) {
32      return browser.setValue(commentsSelector, comments);
33  }
34
35  function clickSubmitButton() {
36      return browser.click(submitButtonSelector);
37  }
38
39  function confirmSuccessfulSubmission() {
40      var validateSubmissionHeader = browser.waitUntil(function() {
41          return browser.getText(successfulSubmissionSelector) == 'Thank You for your Message!'
42      }, 3000)
43      expect(validateSubmissionHeader, 'Successful Submission Message does not Exist!').to.be.true;
44  }
45
```

5. Most of the functions do exactly the same as the one demonstrated, apart from:
   a. **clickSubmitButton** – which doesn't take a parameter value because when this function is called, we just expect the submit button to be clicked
   b. **confirmSuccessfulSubmission** – this function uses *waitUntil* and we have developed this function to act in the following way:
      i. We create a new variable *validateSubmissionHeader* which is assigned a value of browser.waitUntil and then an inner function, where;
      ii. We return browser.getText and provide the variable from line 15:

```
15      var successfulSubmissionSelector = "#contact_reply h1";
```

      iii. This variable contains a value of a selector and we know the text associated to this element is "Thank you for your Message!" – indicating a successful submission, as shown on the webpage below:




```
41          return browser.getText(successfulSubmissionSelector) == 'Thank You for your Message!'
```

      iv. This line compares the text returned to the message we expect it to contain

```
}, 3000)
expect(validateSubmissionHeader, 'Successful Submission Message does not Exist!').to.be.true;
```

v. We then add a 3000 wait which instructs our function to wait for 3 seconds for the text to appear
vi. Lastly, we add an assertion using the *expect* method and pass the variable set in line 40
vii. We provide an error output message if the assertion doesn't equal the value we expect it to be
viii. Finally, we state .to.be.true because this is the value we are expecting it to return e.g. when we call the function **confirmSuccessfulSubmission** we expect the value to be true because we expect a successful contact us form submission will show the "Thank you for…" message

We continue with phase one in the next lecture

Lecture 75 - Page Object Model (POM) - Phase 1 - Part 3

In this lecture, we continue improving our contactUsTest.js file from the work we have completed from Part 2.

Key Points:

- We have now created functions for the variables created in Part 1
- The functions use the selectors associated to the variables as the first argument
- The functions themselves then have parameter values that allow us to set a value during our "it" tests
- We have created a function that deals with successful submissions
- We now need to create a function that deals with unsuccessful submissions

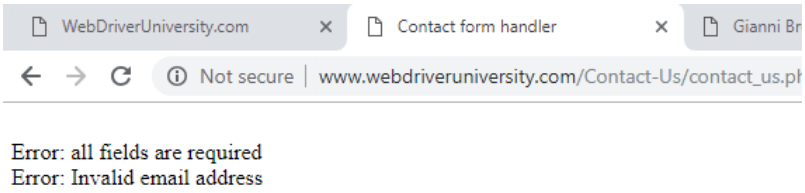
Instructions:

1. You code at this point should look like the following:

ORIG_contactUsTest.js | FINAL_contactUsTest.js

```
var request = require('sync-request');

beforeEach(function() {
    browser.url('/Contact-Us/contactus.html');
})

describe('Test Contact Us form WebdriverUni', function() {
    var res = request('GET', 'http://jsonplaceholder.typicode.com/posts/1/comments');
    var contactusDetails = JSON.parse(res.getBody().toString('utf8'));

    var firstNameSelector = "[name='first_name']";
    var lastNameSelector = "[name='last_name']";
    var emailAddresssSelector = "[name='email']";
    var commentsSelector = "textarea";
    var successfulSubmissionSelector = "#contact_reply h1";
    var unsuccessfulSubmissionSelector = "body";
    var submitButtonSelector = "[type='submit']";

    function setFirstName(firstName) {
        return browser.setValue(firstNameSelector, firstName);
    }

    function setLastName(lastName) {
        return browser.setValue(lastNameSelector, lastName);
    }

    function setEmailAddress(emailAddress) {
        return browser.setValue(emailAddressSelector, emailAddress);
    }

    function setComments(comments) {
        return browser.setValue(commentsSelector, comments);
    }

    function clickSubmitButton() {
        return browser.click(submitButtonSelector);
    }

    function confirmSuccessfulSubmission() {
        var validateSubmissionHeader = browser.waitUntil(function() {
            return browser.getText(successfulSubmissionSelector) == 'Thank You for your Message!'
        }, 3000)
        expect(validateSubmissionHeader, 'Successful Submission Message does not Exist!').to.be.true;
    }


contactusDetails.forEach(function (contactDetail) {
  it('Should be able to submit a successful submission via contact us form', function(done) {
    browser.submitDataViaContactUsForm('joe', 'Blogs', contactDetail.email, contactDetail.body);

    var successfulContactConfirmation = browser.isExisting('#contact_reply h1');
    expect(successfulContactConfirmation, 'Successful submission Message does not exist').to.be.true;

    var successfulSubmission = browser.getText('#contact_reply h1');
    expect(successfulSubmission).to.equal('Thank You for your Message!');
        })
    });

 it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
    browser.setValue("[name='first_name']",'Mike');
    browser.setValue("[name='last_name']", 'Woods');
    browser.setValue("[name='email']", 'mike_woods@mail.com');
    browser.click("[type='submit']");

    var successfulContactConfirmation = browser.isExisting('#contact_reply h1');
    expect(successfulContactConfirmation, 'Successful submission Message does not exist').to.be.false;
    });


 it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
    browser.setValue("[name='first_name']",'Sarah');
    browser.setValue("[name='email']", 'sarah_woods@mail.com');
    browser.click("[type='submit']");

    var successfulContactConfirmation = browser.isExisting('#contact_reply h1');
    expect(successfulContactConfirmation, 'Successful submission Message does not exist').to.be.false;
    });

 it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
    browser.setValue("[name='first_name']",'Jim');
    browser.setValue("[name='last_name']",'Jomes');
    browser.click("[type='submit']");

    var errorText = browser.getText('body');
    expect(errorText).to.include('Error: all fields are required');

    var errorText = browser.isVisible('body');
    expect(errorText, 'Error message is not visible').to.be.true;
    });
});
```

2. We now need to add a function for an unsuccessful contact us form submission
    a. Remember, if all fields are not provided on the contact us form, we are not shown the "Thank you for your message!" page but are instead shown the following:



    b. We are going to use the "Error: all fields are required" message so that we know we are taken to the error page, indicating a successful submission did not take place
3. Add the following function to your code:

```
46      function confirmUnsuccessfulSubmission() {
47          var validateSubmissionHeader = browser.waitUntil(function() {
48              return browser.getText(unsuccessfulSubmissionSelector) == 'Error: all fields are required'
49          }, 3000)
50          expect(browser.getText(unsuccessfulSubmissionSelector)).to.include('Error: all fields are required');
51      }
52
53
```

    a. This is very similar to the successful submission function, but the key differences are;
        i. We use a different variable name
        ii. The getText parameter points to unsuccessfulSubmissionSelector which uses a different variable from line 16 of our code that contains a locator of the error message element
        iii. We expect the text to equal (==) the error message output
        iv. We then amend the *expect* assertion to include the text of the error message output
4. Now we have completed our functions and are ready to start amending our tests to use the functions!
5. Let's focus on the first "it" code block, which currently looks like this:

```
48  contactusDetails.forEach(function (contactDetail) {
49    it('Should be able to submit a successful submission via contact us form', function(done) {
50      browser.submitDataViaContactUsForm('joe', 'Blogs', contactDetail.email, contactDetail.body);
51
52      var successfulContactConfirmation = browser.isExisting('#contact_reply h1');
53      expect(successfulContactConfirmation, 'Successful submission Message does not exist').to.be.true;
54
55      var successfulSubmission = browser.getText('#contact_reply h1');
56      expect(successfulSubmission).to.equal('Thank You for your Message!');
57          })
58      });
59
```

6. We can now change this code to make use of the functions, as shown:

```
55  contactusDetails.forEach(function (contactDetail) {
56    it.only('Should be able to submit a successful submission via contact us form', function(done) {
57      setFirstName('joe');
58      setLastName('Blogs');
59      setEmailAddress(contactDetail.email);
60      setComments(contactDetail.body);
61      clickSubmitButton();
62      confirmSuccessfulSubmission();
63      });
64
```

7. We now call the function and provide a parameter value e.g. 'joe' and 'Blogs' and these values will be used when the functions get called
8. Also notice how much code we were able to remove
9. We then amend our second "it" test blocks in a similar way, so that your overall test should look like the following:

ORIG_contactUsTest.js × FINAL_contactUsTest.js ×

```
var request = require('sync-request');

beforeEach(function() {
    browser.url('/Contact-Us/contactus.html');
})

describe('Test Contact Us form WebdriverUni', function() {
    var res = request('GET', 'http://jsonplaceholder.typicode.com/posts/1/comments');
    var contactusDetails = JSON.parse(res.getBody().toString('utf8'));

    var firstNameSelector = "[name='first_name']";
    var lastNameSelector = "[name='last_name']";
    var emailAddressSelector = "[name='email']";
    var commentsSelector = "textarea";
    var successfulSubmissionSelector = "#contact_reply h1";
    var unsuccessfulSubmissionSelector = "body";
    var submitButtonSelector = "[type='submit']";

    function setFirstName(firstName) {
        return browser.setValue(firstNameSelector, firstName);
    }

    function setLastName(lastName) {
        return browser.setValue(lastNameSelector, lastName);
    }

    function setEmailAddress(emailAddress) {
        return browser.setValue(emailAddressSelector, emailAddress);
    }

    function setComments(comments) {
        return browser.setValue(commentsSelector, comments);
    }

    function clickSubmitButton() {
        return browser.click(submitButtonSelector);
    }

    function confirmSuccessfulSubmission() {
        var validateSubmissionHeader = browser.waitUntil(function() {
            return browser.getText(successfulSubmissionSelector) == 'Thank You for your Message!'
        }, 3000)
        expect(validateSubmissionHeader, 'Successful Submission Message does not Exist!').to.be.true;
    }

    function confirmUnsuccessfulSubmission() {
        var validateSubmissionHeader = browser.waitUntil(function() {
            return browser.getText(unsuccessfulSubmissionSelector) == 'Error: all fields are required'
        }, 3000)
        expect(browser.getText(unsuccessfulSubmissionSelector)).to.include('Error: all fields are required');
    }


contactusDetails.forEach(function (contactDetail) {
  it.only('Should be able to submit a successful submission via contact us form', function(done) {
    setFirstName('joe');
    setLastName('Blogs');
    setEmailAddress(contactDetail.email);
    setComments(contactDetail.body);
    clickSubmitButton();
    confirmSuccessfulSubmission();
    });

  it.only('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
    setFirstName('Mike');
    setLastName('Woods');
    setEmailAddress('mike_woods@mail.com');
    clickSubmitButton();
    confirmUnsuccessfulSubmission();
    });


  it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
    browser.setValue("[name='first_name']",'Sarah');
    browser.setValue("[name='email']", 'sarah_woods@mail.com');
    browser.click("[type='submit']");

    var successfulContactConfirmation = browser.isExisting('#contact_reply h1');
    expect(successfulContactConfirmation, 'Successful submission Message does not exist').to.be.false;
    });

  it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
    browser.setValue("[name='first_name']",'Jim');
    browser.setValue("[name='last_name']",'Jomes');
    browser.click("[type='submit']");

    var errorText = browser.getText('body');
    expect(errorText).to.include('Error: all fields are required');

    var errorText = browser.isVisible('body');
    expect(errorText, 'Error message is not visible').to.be.true;
    });
});
```

We continue with phase one in the next lecture

Lecture 76 - Page Object Model (POM) - Phase 1 - Part 4

In this lecture, we continue improving our contactUsTest.js file from the work we have completed from Part 3.

Key Points:

- We have now completed all the functions that we need
- We have amended our first and second “it” test blocks to now use the functions
- This has reduced the amount of code by a fair amount
- We now need to finish amending our third and fourth “it” test blocks to use the functions

Instructions:

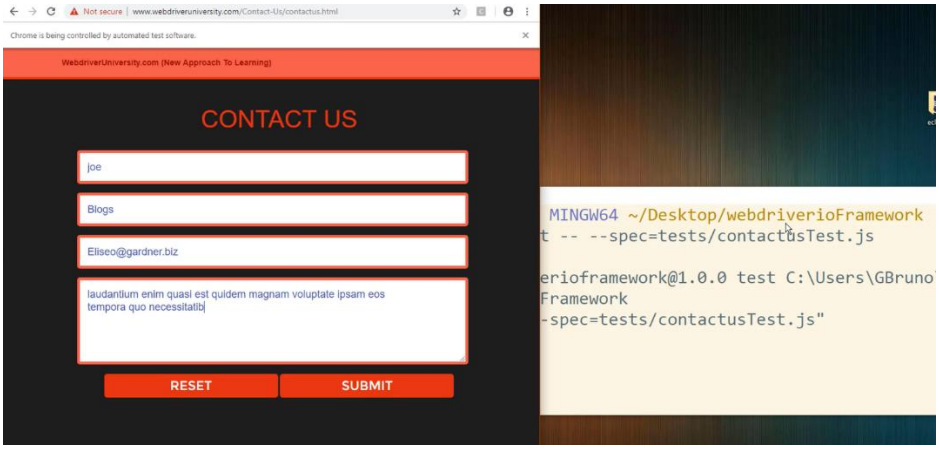
1. Now we need to make changes to our third and fourth “it” test code blocks:
2. These currently look like this:

```
73
74  it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
75    browser.setValue("[name='first_name']",'Sarah');
76    browser.setValue("[name='email']", 'sarah_woods@mail.com');
77    browser.click("[type='submit']");
78
79    var successfulContactConfirmation = browser.isExisting('#contact_reply h1');
80    expect(successfulContactConfirmation, 'Successful submission Message does not exist').to.be.false;
81    });
82
83  it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
84    browser.setValue("[name='first_name']",'Jim');
85    browser.setValue("[name='last_name']",'Jomes');
86    browser.click("[type='submit']");
87
88    var errorText = browser.getText('body');
89    expect(errorText).to.include('Error: all fields are required');
90
91    var errorText = browser.isVisible('body');
92    expect(errorText, 'Error message is not visible').to.be.true;
93    });
94  });
```

3. As you can see, they currently use the old method where we explicitly state the locator within the test block
4. Change you code so that it looks like the following:

```
75
76  it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
77    setFirstName('Sarah');
78    setEmailAddress('sarah_woods@mail.com');
79    clickSubmitButton();
80    confirmUnsuccessfulSubmission();
81    });
82
83  it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
84    setLastName('Jomes');
85    setEmailAddress('sarah_jomes@mail.com');
86    clickSubmitButton();
87    confirmUnsuccessfulSubmission();
88    });
89  });
```

5. These tests now use the functions (similar to the last lecture)
6. Once you have made the changes above, save your file (if you downloaded the code from the Udemy resource section, make sure to remove the prefix ORIG_ or FINAL_ from the file name) in the “tests” folder directory
7. Then, close Sublime Text and open up GitBash / iTerm2 and navigate to our webdriverioFramework directory
8. Run the following command:
9. **npm test -- -- specs=tests/contactUsTest.js** + press enter
10. You should see the tests run, as shown below:



11. Notice how the data is still passed to the test during execution using our recent code changes

12. You should then see that 8 tests have passed, as all our tests from out contactUsTest.js file how now executed and completed successfully.

We continue with phase one in the next lecture

Lecture 77 - Page Object Model (POM) - Phase 1 - Part 5

In this lecture, we check to see if one of our assertions work correctly. We do this by amending the code to give an unexpected value for one of the "it" tests. This is to show you what happens when a value is returned that we were not expecting.

Key Points:

- We have now completed all the functions that we need
- We have now updated all "it" test code blocks
- We are now temporary going to amend one of the functions so that we receive an unexpected value during one of the "it" tests

Instructions:

1. Take a look at the current confirmUnsuccessfulSubmission function:

```
46      function confirmUnsuccessfulSubmission() {
47          var validateSubmissionHeader = browser.waitUntil(function() {
48              return browser.getText(unsuccessfulSubmissionSelector) == 'Error: all fields are required'
49          }, 3000)
50          expect(browser.getText(unsuccessfulSubmissionSelector)).to.include('Error: all fields are required');
51      }
52
```

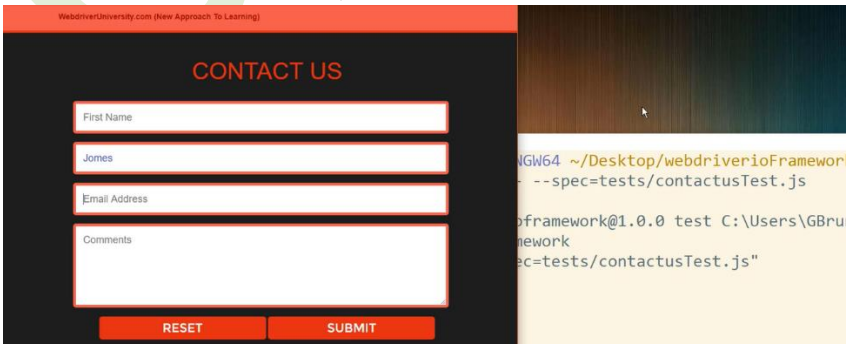
2. We are going to change this to:

```
46      function confirmUnsuccessfulSubmission() {
47          var validateSubmissionHeader = browser.waitUntil(function() {
48              return browser.getText(unsuccessfulSubmissionSelector) == 'Error: all fields are required'
49          }, 3000)
50          expect(browser.getText(unsuccessfulSubmissionSelector)).to.include('THIS CAN BE ANYTHING');
51      }
52
```

3. Now focus on the last "it" test by using the .only command so that only the last "it" test is executed:

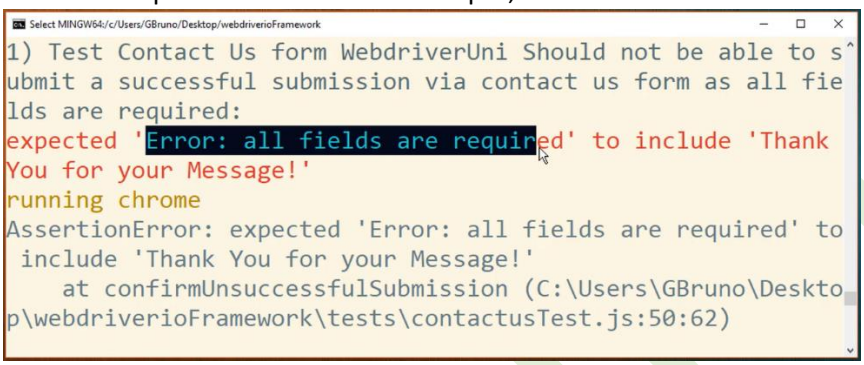
```
83      it.only('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
84        setLastName('Jomes');
85        setEmailAddress('sarah_Jomes@mail.com');
86        clickSubmitButton();
87        confirmUnsuccessfulSubmission();
88        });
89  });
```

4. Now save your file and close Sublime Text
5. Open up GitBash / iTerm2 and navigate to our webdriverioFramework directory
6. Run the following command:
7. **npm test -- -- specs=tests/contactUsTest.js** + press enter
8. You should see the tests run, as shown below:

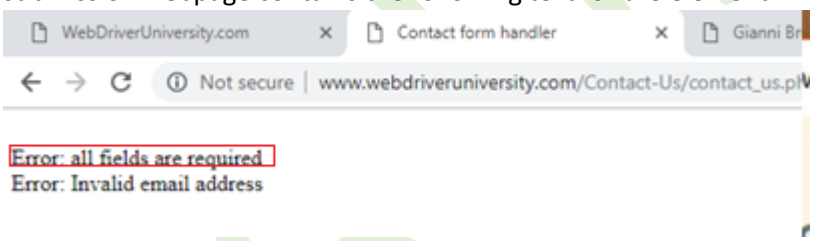


9. But if we look at the console output, we should see the following:

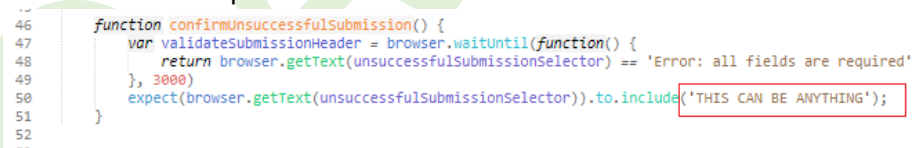
10. If we scroll up and review the error output, we should see:



11. This is because the function has been amended and for unsuccessful contact us form submission webpage contains the following text for the element in focus:



12. But the function expects:

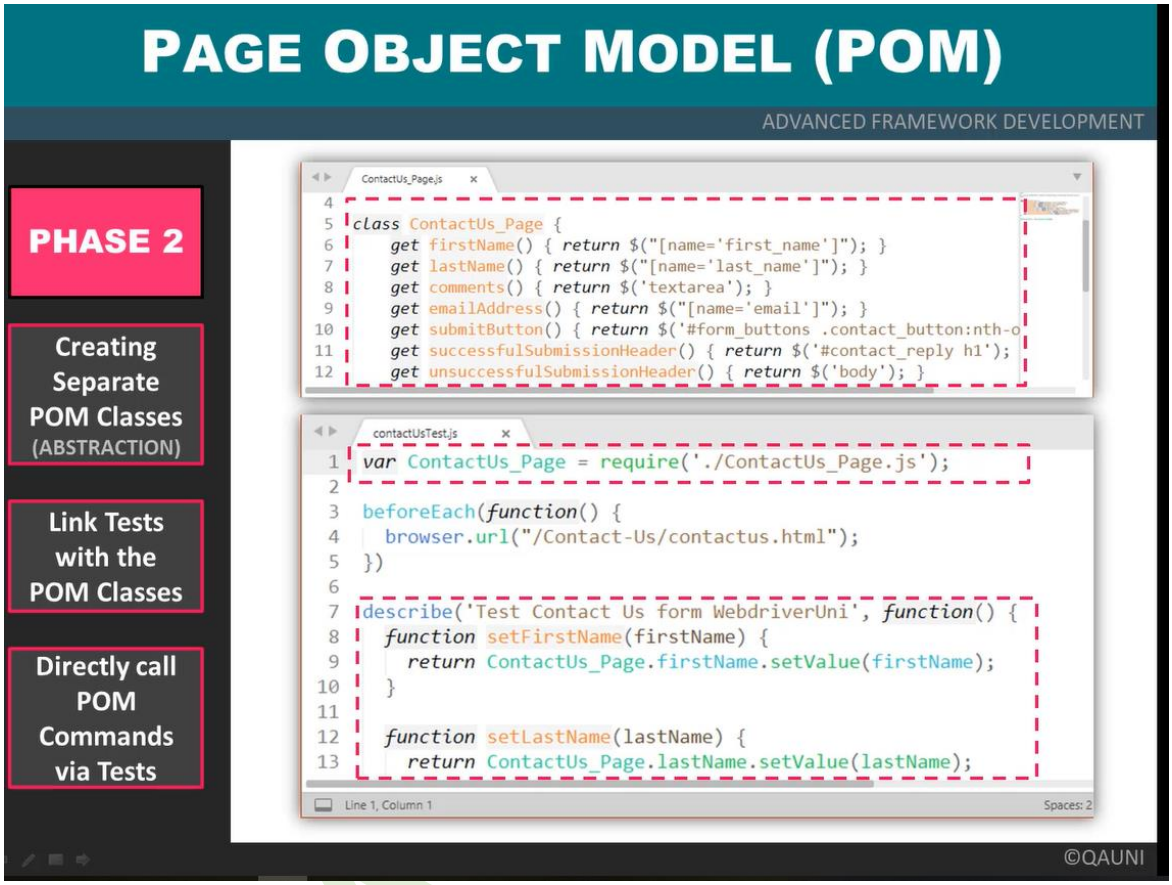


13. Because there is a mismatch between the element text and the assertion expected output for unsuccessful submissions, an error is thrown, and the test fails.
14. Change the code back (or download the FINAL_contactUsForm.js file which doesn't contain the temporary change we went through in this lecture).

I hope phase one has shown you the benefits of writing good, maintainable code. If the locator values change in the future, all we have to do is update the variables values in lines 11-17 and these values will be inherited into your test blocks.

Lecture 78 - Page Object Model (POM) - Phase 2 - Part 1

In this lecture, we move onto phase 2. In this phase, look at creating separate POM classes, link tests with the POM classes and directly call POM commands in our tests.



What is abstraction?

Stackify.com does a great job at explaining abstraction and you can read more about it here:

https://stackify.com/oop-concept-abstraction/

Some of the key points are:

Abstraction is one of the key concepts of object-oriented programming (OOP) languages. Its main goal is to handle complexity by hiding unnecessary details from the user. That enables the user to implement more complex logic on top of the provided abstraction without understanding or even thinking about all the hidden complexity.

That's a very generic concept that's not limited to object-oriented programming. You can find it everywhere in the real world.

Lecture 79 - Page Object Model (POM) - Phase 2 - Part 2

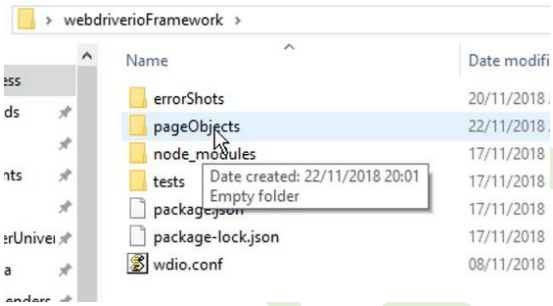
In this lecture, we start making changes to our code. We will again be using the contactUsTest.js file to make further changes.

Key Points:

- Our contactUsTest.js file now contains the changes from Phase 1
- We are now going to create a new folder that will store our Javascript classes
- We are going to transfer the locator variables to their own class file and we can then link our tests to it
- This will make it far easier to maintain our code going forward

Instructions:

1. Either download the file from the Udemy resources section (prefix ORIG_) or use your local copy of the contactUsTest.js file from within your "tests" folder
    a. If you do download the file from Udemy, ensure to rename it (remove the prefix) and place it inside your "tests" folder
    b. If a file already exists with the same name inside your "tests" folder, place the old version into a separate folder for safe-keeping, if you choose to keep a backup
2. We need to create a new folder to store our page objects (rather than having numerous files inside our "tests" folder – which is messy



3. Create a folder called pageObjects as shown above
4. This is where we are going to store all our Javascript classes
5. Go to Sublime Text and open up the contactUsTest.js file

Explanation of the next few steps:

If we look at our current file, we can see the following:

```
10
11      var firstNameSelector = "[name='first_name']";
12      var lastNameSelector = "[name='last_name']";
13      var emailAddressSelector = "[name='email']";
14      var commentsSelector = "textarea";
15      var successfulSubmissionSelector = "#contact_reply h1";
16      var unsuccessfulSubmissionSelector = "body";
17      var submitButtonSelector = "[type='submit']";
18
19      function setFirstName(firstName) {
20          return browser.setValue(firstNameSelector, firstName);
21      }
```

We intend to place the selectors into their own file. This is because if we have multiple test files and we have selectors located inside of each, we would need to still update each individual file in order to use the updated selectors.

The changes in Phase 1 has made improvements, since we only need to update the variable values rather than each individual "it" test. Now we intend to improve this further by removing the selectors completely from each individual test file and linking to it instead. This will mean we just need to update the one file if changes are required going forward, instead of each individual test file.

6. Create a new file in Sublime Text and save it into your newly created pageObjects folder. Give the file a name of **ContactUs_Page.js**
7. We are going to use this file to store to locator variables
8. Type the following into the file:

```
class ContactUs_Page {
    get firstName() { return $("[name='first_name']");}
    get lastName() { return $("[name='last_name']");}
    get comments() { return $("textarea");}
    get emailAddress() { return $("[name='email']");}
    get submitButton() { return $("[type='submit']");}
    get successfulSubmissionHeader() { return $("#contact_reply h1");}
    get unsuccessfulSubmissionHeader() return $("body");}
}

module.exports = new ContactUs_Page();
```

9. What we have done here is:
    a. Created a new class called ContactUs_Page
    b. That contains a number of get methods
    c. Each of these get methods has a name similar to the variables from our contactUsTest.js file
    d. For each one, we then return a value which is the locator value
    e. We then use module.exports = new ContactUs_Page(); which means this class can now be exported and used by other files

We continue with phase two in the next lecture

Lecture 80 - Page Object Model (POM) - Phase 2 - Part 3

In this lecture, we continue from the last lecture and continue to improve our contactUsTest.js file by using the newly created ContactUs_Page.js class file.

Key Points:

- We have now created a class file that contains our selectors
- We have named it ContactUs_Page.js
- We have placed this file in the newly created classObjects folder
- We now need to amend our contactUsTest.js folder to make use of the ContactUs_Page.js file
- We also need to change the WDIO.conf file to exclude the classObjects folder so that these files are not executed when we run a test

Instructions:

1. We now need to amend our WDIO.conf file – open this file using Sublime Text
2. If we look at lines 22-28, you can see there is a spec section where we have placed the path to our tests files:

```
22    specs: [
23        './tests/**/*.js'
24    ],
25    // Patterns to exclude.
26    exclude: [
27        // 'path/to/excluded/files'
28    ],
```

3. We need to make modifications to this section, by making the following changes:

```
22    specs: [
23        'tests/*.js'
24    ],
25    // Patterns to exclude.
26    exclude: [
27        'pageObjects/*_Page.js'
28    ],
29    //
```

   a. We have shortened the specs section code (which we could have done earlier), but this code basically goes from the base directory (webdriverFramework folder) then to the tests folder where our test files are located. The star indicates a wildcard, which means if we run 'npm test' using GitBash or iTerm2 it will execute all tests in the tests folder.
   b. We then add a line to the exclude section. We have placed the path of our newly created pageObjects folder. We have added this here because we do not want to execute class files when we run a test because they are not test cases. We use these files in our test code, but they are not specifically tests, so there's no need to run them. By placing the path here and by using *_Page.js, it means all files that follow this file name prefix will be excluded from running when a test starts.
4. Once you've made the changes above, save the WDIO.conf file and close the tab
5. Open up the contactUsTest.js file using Sublime Text
6. Remove the following lines:

contactUsTest.js

```
var request = require('sync-request');

beforeEach(function() {
    browser.url('/Contact-Us/contactus.html');
})

describe('Test Contact Us form WebdriverUni', function() {
    var res = request('GET', 'http://jsonplaceholder.typicode.com/posts/1/comments');
    var contactusDetails = JSON.parse(res.getBody().toString('utf8'));

    var firstNameSelector = "[name='first_name']";
    var lastNameSelector = "[name='last_name']";
    var emailAddressSelector = "[name='email']";
    var commentsSelector = "textarea";
    var successfulSubmissionSelector = "#contact_reply h1";
    var unsuccessfulSubmissionSelector = "body";
    var submitButtonSelector = "[type='submit']";

    function setFirstName(firstName) {
        return browser.setValue(firstNameSelector, firstName);
    }
```

7. Now we need to link the page object to this file by adding the following line:

contactUsTest.js

```
var ContactUs_Page = require("./pageObjects/ContactUs_Page.js");

beforeEach(function() {
    browser.url('/Contact-Us/contactus.html');
})
```

We continue with phase two in the next lecture

Lecture 81 - Page Object Model (POM) - Phase 2 - Part 4

In this lecture, we continue from the last lecture and continue to improve our contactUsTest.js file by amending the functions to use the ContactUs_Page.js file.

Key Points:

- We now have the ContactUs_Page.js class file in place
- We have removed all redundant code from our contactUsTest.js file
- We now need to amend our functions to use the ContactUs_Page.js class file

Instructions:

1. Our contactUsTest.js file should now look like this:

ORIG_contactUsTest.js

```
var ContactUs_Page = require("./pageObjects/ContactUs_Page.js");

beforeEach(function() {
    browser.url('/Contact-Us/contactus.html');
})

describe('Test Contact Us form WebdriverUni', function() {
    function setFirstName(firstName) {
        return browser.setValue(firstNameSelector, firstName);
    }

    function setLastName(lastName) {
        return browser.setValue(lastNameSelector, lastName);
    }

    function setEmailAddress(emailAddress) {
        return browser.setValue(emailAddressSelector, emailAddress);
    }

    function setComments(comments) {
        return browser.setValue(commentsSelector, comments);
    }

    function clickSubmitButton() {
        return browser.click(submitButtonSelector);
    }

    function confirmSuccessfulSubmission() {
        var validateSubmissionHeader = browser.waitUntil(function() {
            return browser.getText(successfulSubmissionSelector) == 'Thank You for your Message!'
        }, 3000)
        expect(validateSubmissionHeader, 'Successful Submission Message does not Exist!').to.be.true;
    }

    function confirmUnsuccessfulSubmission() {
        var validateSubmissionHeader = browser.waitUntil(function() {
            return browser.getText(unsuccessfulSubmissionSelector) == 'Error: all fields are required'
        }, 3000)
        expect(browser.getText(unsuccessfulSubmissionSelector)).to.include('Error: all fields are required');
    }


contactusDetails.forEach(function (contactDetail) {
  it('Should be able to submit a successful submission via contact us form', function(done) {
    setFirstName('joe');
    setLastName('Blogs');
    setEmailAddress(contactDetail.email);
    setComments(contactDetail.body);
    clickSubmitButton();
    confirmSuccessfulSubmission();
    });
  });


  it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
    setFirstName('Mike');
    setLastName('Woods');
    setEmailAddress('mike_woods@mail.com');
    clickSubmitButton();
    confirmUnsuccessfulSubmission();
    });


  it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
    setFirstName('Sarah');
    setEmailAddress('sarah_woods@mail.com');
    clickSubmitButton();
    confirmUnsuccessfulSubmission();
    });

  it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
    setLastName('Jomes');
    setEmailAddress('sarah_Jomes@mail.com');
    clickSubmitButton();
    confirmUnsuccessfulSubmission();
    });
});
```

2. We now need to start changing the functions so that they use the get methods from the ContactUs_Page.js file.

3. Change your first function to the following:

```
8       function setFirstName(firstName) {
9           return ContactUs_Page.firstName.setValue(firstName);
10      }
```

   a. We have used the require variable (set on line 1) and then called .firstName (which is a get method)
   b. We then setValue and pass it a parameter (firstName) which we give a value in our "it" test blocks

4. We follow this process for all the remaining functions. Your functions should look like this once all changes are completed:

```
1   var ContactUs_Page = require("./pageObjects/ContactUs_Page.js");
2
3   beforeEach(function() {
4       browser.url('/Contact-Us/contactus.html');
5   })
6
7   describe('Test Contact Us form WebdriverUni', function() {
8       function setFirstName(firstName) {
9           return ContactUs_Page.firstName.setValue(firstName);
10      }
11
12      function setLastName(lastName) {
13          return ContactUs_Page.lastName.setValue(lastName);
14      }
15
16      function setEmailAddress(emailAddress) {
17          return ContactUs_Page.emailAddress.setValue(emailAddress);
18      }
19
20      function setComments(comments) {
21          return ContactUs_Page.comments.setValue(comments);
22      }
23
24      function clickSubmitButton() {
25          return ContactUs_Page.submitButton.click();
26      }
27
28      function confirmSuccessfulSubmission() {
29          var validateSubmissionHeader = browser.waitUntil(function() {
30              return ContactUs_Page.successfulSubmissionHeader.getText() == 'Thank You for your Message!'
31          }, 3000)
32          expect(validateSubmissionHeader, 'Successful Submission Message does not Exist!').to.be.true;
33      }
34
35      function confirmUnsuccessfulSubmission() {
36          var validateSubmissionHeader = browser.waitUntil(function() {
37              return ContactUs_Page.unsuccessfulSubmissionHeader.getText() == 'Error: all fields are required'
38          }, 3000)
39          expect(ContactUs_Page.unsuccessfulSubmissionHeader.getText()).to.include('Error: all fields are required');
40      }
41
```

We continue with phase two in the next lecture

Lecture 82 - Page Object Model (POM) - Phase 2 - Part 5

In this lecture, we continue from the last lecture and make a few minor adjustments to our WDIO file and our contactUsTest.js file. We also go through our code to ensure it's correct before attempting to run a test.

Key Points:

- We have now created our page object model (POM)
- We have made amendments to our test class
- We have made changes to our WDIO file
- We now need to make some additional (minor) adjustments to our WDIO file and our test file – once we've completed a review

Instructions:

1. We make the following change to the spec section of the WDIO.conf file:
2. Open your WDIO.conf file using sublime text
3. Add the following ./ to your spec directory references:

```
22      specs: [
23          './tests/*.js'
24      ],
25      // Patterns to exclude.
26      exclude: [
27          './pageObjects/*_Page.js'
28      ],
```

4. What does the ./ do?
    a. A dot slash is a dot followed immediately by a forward slash (./). It is used in Linux and Unix to execute a compiled program in the current directory. For example, if you had an executable file in the current directory called "hope" and typed "hope" at the command line, you would get an error, such as "-bash: hope: command not found." The error occurs because "hope" is neither a built-in command or an executable file found in your PATH environment variable. When listing files in a directory a ./ will be listed first, this represents the current directory. The ../ represents the parent directory
5. Next we review our ContactUs_Page.js page objects file. Open it using Sublime Text:

wdio.conf.js | ContactUs_Page.js

```
1  class ContactUs_Page {
2      get firstName() { return $("[name='first_name']");}
3      get lastName() { return $("[name='last_name']");}
4      get comments() { return $("textarea");}
5      get emailAddress() { return $("[name='email']");}
6      get submitButton() { return $("[type='submit']");}
7      get successfulSubmissionHeader() { return $("#contact_reply h1");}
8      get unsuccessfulSubmissionHeader() return $("body");}
9  }
10
11 module.exports = new ContactUs_Page();
```

6. Notice how the return on the last line is indicated in a different colour. This indicates there is something different here compared to the other lines. Notice we are missing a { before the return. Add this to line 8, just before the return word, so:

```
8      get unsuccessfulSubmissionHeader() { return $("body");}
```

7. Save your file once you've made the change above

8. Now we need to review our contactUsTest.js test file. Open it with Sublime Text
9. Notice on line 1 we have the following:

```
wdio.conf.js | ContactUs_Page.js | contactUsTest.js
1  var ContactUs_Page = require("pageObjects/ContactUs_Page.js");
```

10. We need to add ../ before the pageObjects directory reference (this tells our test where to look for when trying to find the ContactUs_Page.js file)
11. Once you've made the change, your file should look like:

```
wdio.conf.js | ContactUs_Page.js | contactUsTest.js
1  var ContactUs_Page = require("../pageObjects/ContactUs_Page.js");
2
```

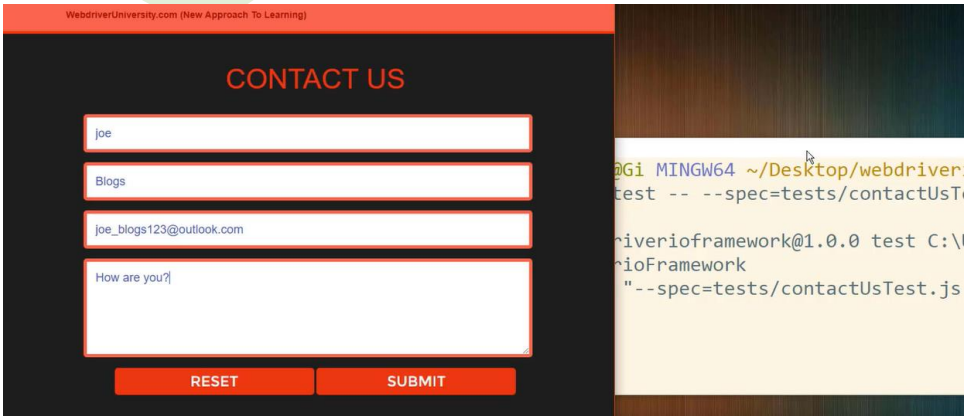
12. Keeping scrolling down in the file until you come across the following section:

```
44 contactusDetails.forEach(function (contactDetail) {
45   it('Should be able to submit a successful submission via contact us form', function(done) {
46     setFirstName('joe');
47     setLastName('Blogs');
48     setEmailAddress(contactDetail.email);
49     setComments(contactDetail.body);
50     clickSubmitButton();
51     confirmSuccessfulSubmission();
52     });
53    });
54
```

13. Notice on line 44 we have a forEach loop and as we are not using Sync requests in this test anymore, we can remove line 44.
14. We also need to remove the sync request parameters on lines 48 and 49
15. We replace this section with the following:

```
43
44   it('Should be able to submit a successful submission via contact us form', function(done) {
45     setFirstName('joe');
46     setLastName('Blogs');
47     setEmailAddress('joe_blogs123@outlook.com');
48     setComments('How are you?');
49     clickSubmitButton();
50     confirmSuccessfulSubmission();
51     });
```

    a. We are now passing actual values for setEmailAddress and setComments instead of inheriting these values from the JSON file from past lectures
16. The remainder of our file now looks ok.
17. Save your files and close Sublime Text
18. Open up GitBash / iTerm2 and navigate to our webdriverioFramework directory
19. Run the following command:
20. **npm test -- -- specs=tests/contactUsTest.js** + press enter
21. You should see the tests run, as shown below:



22. You should see each of our "it" tests run

23. And if we review the console output window once the test has completed, we should see:

24. All four tests have passed successfully, and no issues are reported!

We continue with phase two in the next lecture

Lecture 83 - Page Object Model (POM) - Phase 2 - Part 6

In this lecture, we review the changes that we have made throughout phase 2.

As we have already documented our changes in lectures 76-80, please refer back to the documentation from the past lectures above for a recap.

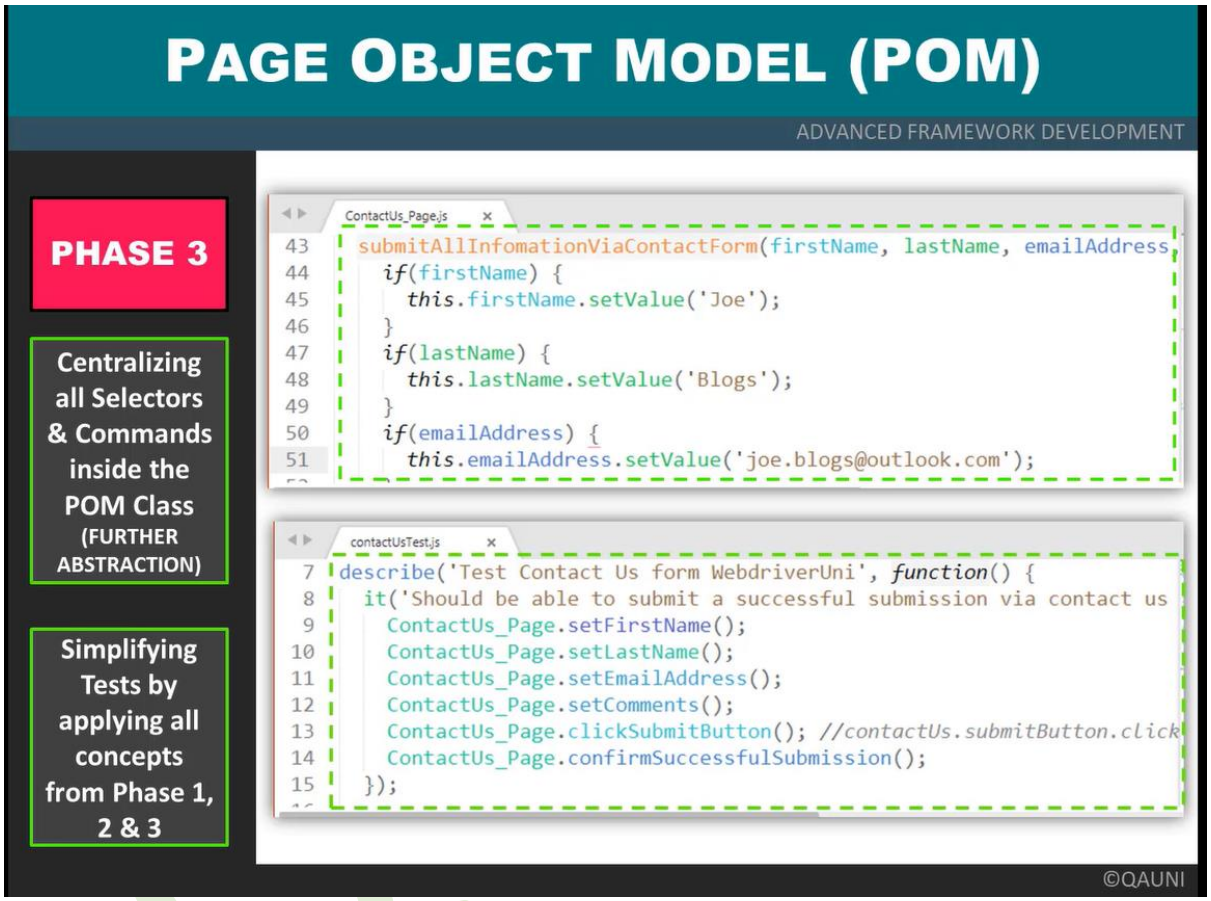
I would also advise reviewing this lecture video for a quick recap of the changes made.

Lecture 84 - Page Object Model (POM) - Phase 3 - Part 1

In this lecture, we start Phase 3 and look at centralizing all sectors and commands inside the POM class (further abstraction). We also look at simplifying our tests even further.

Key Points:

- We have now completed a number of improvements from Phase 1 and 2
- It's now easier to maintain our locators, as we can simply update the locator values in the POM class file
- We now look at further improvements to our code



We continue with phase three in the next lecture

Lecture 85 - Page Object Model (POM) - Phase 3 - Part 2

In this lecture, we begin amending our contactUsTest.js file and our ContactUs_Page.js file by introducing further abstraction.

Key Points:

- Next, we are going to review the contactUsTest.js file to see what we can instead move into the ContactUs_Page.js
- Currently, our contactUsTest.js file contains a number of functions, but what happens if we have a number of additional test files that also uses webdriveruniversity.com and needs to use the functions? We would need to repeat the code for each individual file. Instead we should look at moving the functions to the ContactUs_Page.js class file.

Instructions:

1. Open your contactUsTest.js and ContactUs_Page.js files in Sublime Text (you can also download the files from the Udemy resources section)
2. If you review the contactUsTest.js file, you will see we currently have a number of functions present:

```
var ContactUs_Page = require("../pageObjects/ContactUs_Page.js");

beforeEach(function() {
    browser.url('/Contact-Us/contactus.html');
})

describe('Test Contact Us form WebdriverUni', function() {
    function setFirstName(firstName) {
        return ContactUs_Page.firstName.setValue(firstName);
    }

    function setLastName(lastName) {
        return ContactUs_Page.lastName.setValue(lastName);
    }

    function setEmailAddress(emailAddress) {
        return ContactUs_Page.emailAddress.setValue(emailAddress);
    }

    function setComments(comments) {
        return ContactUs_Page.comments.setValue(comments);
    }

    function clickSubmitButton() {
        return ContactUs_Page.submitButton.click();
    }

    function confirmSuccessfulSubmission() {
        var validateSubmissionHeader = browser.waitUntil(function() {
            return ContactUs_Page.successfulSubmissionHeader.getText() == 'Thank You for your Message!'
        }, 3000)
        expect(validateSubmissionHeader, 'Successful Submission Message does not Exist!').to.be.true;
    }

    function confirmUnsuccessfulSubmission() {
        var validateSubmissionHeader = browser.waitUntil(function() {
            return ContactUs_Page.unsuccessfulSubmissionHeader.getText() == 'Error: all fields are required'
        }, 3000)
        expect(ContactUs_Page.unsuccessfulSubmissionHeader.getText()).to.include('Error: all fields are required');
    }


```

3. If you review the ContactUs_Page.js file, you will see it currently looks like this:

```
File  Edit  Selection  Find  View  Goto  Tools  Project  Preferences  Help
ContactUs_Page.js ×   contactUsTest.js ×
class ContactUs_Page {
    get firstName() { return $("[name='first_name']");}
    get lastName() { return $("[name='last_name']");}
    get comments() { return $("textarea");}
    get emailAddress() { return $("[name='email']");}
    get submitButton() { return $("[type='submit']");}
    get successfulSubmissionHeader() { return $("#contact_reply h1");}
    get unsuccessfulSubmissionHeader() { return $("body");}
}

module.exports = new ContactUs_Page();
```

4. Cut (CTRL + V) all the functions in the contactUsTest.js file and place them into your ContactUs_Page.js file, like so:

```
class ContactUs_Page {
    get firstName() { return $("[name='first_name']");}
    get lastName() { return $("[name='last_name']");}
    get comments() { return $("textarea");}
    get emailAddress() { return $("[name='email']");}
    get submitButton() { return $("[type='submit']");}
    get successfulSubmissionHeader() { return $("#contact_reply h1");}
    get unsuccessfulSubmissionHeader() { return $("body");}
}

    function setFirstName(firstName) {
        return ContactUs_Page.firstName.setValue(firstName);
    }

    function setLastName(lastName) {
        return ContactUs_Page.lastName.setValue(lastName);
    }

    function setEmailAddress(emailAddress) {
        return ContactUs_Page.emailAddress.setValue(emailAddress);
    }

    function setComments(comments) {
        return ContactUs_Page.comments.setValue(comments);
    }

    function clickSubmitButton() {
        return ContactUs_Page.submitButton.click();
    }

    function confirmSuccessfulSubmission() {
        var validateSubmissionHeader = browser.waitUntil(function() {
            return ContactUs_Page.successfulSubmissionHeader.getText() == 'Thank You for your Message!'
        }, 3000)
        expect(validateSubmissionHeader, 'Successful Submission Message does not Exist!').to.be.true;
    }

    function confirmUnsuccessfulSubmission() {
        var validateSubmissionHeader = browser.waitUntil(function() {
            return ContactUs_Page.unsuccessfulSubmissionHeader.getText() == 'Error: all fields are required'
        }, 3000)
        expect(ContactUs_Page.unsuccessfulSubmissionHeader.getText()).to.include('Error: all fields are required');
    }

module.exports = new ContactUs_Page();
```

5. We now need to make some additional changes to the ContactUs_Page.js file. Previously, each function had to call ContactUs_Page but the functions are now inside this file, meaning we can make the following changes:

```
class ContactUs_Page {
    get firstName() { return $("[name='first_name']");}
    get lastName() { return $("[name='last_name']");}
    get comments() { return $("textarea");}
    get emailAddress() { return $("[name='email']");}
    get submitButton() { return $("[type='submit']");}
    get successfulSubmissionHeader() { return $("#contact_reply h1");}
    get unsuccessfulSubmissionHeader() { return $("body");}
}

function setFirstName(firstName) {
        return this.firstName.setValue(firstName);
    }

    function setLastName(lastName) {
        return this.lastName.setValue(lastName);
    }

    function setEmailAddress(emailAddress) {
        return this.emailAddress.setValue(emailAddress);
    }

    function setComments(comments) {
        return this.comments.setValue(comments);
    }

    function clickSubmitButton() {
        return this.submitButton.click();
    }

    function confirmSuccessfulSubmission() {
        var validateSubmissionHeader = browser.waitUntil(function() {
            return this.successfulSubmissionHeader.getText() == 'Thank You for your Message!'
        }, 3000)
        expect(validateSubmissionHeader, 'Successful Submission Message does not Exist!').to.be.true;
    }

    function confirmUnsuccessfulSubmission() {
        var validateSubmissionHeader = browser.waitUntil(function() {
            return this.unsuccessfulSubmissionHeader.getText() == 'Error: all fields are required'
        }, 3000)
        expect(this.unsuccessfulSubmissionHeader.getText()).to.include('Error: all fields are required');
    }

module.exports = new ContactUs_Page();
```

We continue with phase three in the next lecture

Lecture 86 - Page Object Model (POM) - Phase 3 - Part 3

In this lecture, we make a further improvement to our ContactUs_Page.js file by creating a new method that handles a test scenario.

Key Points:

- We now have our functions in the ContactUs_Page.js page
- We now need to create a method that will contain the existing functionality of the individual functions
- This method uses IF statements to see if data has been provided (e.g. a first name or last name) and the code within the IF statement would get executed if true

Instructions:

1. Open up your ContactUs_Page.js file using Sublime Text. It should currently look like this:

ORIG_ContactUs_Page.js

```
class ContactUs_Page {
    get firstName() { return $("[name='first_name']");}
    get lastName() { return $("[name='last_name']");}
    get comments() { return $("textarea");}
    get emailAddress() { return $("[name='email']");}
    get submitButton() { return $("[type='submit']");}
    get successfulSubmissionHeader() { return $("#contact_reply h1");}
    get unsuccessfulSubmissionHeader() { return $("body");}
}

function setFirstName(firstName) {
        return this.firstName.setValue(firstName);
    }

    function setLastName(lastName) {
        return this.lastName.setValue(lastName);
    }

    function setEmailAddress(emailAddress) {
        return this.emailAddress.setValue(emailAddress);
    }

    function setComments(comments) {
        return this.comments.setValue(comments);
    }

    function clickSubmitButton() {
        return this.submitButton.click();
    }

    function confirmSuccessfulSubmission() {
        var validateSubmissionHeader = browser.waitUntil(function() {
            return this.successfulSubmissionHeader.getText() == 'Thank You for your Message!'
        }, 3000)
        expect(validateSubmissionHeader, 'Successful Submission Message does not Exist!').to.be.true;
    }

    function confirmUnsuccessfulSubmission() {
        var validateSubmissionHeader = browser.waitUntil(function() {
            return this.unsuccessfulSubmissionHeader.getText() == 'Error: all fields are required'
        }, 3000)
        expect(this.unsuccessfulSubmissionHeader.getText()).to.include('Error: all fields are required');
    }

module.exports = new ContactUs_Page();
```

2. Create a new method as shown below:

```
    function clickSubmitButton() {
        return this.submitButton.click();
    }

    submitAllInformationViaContactUsForm(firstName, lastName, emailAddress, comments) {
        if(firstName) {
            this.firstName.setValue(firstName);
        }
        if(lastName) {
            this.lastName.setValue(lastName);
        }
        if(emailAddress) {
            this.emailAddress.setValue(emailAddress);
        }
        if(comments) {
            this.comments.setValue(comments);
        }
        this.submitButton.click();
        this.confirmSuccessfulSubmission();
    }

    function confirmSuccessfulSubmission() {
```

a. This method has a number of parameters for which we check to see if they have a value at the time when the method is called.
b. We then have IF statement that checks to see if a value is present (would return true) and the code within the IF statement would execute
c. So, for example, if we provide a firstName parameter value then the following statement:

```
submitAllInformationViaContactUsForm(firstName, lastName, emailAddress, comments) {
    if(firstName) {
        this.firstName.setValue(firstName);
    }
```

d. Would return TRUE and the firstName variable set at the top of the page would be assigned the value passed to this method parameter, firstName
e. We finish this method by calling the submitButton.click() and this.confirmSuccessfulSubmission which will always get exectured when this method is called (it's outside any IF condition)

3. What we have done here is combined all functions into one method call. This means we can now call the submitAllInformationViaContactUsForm method instead of calling each individual function.

We continue with phase three in the next lecture

Lecture 87 - Page Object Model (POM) - Phase 3 - Part 4

In this lecture, we make a further improvement to our ContactUs_Page.js by removing the individual *function* keywords to each of the former functions listed within the ContactUs_Page.js page. We then make changes to our contactUsTest.js file by referencing the ContactUs_Page variable in each of the "it" code blocks.

Key Points:

- We now have a method that uses IF statements to determine what values are being passed from a test and then uses the locators requested
- We now need to remove the function keywords from our ContactUs_Page.js file for the former individual functions
- We also need to amend our contactUsTest.js file to reference the ContactUs_Page variable

1. Open up your ContactUs_Page.js and contactUsTest.js files using Sublime Text
2. Your ContactUs_Page.js should now look like this:

ORIG_ContactUs_Page.js

```
class ContactUs_Page {
    get firstName() { return $("[name='first_name']");}
    get lastName() { return $("[name='last_name']");}
    get comments() { return $("textarea");}
    get emailAddress() { return $("[name='email']");}
    get submitButton() { return $("[type='submit']");}
    get successfulSubmissionHeader() { return $("#contact_reply h1");}
    get unsuccessfulSubmissionHeader() { return $("body");}
}

function setFirstName(firstName) {
        return this.firstName.setValue(firstName);
    }

    function setLastName(lastName) {
        return this.lastName.setValue(lastName);
    }

    function setEmailAddress(emailAddress) {
        return this.emailAddress.setValue(emailAddress);
    }

    function setComments(comments) {
        return this.comments.setValue(comments);
    }

    function clickSubmitButton() {
        return this.submitButton.click();
    }

    submitAllInformationViaContactUsForm(firstName, lastName, emailAddress, comments) {
        if(firstName) {
            this.firstName.setValue(firstName);
        }
        if(lastName) {
            this.lastName.setValue(lastName);
        }
        if(emailAddress) {
            this.emailAddress.setValue(emailAddress);
        }
        if(comments) {
            this.comments.setValue(comments);
        }
        this.submitButton.click();
        this.confirmSuccessfulSubmission();
    }

    function confirmSuccessfulSubmission() {
        var validateSubmissionHeader = browser.waitUntil(function() {
            return this.successfulSubmissionHeader.getText() == 'Thank You for your Message!'
        }, 3000)
        expect(validateSubmissionHeader, 'Successful Submission Message does not Exist!').to.be.true;
    }

    function confirmUnsuccessfulSubmission() {
        var validateSubmissionHeader = browser.waitUntil(function() {
            return this.unsuccessfulSubmissionHeader.getText() == 'Error: all fields are required'
        }, 3000)
        expect(this.unsuccessfulSubmissionHeader.getText()).to.include('Error: all fields are required');
    }

module.exports = new ContactUs_Page();
```

3. Make the following changes (removing the *function* keyword):

```
    setFirstName(firstName) {
        return this.firstName.setValue(firstName);
    }

    setLastName(lastName) {
        return this.lastName.setValue(lastName);
    }

    setEmailAddress(emailAddress) {
        return this.emailAddress.setValue(emailAddress);
    }

    setComments(comments) {
        return this.comments.setValue(comments);
    }

    clickSubmitButton() {
        return this.submitButton.click();
    }

    submitAllInformationViaContactUsForm(firstName, lastName, emailAddress, comments) {
        if(firstName) {
            this.firstName.setValue(firstName);
        }
        if(lastName) {
            this.lastName.setValue(lastName);
        }
        if(emailAddress) {
            this.emailAddress.setValue(emailAddress);
        }
        if(comments) {
            this.comments.setValue(comments);
        }
        this.submitButton.click();
        this.confirmSuccessfulSubmission();
    }

    confirmSuccessfulSubmission() {
        var validateSubmissionHeader = browser.waitUntil(function() {
            return this.successfulSubmissionHeader.getText() == 'Thank You for your Message!'
        }, 3000)
        expect(validateSubmissionHeader, 'Successful Submission Message does not Exist!').to.be.true;
    }

    confirmUnsuccessfulSubmission() {
        var validateSubmissionHeader = browser.waitUntil(function() {
            return this.unsuccessfulSubmissionHeader.getText() == 'Error: all fields are required'
        }, 3000)
        expect(this.unsuccessfulSubmissionHeader.getText()).to.include('Error: all fields are required');
    }
```

4. Now refer to your contactUsTest.js file and it should currently look like:

```
var ContactUs_Page = require("../pageObjects/ContactUs_Page.js");

beforeEach(function() {
    browser.url('/Contact-Us/contactus.html');
})

describe('Test Contact Us form WebdriverUni', function() {
    it('Should be able to submit a successful submission via contact us form', function(done) {
        setFirstName('joe');
        setLastName('Blogs');
        setEmailAddress('joe_blogs123@outlook.com');
        setComments('How are you?');
        clickSubmitButton();
        confirmSuccessfulSubmission();
    });


    it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
        setFirstName('Mike');
        setLastName('Woods');
        setEmailAddress('mike_woods@mail.com');
        clickSubmitButton();
        confirmUnsuccessfulSubmission();
    });


    it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
        setFirstName('Sarah');
        setEmailAddress('sarah_woods@mail.com');
        clickSubmitButton();
        confirmUnsuccessfulSubmission();
    });

    it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
        setLastName('Jomes');
        setEmailAddress('sarah_Jomes@mail.com');
        clickSubmitButton();
        confirmUnsuccessfulSubmission();
    });
});
```

5. We need to reference the ContactUs_Page variable for each line within the “it” blocks

6. Make the following changes to your file:

```
var ContactUs_Page = require("../pageObjects/ContactUs_Page.js");

beforeEach(function() {
  browser.url('/Contact-Us/contactus.html');
})

describe('Test Contact Us form WebdriverUni', function() {
  it('Should be able to submit a successful submission via contact us form', function(done) {
    ContactUs_Page.submitAllInformationViaContactUsForm('joe', 'Blogs', 'joe_blogs123@outlook.com', 'How are you?');
    });


  it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
    ContactUs_Page.setFirstName('Mike');
    ContactUs_Page.setLastName('Woods');
    ContactUs_Page.setEmailAddress('mike_woods@mail.com');
    ContactUs_Page.clickSubmitButton();
    ContactUs_Page.confirmUnsuccessfulSubmission();
    });


  it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
    ContactUs_Page.setFirstName('Sarah');
    ContactUs_Page.setEmailAddress('sarah_woods@mail.com');
    ContactUs_Page.clickSubmitButton();
    ContactUs_Page.confirmUnsuccessfulSubmission();
    });

  it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
    ContactUs_Page.setLastName('Jomes');
    ContactUs_Page.setEmailAddress('sarah_Jomes@mail.com');
    ContactUs_Page.clickSubmitButton();
    ContactUs_Page.confirmUnsuccessfulSubmission();
    });
});

```

   a. We have replaced the first “it” code block which originally looked like:

```
describe('Test Contact Us form WebdriverUni', function() {
    it('Should be able to submit a successful submission via contact us form', function(done) {
        setFirstName('joe');
        setLastName('Blogs');
        setEmailAddress('joe_blogs123@outlook.com');
        setComments('How are you?');
        clickSubmitButton();
        confirmSuccessfulSubmission();
    });
```

   b. We have called the submitAllInformationViaContactUsForm method from the ContactUs_Page.js file, meaning we no longer have to set the individual values to pass to the locators. We instead can pass the parameter values that we set:

```
describe('Test Contact Us form WebdriverUni', function() {
  it('Should be able to submit a successful submission via contact us form', function(done) {
    ContactUs_Page.submitAllInformationViaContactUsForm('joe', 'Blogs', 'joe_blogs123@outlook.com', 'How are you?');
    });
```

Note – if you do not want to set a parameter value for a particular test case, then you can simply use the *null* reference as shown:

E.g. in this example, I want to run a test that does not send a first name value. I could so this:

```
ContactUs_Page.submitAllInformationViaContactUsForm(null, 'Blogs', 'joe_blogs123@outlook.com', 'How are you?');
```

This will send a NULL value to the submitAllInformationViaContactUsForm method’s first parameter:

```
submitAllInformationViaContactUsForm(firstName, lastName, emailAddress, comments) {
    if(firstName) {
        this.firstName.setValue(firstName);
    }
    if(lastName) {
        this.lastName.setValue(lastName);
    }
    if(emailAddress) {
        this.emailAddress.setValue(emailAddress);
    }
    if(comments) {
        this.comments.setValue(comments);
    }
    this.submitButton.click();
    this.confirmSuccessfulSubmission();
}

```

We continue with phase three in the next lecture

Lecture 88 - Page Object Model (POM) - Phase 3 - Part 5

In this lecture, we review our code from phase 3 and look to see if we need to make any more changes before finalising our code.

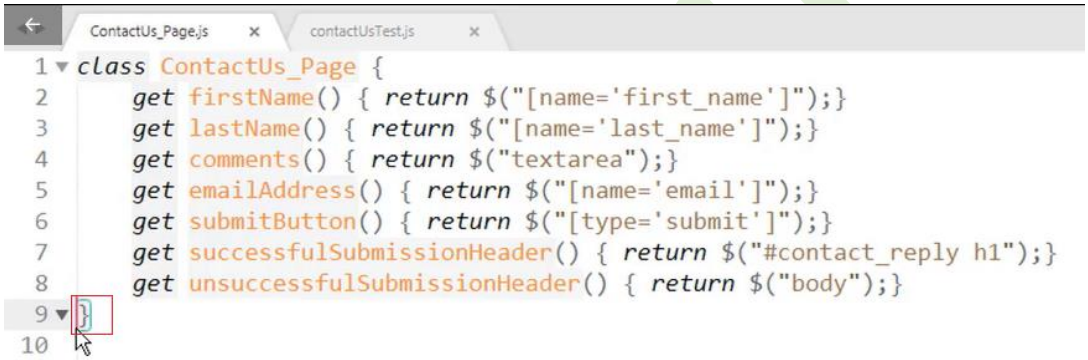
There are some changes that we identify**, so ensure to watch this lecture**.

Key Points:

- We now amended our contactUsTest.js file and ContactUs_Page file.
- We now need to review these files to ensure there aren't any errors or mistakes and to ensure our code is efficient

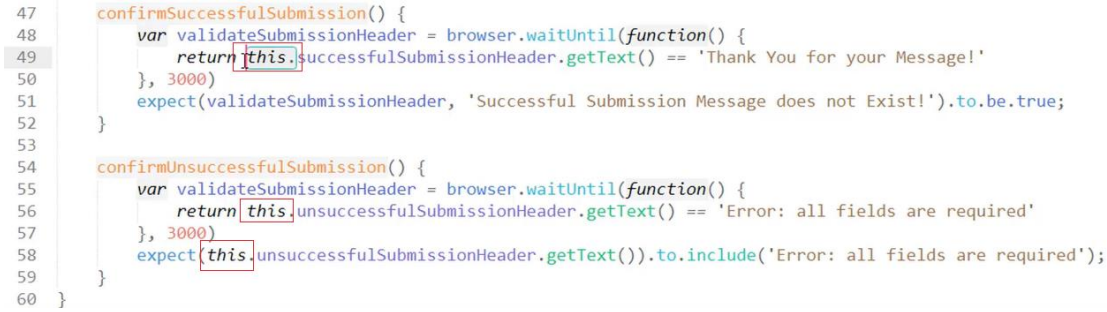
Instructions:

1. Open up your contactUsTest.js file
2. I notice that one of our closing brackets is in the wrong place:



3.
4. This closing bracket must be placed near the end of our file, since it refers to the opening bracket on line 1
5. Delete the bracket on line 9 and place a new bracket on line 60, as shown:

6. I also notice that we need to make a couple of changes to the confirmSuccessfulSubmission and confirmUnsuccessfulSubmission methods. This is because we have used the *this* keyword, but unfortunately it cannot be used as intended here:



7. Replace the sections highlighted above with:

8. Your two methods above should look like the following once all changes have been made:

```
47      confirmSuccessfulSubmission() {
48          var validateSubmissionHeader = browser.waitUntil(function() {
49              return browser.getText(successfulSubmissionHeader) == 'Thank You for your Message!'
50          }, 3000)
51          expect(validateSubmissionHeader, 'Successful Submission Message does not Exist!').to.be.true;
52      }
53
54      confirmUnsuccessfulSubmission() {
55          var validateSubmissionHeader = browser.waitUntil(function() {
56              return browser.getText(unsuccessfulSubmissionHeader) == 'Error: all fields are required'
57          }, 3000)
58          expect(browser.getText(unsuccessfulSubmissionHeader)).to.include('Error: all fields are required');
59      }
60  }
61
62  module.exports = new ContactUs_Page();
```

9. Now, we can make one more improvement to our code
10. Notice on lines 7 and 8 we have the following:

```
7       get successfulSubmissionHeader() { return $("#contact_reply h1");}
8       get unsuccessfulSubmissionHeader() { return $("body");}
```

11. We are using get methods to return our selectors at the top of this file. We could instead place these within the confirmSuccessfulSubmission and confirmUnsuccessfulSubmission methods
12. Create two new variables as highlighted below:

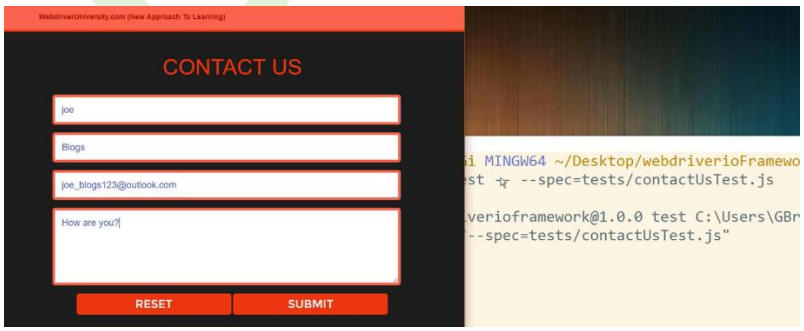
```
45      confirmSuccessfulSubmission() {
46          var successfulSubmissionHeader = "#contact_reply h1";
47          var validateSubmissionHeader = browser.waitUntil(function() {
48              return browser.getText(successfulSubmissionHeader) == 'Thank You for your Message!'
49          }, 3000)
50          expect(validateSubmissionHeader, 'Successful Submission Message does not Exist!').to.be.true;
51      }
52
53      confirmUnsuccessfulSubmission() {
54          var unsuccessfulSubmissionHeader = "body";
55          var validateSubmissionHeader = browser.waitUntil(function() {
56              return browser.getText(unsuccessfulSubmissionHeader) == 'Error: all fields are required'
57          }, 3000)
58          expect(browser.getText(unsuccessfulSubmissionHeader)).to.include('Error: all fields are required');
59      }
60  }
61
```

13. This means we now can remove the two lines on line number 7 and 8:

```
7       get successfulSubmissionHeader() { return $("#contact_reply h1");}
8       get unsuccessfulSubmissionHeader() { return $("body");}
```

Remove

14. We have now made all the necessary changes to our file. Save it and close Sublime Text
15. We also reviewed the contactUsTest.js file during the video and no changes were required
16. Open up GitBash / iTerm2 and navigate to our webdriverioFramework directory
17. Run the following command:
18. **npm test -- -- specs=tests/contactUsTest.js** + press enter
19. You should see the tests run, as shown below:



20. If we then review the console output, you should see:

```
····

4 passing (14.60s)
```

21. Four tests have passed successfully! We've made a number of improvements to our code

Lecture 89 - Page Object Model (POM) - Phase 3 - Part 6

In this lecture, we go over the changes we have made during Phase 3. We recap on the key improvements that we have made to our code.

Key Points:

- We have made a number of code improvements to our files
- They are now more maintainable and follow good coding practises
- We have also reduced the amount of code needed, considerably
- This lecture goes over some of the key changes we have made

As we have already documented our changes in lectures 82-86, please refer back to the documentation from the past lectures above for a recap.

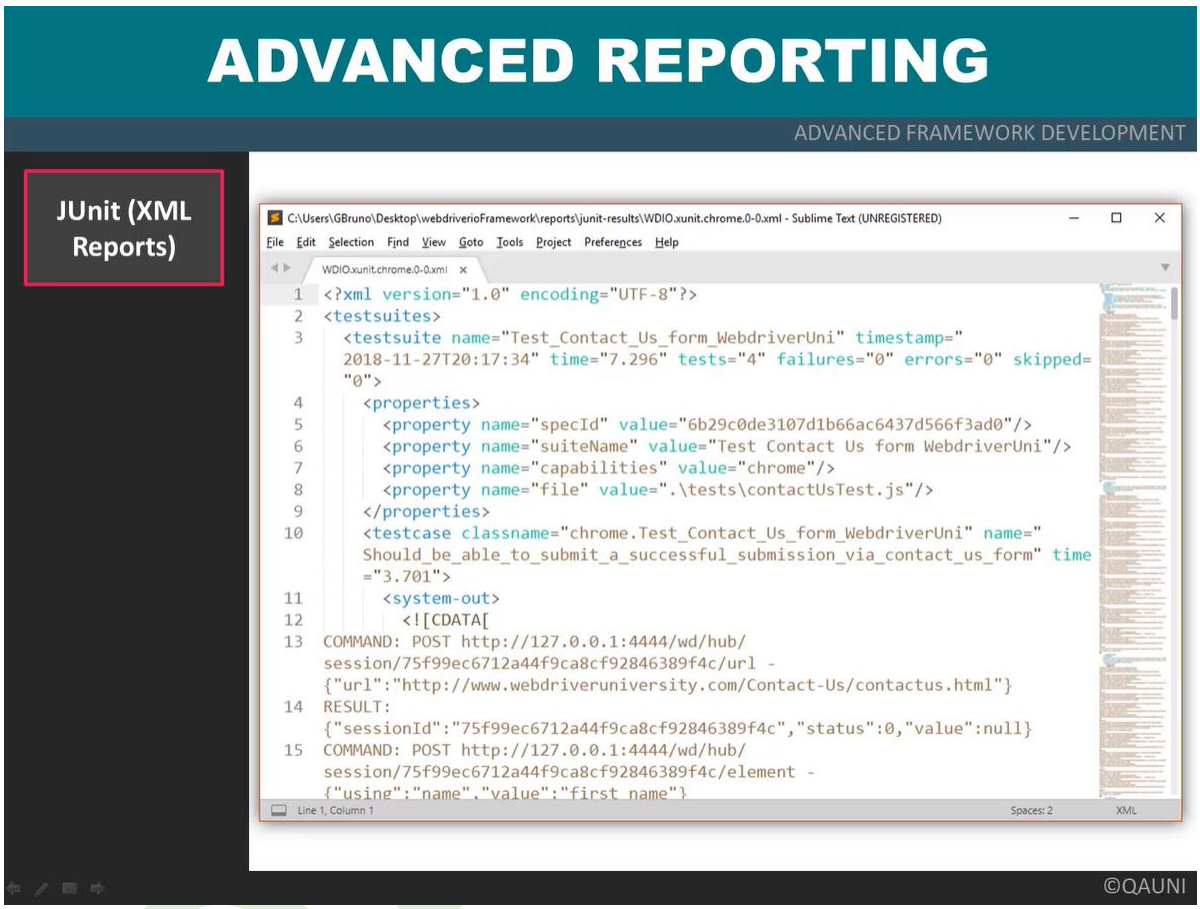
I would also advise reviewing this lecture video for a quick recap of the changes made.

Module 22 – Advanced Reporting

In this module, we take our first look at advanced reporting. I demonstrate Junit reports, JSON reports and Allure reports in this section.

Lecture 90 - Advanced Reporting – Intro

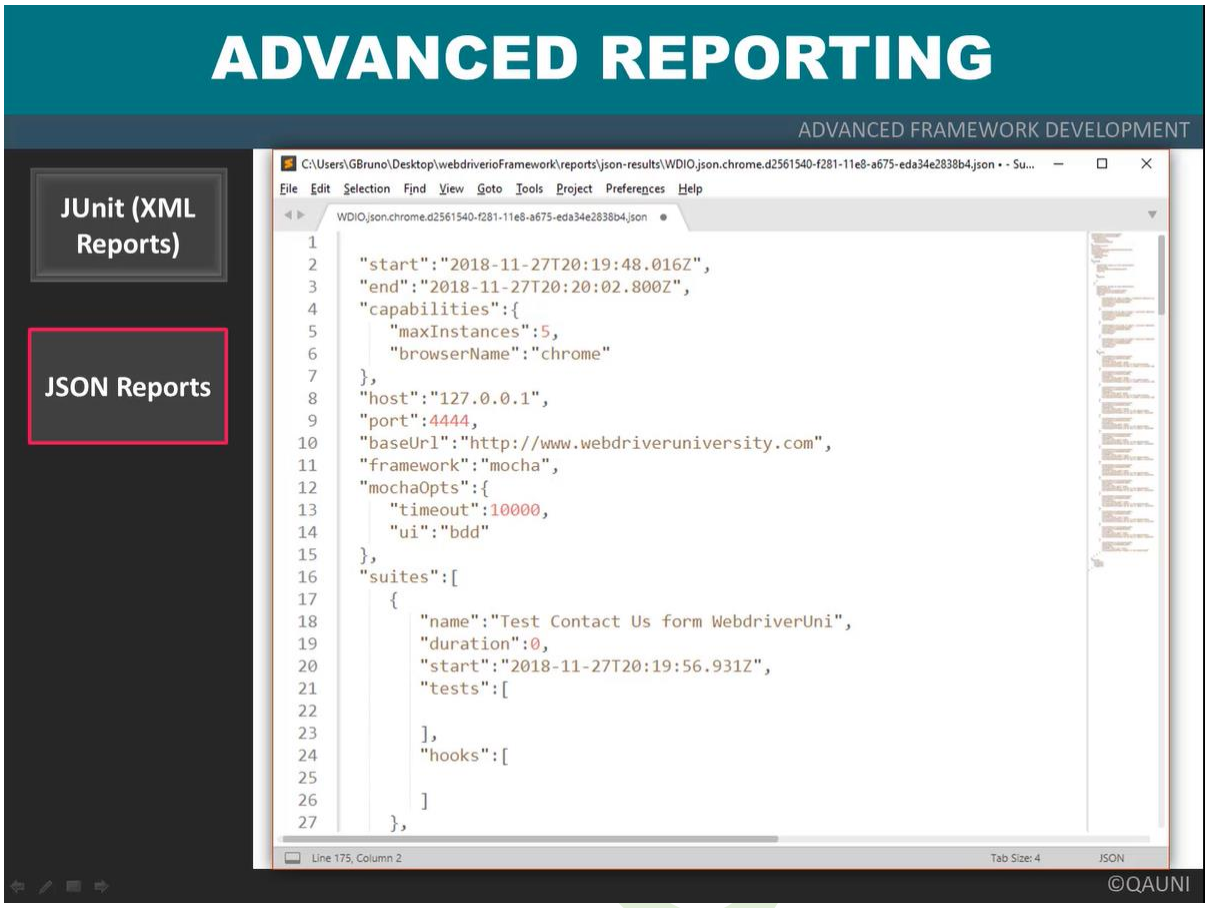
JUnit Reports



- JUnit reports are in the format of XML.
- Many integration systems such as Jenkins (which we cover in the next module) will enable us to execute our tests whenever we want
- Once executed, you'll have the ability to generate reports like that shown above
- In turn, CI (continuous integration) systems such as Jenkins will be able to the XML data above to produce its own reports

Notes continued on next page

JSON Reports



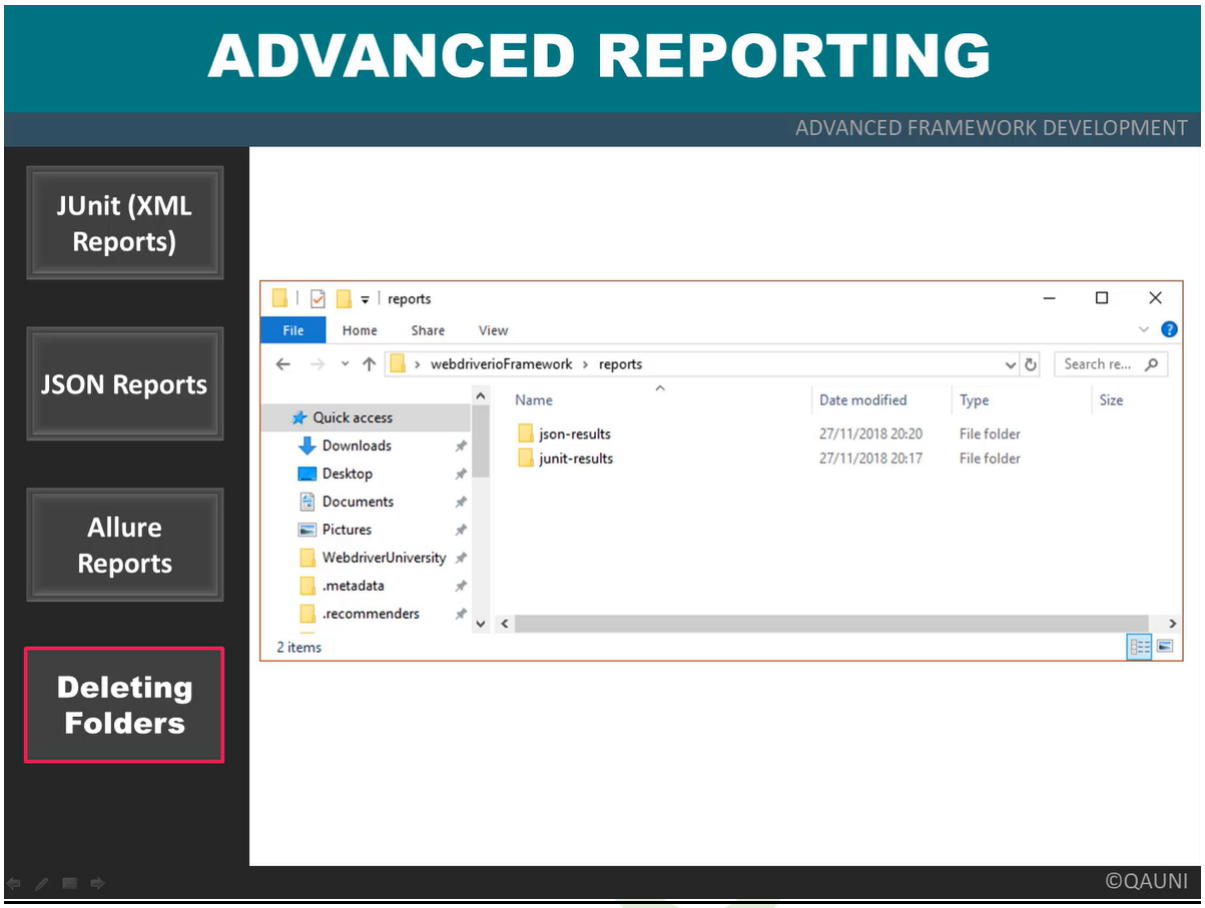
- JSON is very similar to XML, however the data is structured in a different way
- JSON: JavaScript Object Notation
- JSON is a syntax for storing and exchanging data
- JSON is text, written with JavaScript object notation
- Again, other CI systems such as Jenkins can use this data to generate reports

Notes continued on next page

Allure Reports

- Allure Framework is a flexible lightweight multi-language test report tool with the possibility to add screenshots, logs and so on.
- It provides modular architecture and neat web reports with the ability to store attachments, steps, parameters and many more.
- Allure reports are visually appealing and are very easy to read since the data is structured and well organised.

Deleting Folders



- Finally, we will look at deleting folders
- Our reports will generate a number of folders which will contain report outputs
- We need to be clear on how we can manage these folders/files going forward
- For example, we run a test and a report output is generated but we may want to maintain and clear folders from our past test before instructing a new test

Lecture 91 - Advanced Reporting - Junit Reports

In this lecture we start looking at Junit reports.

Key points:

- JUnit reports are in the format of XML.
- Many integration systems such as Jenkins (which we cover in the next module) will enable us to execute our tests whenever we want
- Once executed, you'll have the ability to generate reports like that shown above
- In turn, CI (continuous integration) systems such as Jenkins will be able to the XML data above to produce its own reports

Instructions:

1. Go to your webdriverFramework directory using Windows Explorer or Finder on Mac
2. We need to edit our WDIO.conf file, so open this in Sublime Text
3. Scroll down to around lines 135 and you will see the following:

```
133     // Test reporter for stdout.
134     // The only one supported by default is 'dot'
135     // see also: http://webdriver.io/guide/reporters/dot.html
136     // reporters: ['dot'],
137     //
```

4. We need to uncomment some of this section, to look like:

```
136     reporters: ['dot', 'junit'],
137
138     reporterOptions: {
139         junit: {
140             outputDir: './reports/junit-results/'
141         }
142     },
143     //
```

   a. Online 136 we are setting the use of two report types, 'dot' which is the default reporting type and 'junit' which is the report type will be using for this lecture
   b. We then set reporting options and instruct our Junit output to save in the directory listed

5. Once you have made the changes above, save and then close the file
6. Open up GitBash/ iTerm2 and move into the webdriverFramework directory
7. Run the following command:

```
GBruno@Gi MINGW64 ~/Desktop/webdriverioFramework
$  npm install wdio-junit-reporter --save-dev
```

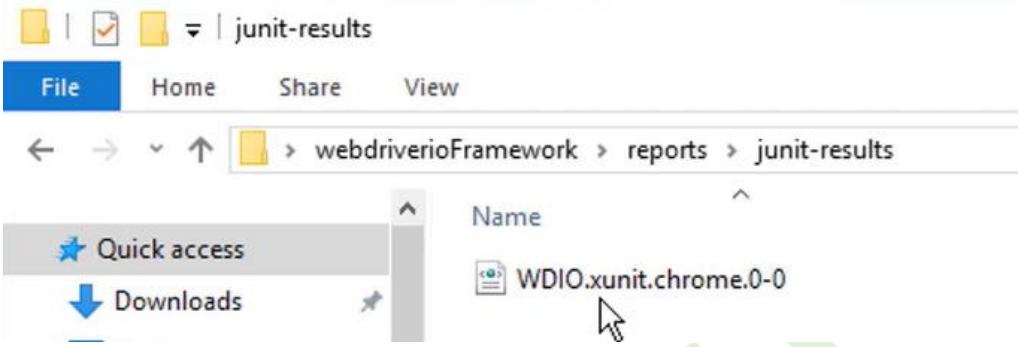
8. Here we are installing the wdio-junit-report package
9. You should see the following once the installation is completed:

```
MINGW64:/c/Users/GBruno/Desktop/webdriverioFramework
$  npm install wdio-junit-reporter --save-dev
npm WARN webdriverioframework@1.0.0 No repository field.

+ wdio-junit-reporter@0.4.4
updated 1 package and audited 760 packages in 4.169s
found 0 vulnerabilities

GBruno@Gi MINGW64 ~/Desktop/webdriverioFramework
$
```

10. Next, run the following command:

```
> wdio "--spec=tests/contactUsTest.js"
```

11. Here we are using our past contactUsTest.js file to generate the report
12. Once the test has completed, look at the following directory location and you will see some folders have been created to store our report output:



13. Open the file above using Sublime Text and you should see:

14. This is information of the test we have just executed.
15. In a later module, we will use this data with Jenkins which is a CI system, using the JUnit Plugin to output the data above in a more readable form

Lecture 92 - Advanced Reporting - JSON Reports

In this lecture we start looking at JSON reports.

Key points:

- JSON is very similar to XML, however the data is structured in a different way
- JSON: JavaScript Object Notation
- JSON is a syntax for storing and exchanging data
- JSON is text, written with JavaScript object notation
- Again, other CI systems such as Jenkins can use this data to generate reports

Instructions:

1. Go to your webdriverFramework directory using Windows Explorer or Finder on Mac
2. We need to edit our WDIO.conf file, so open this in Sublime Text
3. Scroll down to around lines 135 and you will see the following (which includes changes from lecture 89):

```
136    reporters: ['dot', 'junit'],
137
138    reporterOptions: {
139        junit: {
140            outputDir: './reports/junit-results/'
141        }
142    },
143    //
```

4. Change this section to now include JSON reports, as shown:

```
136    reporters: ['dot', 'junit', 'json'],
137
138    reporterOptions: {
139        junit: {
140            outputDir: './reports/junit-results/'
141        },
142        json: {
143            outputDir: './reports/json-results/'
144        },
145    },
```

5. Save your file then open up GitBash/ iTerm2 and move into the webdriverFramework directory
6. Run the following command:

```
GBruno@Gi MINGW64 ~/Desktop/webdriverioFramework
$ npm install wdio-json-reporter --save-dev
```

7. Here we are installing the wdio-json-reporter package
8. You should see the following once the installation is completed:

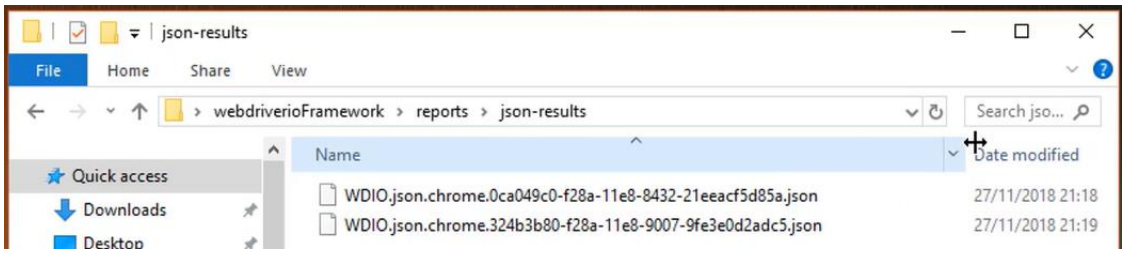
```
MINGW64:/c/Users/GBruno/Desktop/webdriverioFramework
+ wdio-json-reporter@0.4.0
updated 1 package and audited 760 packages in 4.569s
found 0 vulnerabilities
```

9.
10. Next, run the following command:

```
> wdio "--spec=tests/contactUsTest.js"
```

11. Here we are using our past contactUsTest.js file to generate the report

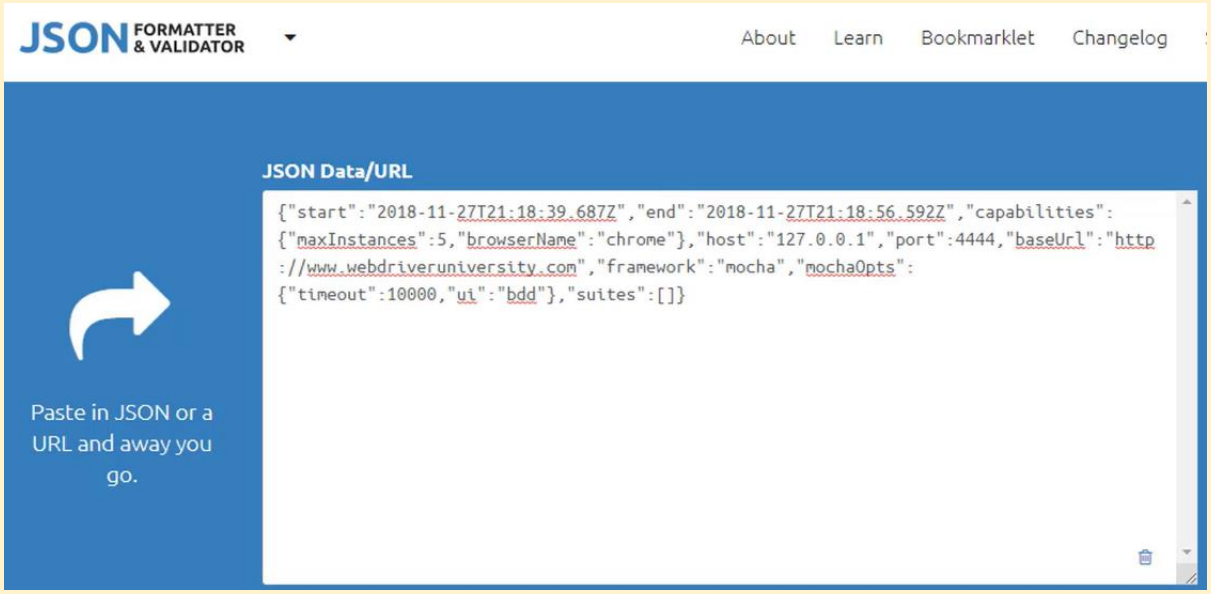
12. Once the test has completed, look at the following directory location and you will see some folders have been created to store our report output:



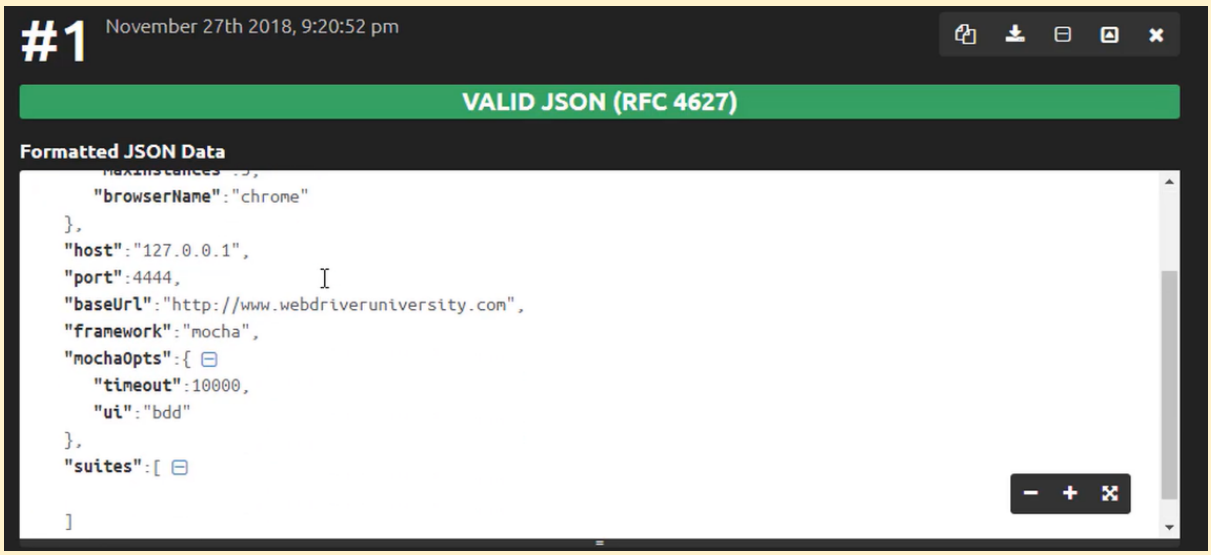
13. Open one of the files above in Sublime Text

TIP!

There are a number of websites that allow you to reformat the JSON data into a more readable format. An example is https://jsonformatter.curiousconcept.com/ where you can copy your JSON data into the following text box:



It will transform your data to:



Lecture 93 - Advanced Reporting - Allure Reports

In this lecture we start looking at Allure reports.

Key Points:

- Allure Framework is a flexible lightweight multi-language test report tool with the possibility to add screenshots, logs and so on.
- It provides modular architecture and neat web reports with the ability to store attachments, steps, parameters and many more.
- Allure reports are visually appealing and are very easy to read since the data is structured and well organised.

Instructions:

1. Go to your webdriverFramework directory using Windows Explorer or Finder on Mac
2. We need to edit our WDIO.conf file, so open this in Sublime Text
3. Scroll down to around lines 135 and you will see the following (which includes changes from lecture 90):

```
136     reporters: ['dot', 'junit', 'json'],
137
138     reporterOptions: {
139         junit: {
140             outputDir: './reports/junit-results/'
141         },
142         json: {
143             outputDir: './reports/json-results/'
144         },
145     },
```

4. Change this section to now include allure reports, as shown:

```
136     reporters: ['dot', 'junit', 'json', 'allure'],
137
138     reporterOptions: {
139         junit: {
140             outputDir: './reports/junit-results/'
141         },
142         json: {
143             outputDir: './reports/json-results/'
144         },
145         allure: {
146             outputDir: './reports/allure-results/',
147             disableWebdriverStepsReporting: true,
148             disableWebdriverScreenshotsReporting: false,
149             useCucumberStepReporter: false
150         }
151     },
```

   a. Notice we add a few additional lines from lines 147. These are report options and you can read more about these options from the official allure report documentation, found here: https://docs.qameta.io/allure/

5. Save your file then open up GitBash/ iTerm2 and move into the webdriverFramework directory
6. Run the following command:

```
GBruno@Gi MINGW64 ~/Desktop/webdriverioFramework
$ npm install wdio-allure-reporter --save-dev
```

7. Here we are installing the wdio-allure-reporter package
8. You should see the following once the installation is completed:

```
npm WARN webdrivierioframework@1.0.0 No repository field.

+ wdio-allure-reporter@0.8.3
added 10 packages from 15 contributors and audited 776 packa
ges in 4.773s
found 0 vulnerabilities
```

9. We then have to also install the following which is a allure command line package that will give us more flexibility and the ability to type additional report commands:

```
GBruno@Gi MINGW64 ~/Desktop/webdriverioFramework
$ npm install -g allure-commandline --save-dev
```

10. You should see the following once the installation is completed:

```
GBruno@Gi MINGW64 ~/Desktop/webdriverioFramework
$ npm install -g allure-commandline --save-dev
C:\Users\GBruno\AppData\Roaming\npm\allure -> C:\Users\GBrun
o\AppData\Roaming\npm\node_modules\allure-commandline\bin\al
lure
+ allure-commandline@2.8.1
added 1 package in 10.008s
```

11. Now we need to see if we can successfully generate an allure report using one of our past test files. This time we use our past 'ajaxClickTest.js' file
12. Next, run the following command:

```
$ npm test -- --spec=ajaxClickTest.js
```

13. You should see the following output:

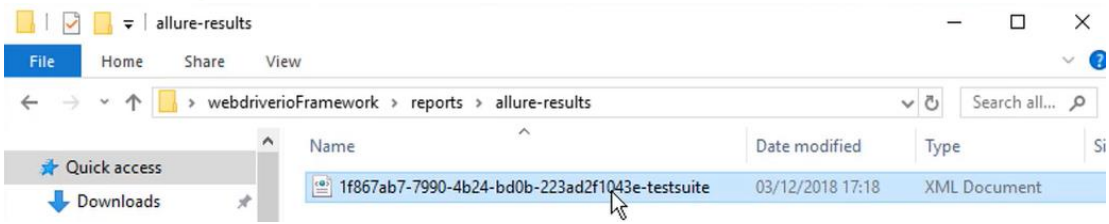
```
1 passing (25.00s)
1 skipped
```

    a. The reason one test was skipped was due to our ajaxClickTest.js test having a 'it.skip' commands set, as shown:

```
2     it.skip('Attempt to click the button asap', function(done) {
```

    b. This isn't a problem however, as we can still generate our report from the test that passed

14. If you navigate to the following directory, you should see that an allure report file has been created:



15. We now need to use this file to generate a report
16. In GitBash/ iTerm2, run the following command:

```
$ allure generate C:/Users/GBruno/Desktop/webdriverioFramework/rep
orts/allure-results
```

17. You should then see the following output message:

```
GBruno@Gi MINGW64 ~/Desktop/webdriverioFramework
$ allure generate C:/Users/GBruno/Desktop/webdriverioFramework/rep
orts/allure-results
Report successfully generated to allure-report

GBruno@Gi MINGW64 ~/Desktop/webdriverioFramework
$
```

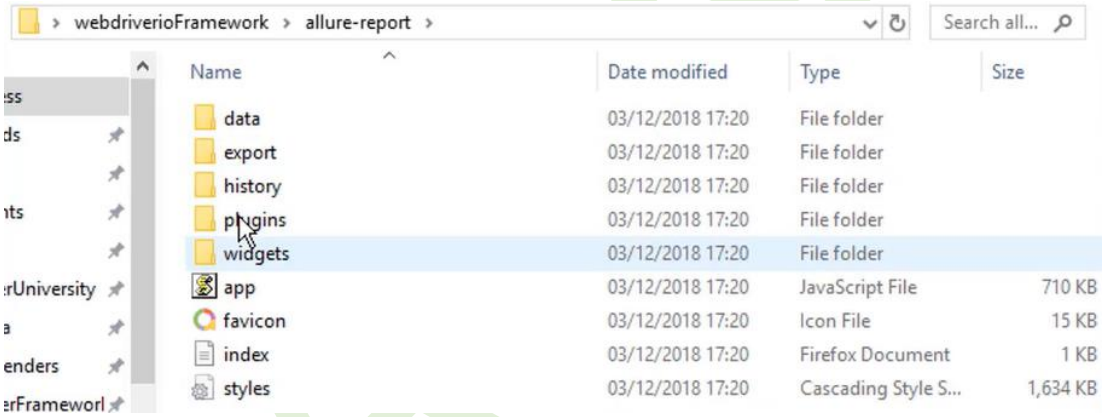
18. This has outputted a successful generated report message

TIP!

If you ever see a message similar to "report already in use" type the following command into GitBash/ iTerm2:

```
GBruno@Gi MINGW64 ~/Desktop/webdriverioFramework
$ allure generate C:/Users/GBruno/Desktop/webdriverioFramework/rep
orts/allure-results --clean
```

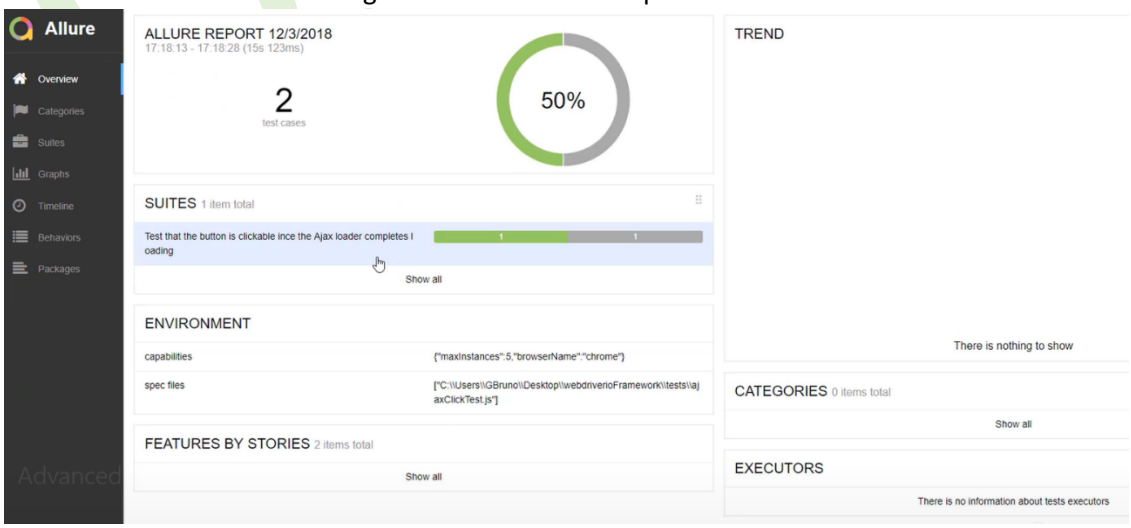
19. Now if you go to the following directory, you should see the allure report files that have been created:



20. Now, all we need to do it write one more command to view our report. Type the following command:

```
$ allure open
```

21. You should then see something similar to the below open:



22. Congratulations, you've just generated a successful Allure report!

Lecture 94 - Advanced Reporting - Allure Reports - Attaching Images - Part 1

In this lecture we start looking at adding exception images to our reports.

Key Points:

- Images are a great way to identify when a problem occurs or to record evidence of a test completing successfully
- This lecture demonstrates how we can record images (screenshots) of our test for which we can use as additional evidence

Instructions:

1. Go to your webdriverFramework directory using Windows Explorer or Finder on Mac
2. We need to edit our WDIO.conf file, so open this in Sublime Text
3. We need to amend a section (hooks) to instruct our tests to record an image when it comes across a failure

```
248       * Gets executed after all tests are done. You still have access to all global variables from
249       * the test.
250       * @param {Number} result 0 - test pass, 1 - test fail
251       * @param {Array.<Object>} capabilities list of capabilities details
252       * @param {Array.<String>} specs List of spec file paths that ran
253       */
254      // after: function (result, capabilities, specs) {
255      // },
256      /**
```

4. This hook executes after all tests are done
5. Amend your code to look like the following:

```
253       */
254       after: function (result, capabilities, specs) {
255          var name = 'ERROR-chrome-' + Date.now();
256          browser.saveScreenshot('./errorShots/' + name + '.png');
257       },
258      /**
```

   a. Here we have entered a new after: function that takes the result, capabilities and specs as arguments
   b. We then create a new variable and give it a value of 'ERROR-chrome-' and provide the date function to record to date of the failure
   c. We then use the saveScreenshot function and provide the directory path plus the name variable then .png file type
      i. This will save the screenshot to the directory listed and give the file a *name* based on the chrome error it comes across and then saves the file as a .png filetype
6. Make the changes above and then save your file
7. Use windows explorer/ iTerm2 and navigate to your "tests" folder
8. We are again going to use the ajaxClickTest.js file, so open this in Sublime Text
9. We are going to purposely make one of the "it" tests fail
10. Your file at this moment in time should look like:

```
wdio.conf.js   ajaxClickTest.js
1  describe('Test that the button is clickable ince the Ajax loader completes loading', function()
2      it.skip('Attempt to click the button asap', function(done) {
3          browser.url('/Ajax-Loader/index.html');
4          browser.click('#button1');
5      });
6
7      it('Attempt to click the button after 7 seconds', function(done) {
8          browser.url('/Ajax-Loader/index.html');
9          this.timeout(20000);
10         browser.pause(7000);
11         browser.click('#button1');
12         browser.pause(7000);
13     });
14 });
```

11. Change the 'it.skip' on line 2 and change it to 'it.'
12. Also change the selector on line 4 to an invalid value, like shown:

```
wdio.conf.js   ajaxClickTest.js
1  describe('Test that the button is clickable ince the Ajax loader completes loading', function()
2      it('Attempt to click the button asap', function(done) {
3          browser.url('/Ajax-Loader/index.html');
4          browser.click('#button1555');
5      });
6
7      it('Attempt to click the button after 7 seconds', function(done) {
8          browser.url('/Ajax-Loader/index.html');
9          this.timeout(20000);
10         browser.pause(7000);
11         browser.click('#button1');
12         browser.pause(7000);
13     });
14 });
```

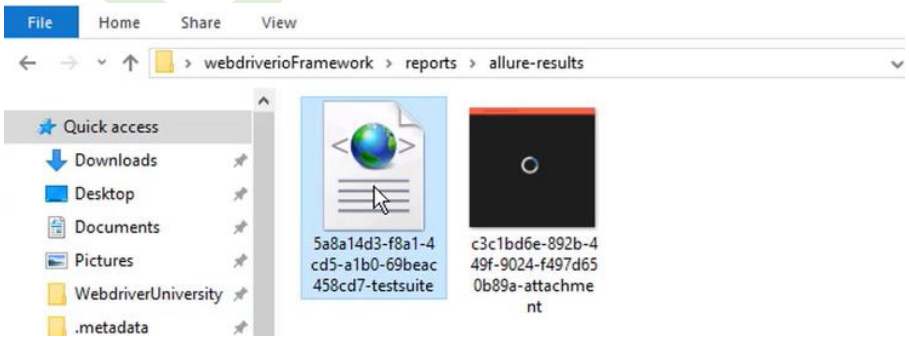
13. Once you have made the changes above, save your file and close Sublime Text
14. Now we need to trigger our test
15. Open up GitBash/ iTerm2 and move into the webdriverFramework directory
16. Run the following command:

```
$ npm test -- --spec=tests/ajaxClickTest.js
```

17. You should see our test fail (as expected):

```
Wrote xunit report "WDIO.xunit.chrome.0-0.xml" to
/junit-results/].
Wrote json report to [./reports/json-results/].
npm ERR! Test failed.  See above for more details.

GBruno@Gi MINGW64 ~/Desktop/webdriverioFramework
$
```

18. Now, if you go to the following directory you should see an Allure result:



19. We can then type the following into the command line:

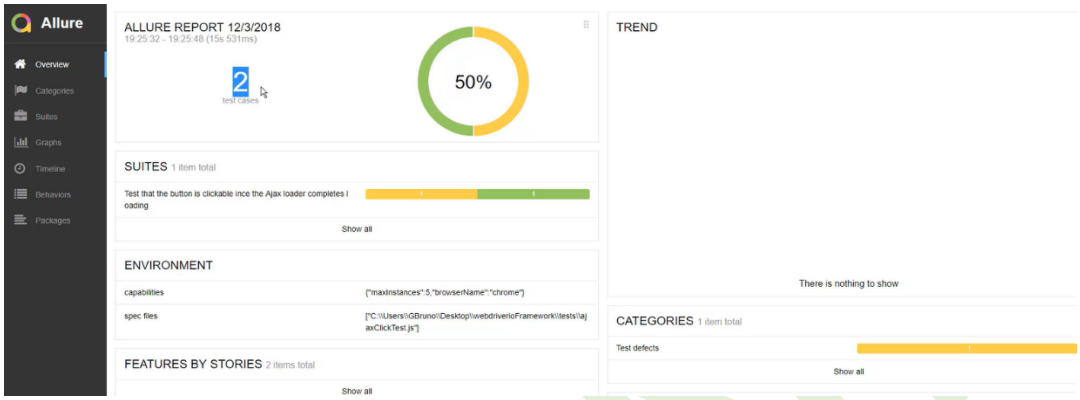
```
$ allure generate C:/Users/GBruno/Desktop/webdriverioFramework/reports/allure-results
```

    a. We are passing the directory of the allure results folder

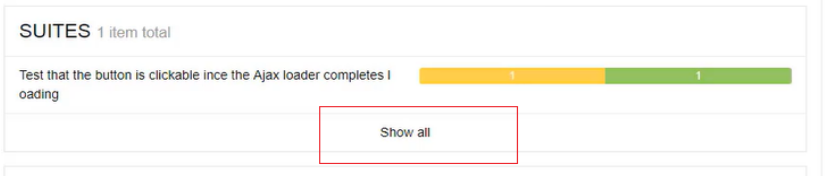
20. You should see an output message to say the report generation was successful:

```
GBruno@Gi MINGW64 ~/Desktop/webdriverioFramework
$ allure generate C:/Users/GBruno/Desktop/webdriverioFramew
ork/reports/allure-results
Report successfully generated to allure-report
```

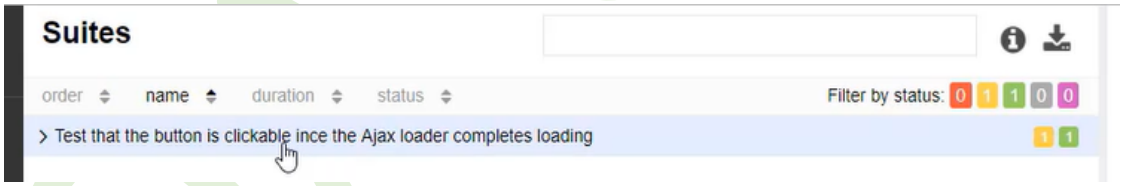
21. You should then see the Allure Report output:



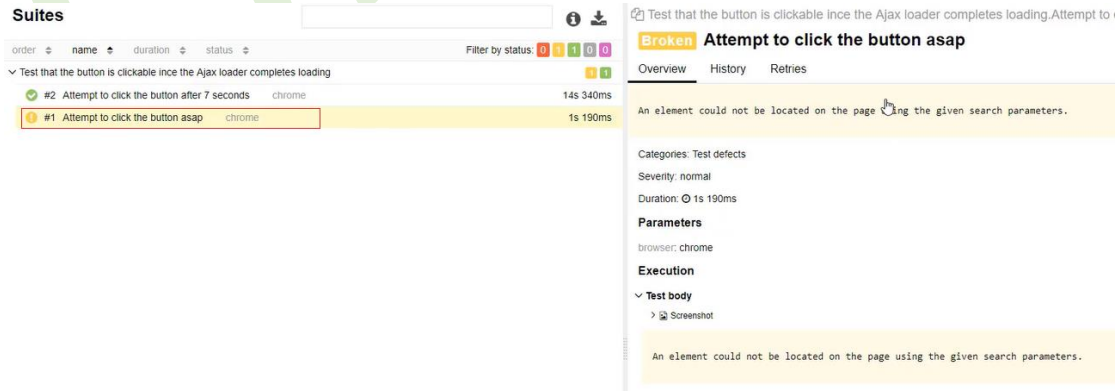
22. Click the Show All button (shown below):



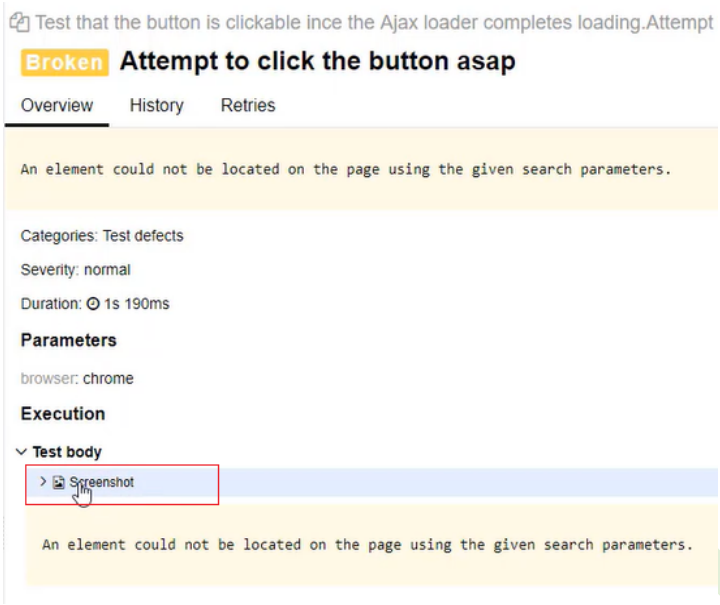
23. Then, on the next screen, click the following:



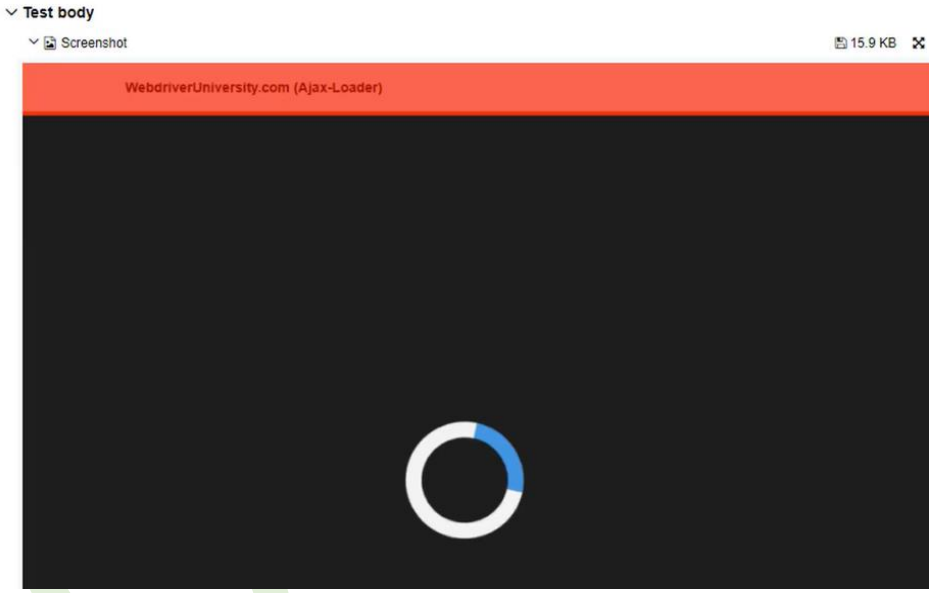
24. Click on the broken test (highlighted by a yellow background):



25. You'll notice there is a screenshot attached, as shown:



26. Click on the screenshot attachment and the image will open, showing the stage of where the test failed:



Lecture 95 - Advanced Reporting - Allure Reports - Attaching Images - Part 2

I decided to create this lecture just to explain the WDIO report configuration and to go through some additional options that we have available.

Key Points:

- We now know how to attach images to our reports from our tests
- We have already configured some options in our WDIO file associated to images
- We now go through the additional configuration and options that we have available to us

We currently have the following configuration in our WDIO file:

```
138    reporterOptions: {
139        junit: {
140            outputDir: './reports/junit-results/'
141        },
142        json: {
143            outputDir: './reports/json-results/'
144        },
145        allure: {
146            outputDir: './reports/allure-results/',
147            disableWebdriverStepsReporting: true,
148            disableWebdriverScreenshotsReporting: false,
149            useCucumberStepReporter: false
150        }
151    },
```

- disableWebdriverStepsReporting: this is an optional parameter(false by default), in order to log only custom steps to the reporter.
- disableWebdriverScreenshotsReporting: this is an optional parameter(false by default), in order to not attach screenshots to the reporter.

Please refer to the lecture video to see demonstrations of the difference and output of when we use different combinations of the options stated above

Lecture 96 - Free Up Space - Deleting Files and Folders

In this lecture we will be making further modifications to the wdio file, to delete specific files and folders prior to test execution, in turn freeing up disk (Hard drive) space.

Key Points:

- We need to use the “beforeSession” hook which is housed within the wdio file, to delete specific files and folders contained within our projects base directory (webdriverioFramework) prior to test suite execution.
- We will then install the following package: https://www.npmjs.com/package/del
- Once the package has been installed it will enable us to delete specific files and folders via the wdio hook; in our case we will be deleting all reports and images (Images, files and folders) generated from the previous test suite execution ('allure-report', 'errorShots', 'reports’)

Instructions:

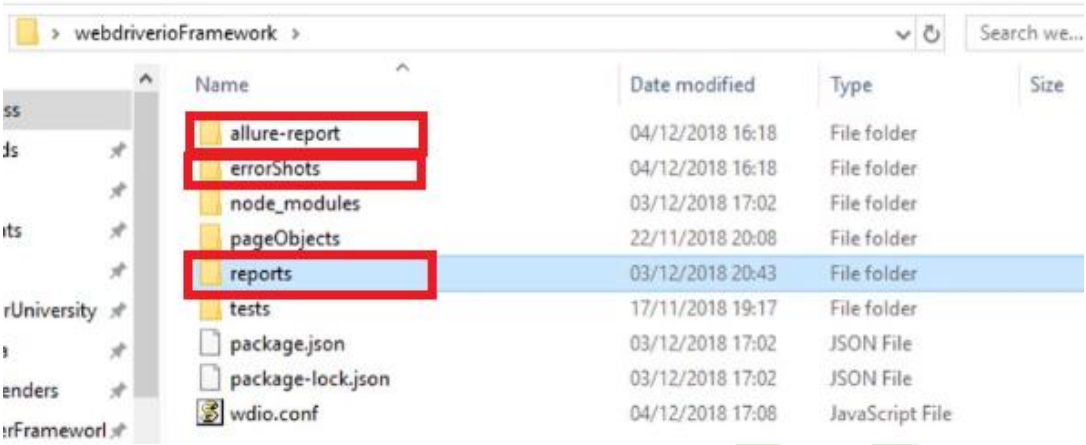
1. Open the wdio file and locate the “beforeSession” hook, currently the file should look like the following:

```
wdio.conf.js
166     // resolved to continue.
167     /**
168      * Gets executed once before all workers get launched.
169      * @param {Object} config wdio configuration object
170      * @param {Array.<Object>} capabilities list of capabilities details
171      */
172     // onPrepare: function (config, capabilities) {
173     // },
174     /**
175      * Gets executed just before initialising the webdriver session and test framework. It allows you
176      * to manipulate configurations depending on the capability or spec.
177      * @param {Object} config wdio configuration object
178      * @param {Array.<Object>} capabilities list of capabilities details
179      * @param {Array.<String>} specs List of spec file paths that are to be run
180      */
181     // beforeSession: function (config, capabilities, specs) {
182     // },
183     /**
184      * Gets executed before test execution begins. At this point you can access to all global
185      * variables like `browser`. It is the perfect place to define custom commands.
186      * @param {Array.<Object>} capabilities list of capabilities details
187      * @param {Array.<String>} specs List of spec file paths that are to be run
188      */
```

2. Now uncomment the “beforeSession” hook code and add the following code to the hook as listed below:

```
 * Gets executed just before initialising the webdriver session and test framework.
 * to manipulate configurations depending on the capability or spec.
 * @param {Object} config wdio configuration object
 * @param {Array.<Object>} capabilities list of capabilities details
 * @param {Array.<String>} specs List of spec file paths that are to be run
 */
beforeSession: function (config, capabilities, specs) {
    const del = require('del');
    del(['allure-report', 'errorShots', 'reports']);
},
/**
```

3. As you can see from the code listed within the previous image, the following command is used: “del” (*Please note we will install the required ‘del command’ package in the next steps*) which will delete the following folders, prior to the execution of the test suite.

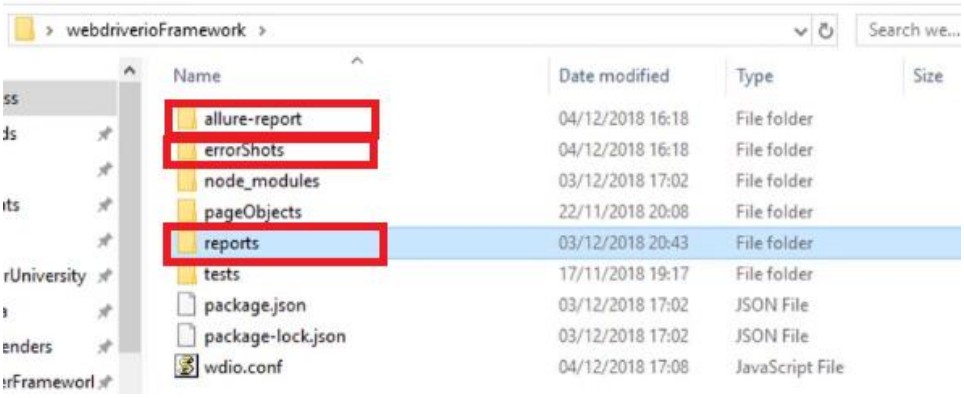


4. Now open up Git Bash and navigate to the projects rooter directory (webdriverIoFramework) and run the command below which will install the “del” packages.

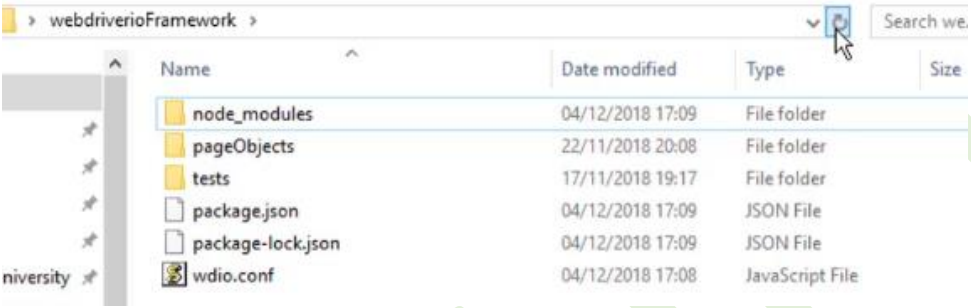
5. Now simply target the “ajaxClickTest.js” test within the same Git Bash window as listed:

6. Now if we inspect the projects root folder (webdriverioFramework) you will see that the targeted folders now get deleted prior to the execution of the test suite.

7. ***Before***:



8. ***After*** (Above folders are now deleted before the test(s) begin executing):



9. Once the test(s) have completed the projects root folder (webdriverioFramework) will get updated with the latest images, files and folders (allure-report, errorShots, reports) related to latest test suite execution.

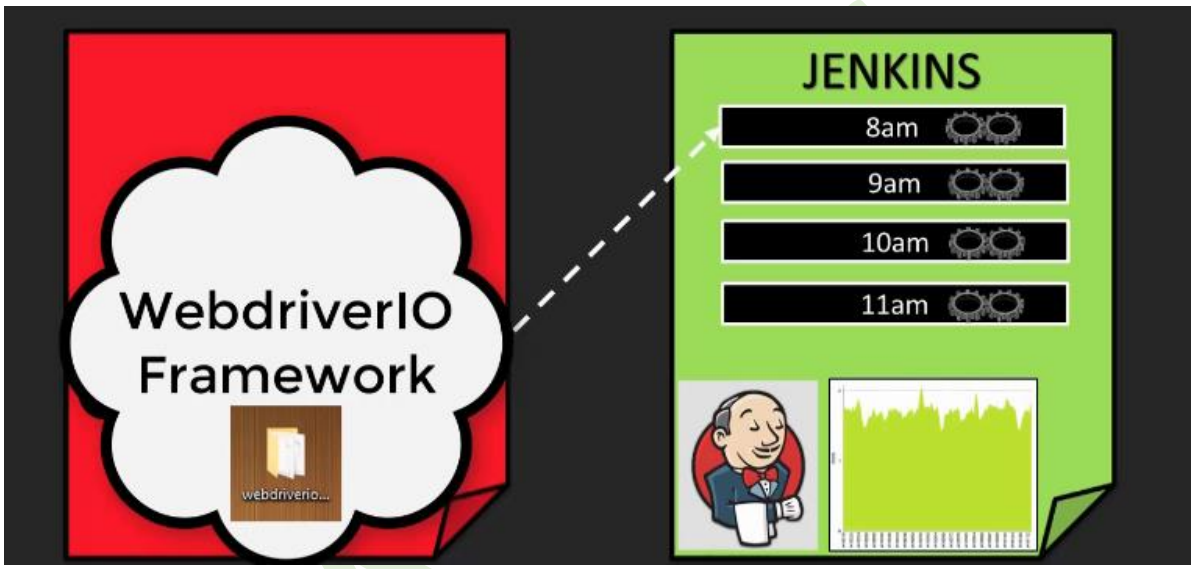
Lecture 97 - Jenkins Introduction and Explanation

In this lecture we will be taking a look at Jenkins (Open Source) continuous integration system (CI).

Key Points:

- Jenkins is the leading open source automation server; Jenkins provides hundreds of plugins to support building, deploying and automating any project.
- Jenkins can be used to trigger our automation framework / tests whenever we want.
- We will look at many plugins which can be used to enhance our Jenkins instance.

As you can see from the image example below, Jenkins even has the functionality to trigger our test suite automatically during specific times of the day (e.g. 8am, 9am, 10am…).



Lecture 98 - Jenkins Installation and Setup

It's fairly simple to setup Jenkins by using a .war file (*WAR file (Web Application Resource or Web application Archive) is a file used to distribute a collection of JAR-files*) making the deployment of Jenkins simple!

Key Points:

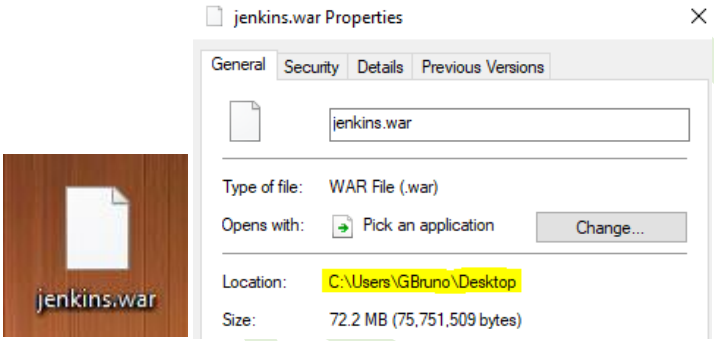
- We will download Jenkins (.war file) via the following url: https://jenkins.io/download/
- We will access Jenkins via a localhost, create admin credentials and take a first glimpse of the Jenkins UI.

Instructions:

1. Access the following url: https://jenkins.io/download/
2. Click on the following option to download the Jenkins .war file:

   Generic Java package (.war)

3. Let the file download, drag and drop the file to your desired location; in my case I used the Desktop:



4. Now open Git Bash and run the following command (Please note I 'am specifying port 5555 because port 8080 is currently in use on my machine):

```
MINGW64:/c/Users/GBruno/Desktop

GBruno@Gi MINGW64 ~
$ cd Desktop

GBruno@Gi MINGW64 ~/Desktop
$ java -jar jenkins.war --httpPort=5555
```

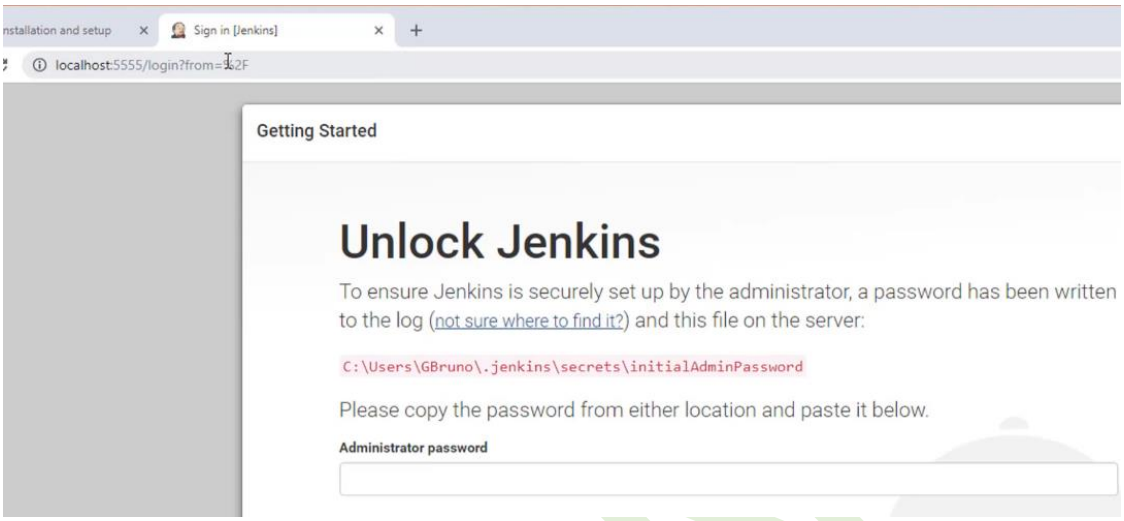
   ***Please also note**: If you specify no port number, by default Jenkins will run on port 8080.*

5. Now you should see the following message, confirming Jenkins is up and running:

```
INFO: Finished Download metadata. 13,546 ms
Dec 10, 2018 8:57:27 PM hudson.model.UpdateSite updateData
INFO: Obtained the latest update center data file for Update
Source default
Dec 10, 2018 8:57:27 PM jenkins.InitReactorRunner$1 onAttain
ed
INFO: Completed initialization
Dec 10, 2018 8:57:27 PM hudson.WebAppMain$3 run
INFO: Jenkins is fully up and running
```

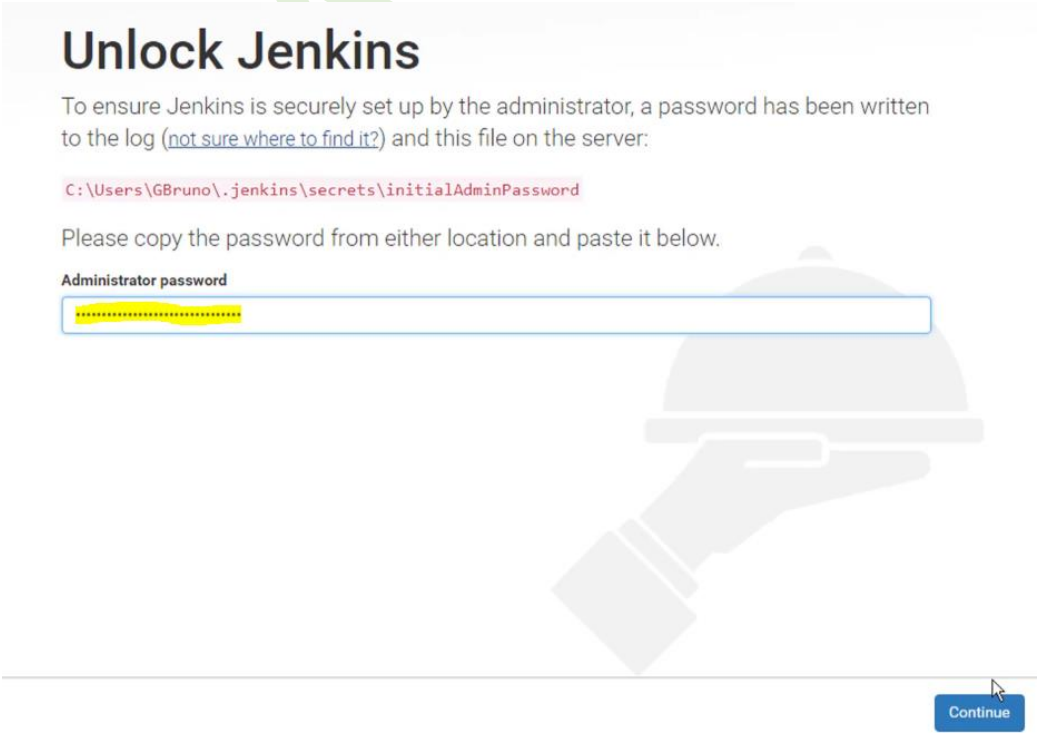
6. Open up Chrome browser and access the following URL (Dependent on the port number you have specified, if you haven't specified a port number then you will need to use 8080). http://localhost:5555/

7. By default you will be taken to the following url:

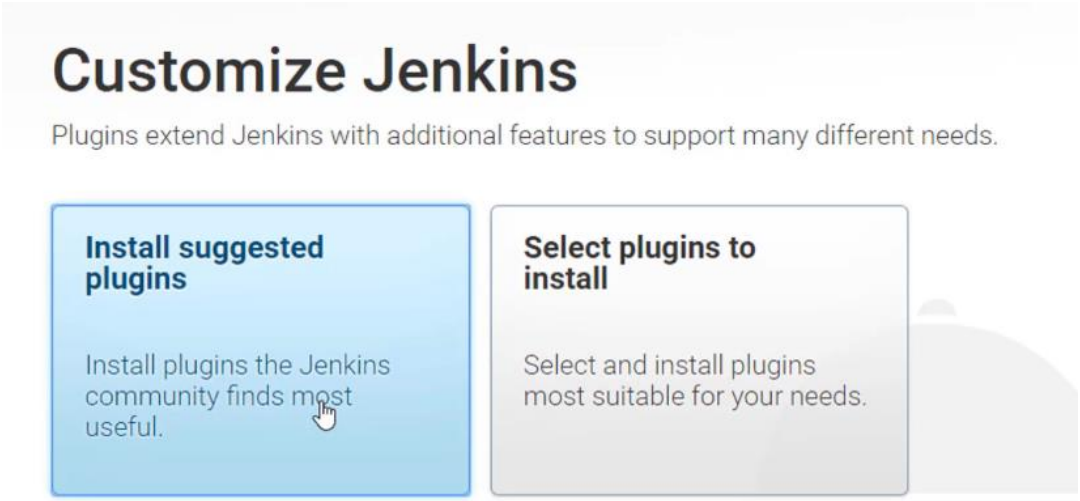


8. Now access the listed directory to open the "initialAdminPassword" file and open the file in the text editor of your choice (i.e. SublimeText).

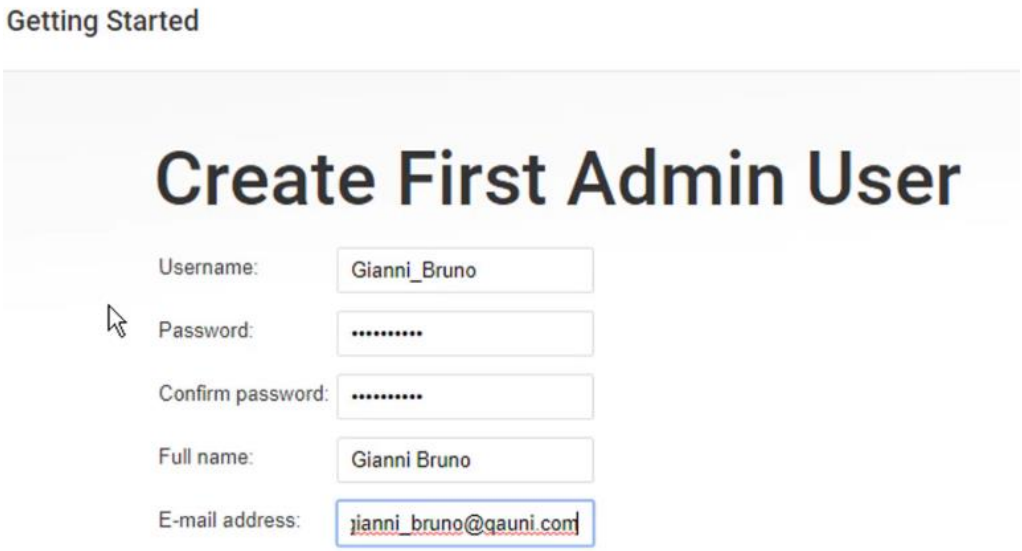
9. Now paste the unique password into the text field and click on the "Continue" button as listed:



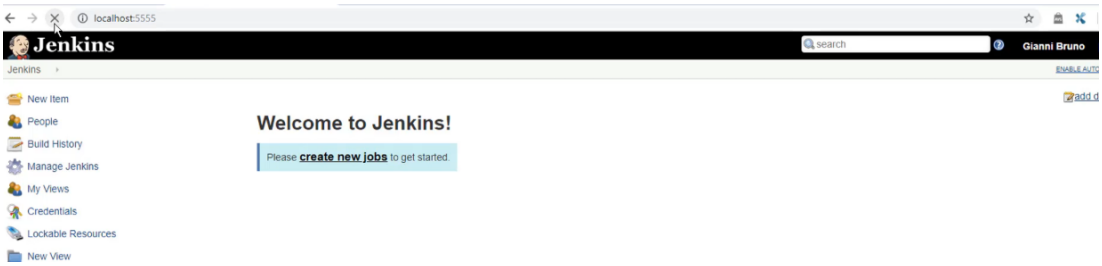
10. Now click on the "Install suggested plugins" option:



11. Now you will be taken to the “Admin account creation” page, specify the relevant credentials in order to create however make sure you remember the details!



12. Click on “Save and Continue” button.
13. On the next page leave the Jenkins URL as it is (http://localhost:5555) and click “Save and Finish” button.
14. Now you will be taken to the Jenkins homepage:



15. Remember you can stop your instance of Jenkins by accessing the Git Bash instance which is running Jenkins and press the following keyboard keys:
    Windows Users: **Ctrl + C**
    Mac Users: **Control + C**

    Or even use the following URLs:

```
http://localhost:8080/exit
http://localhost:8080/restart
http://localhost:8080/reload
```

Lecture 99 - Jenkins Installing Plugins and Setting Up NodeJS

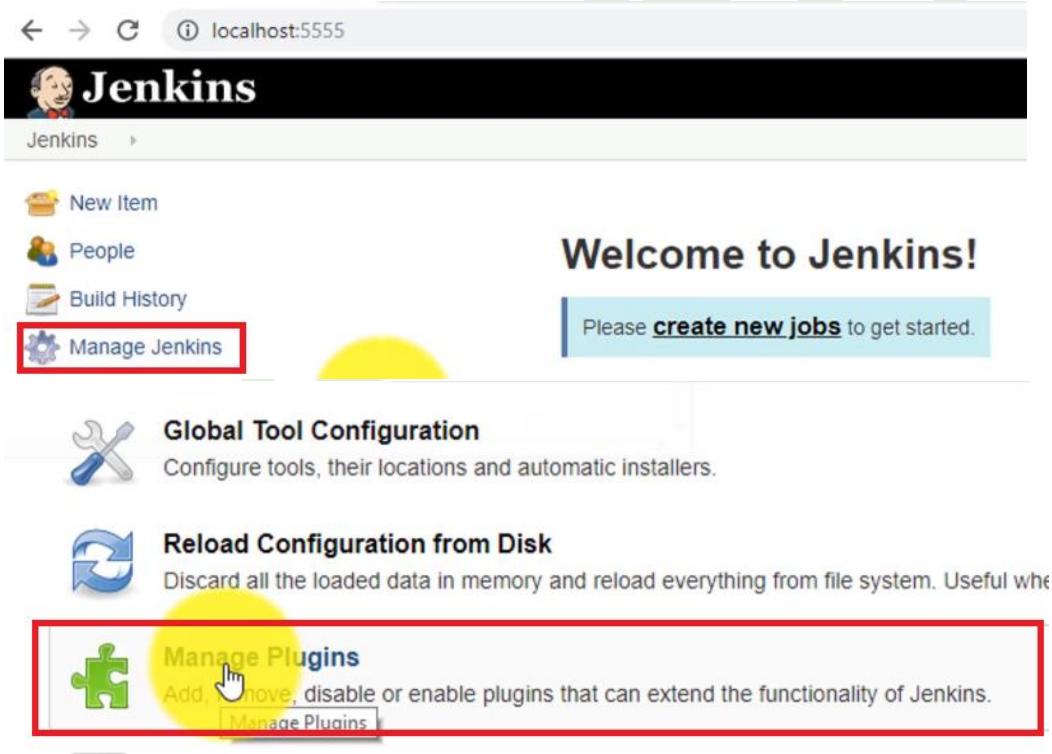
Jenkins enables the test automation engineer to download and install many available (Free) plugins to enhance their instance of Jenkins. We will inspect the Jenkins Plugin Marketplace and in turn select and install specific plugins required to get our tests up and running within Jenkins.

Key Points:

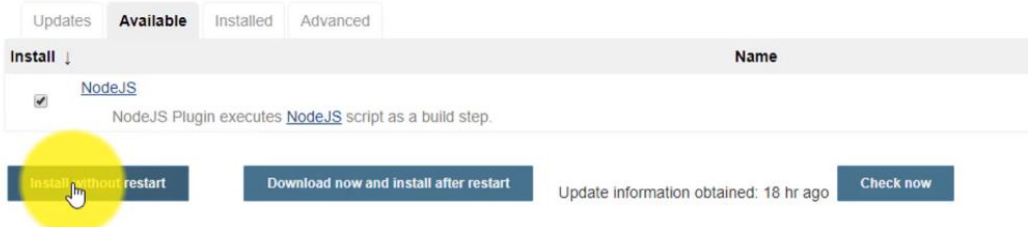
- We will inspect the Jenkins Plugin Marketplace and install the following plugins:
  NodeJS – *Required to execute our automation framework via Jenkins.*
  HTML Publisher – *Will be used in later lectures, to produce test reports within Jenkin.*
  Post build task – *Required to perform post build tasks such as moving files to another location once the Jenkins build has fully completed.*
- We will also setup NodeJS within Jenkins.

Instructions:

1. Login into the Jenkins Portal using your credentials, if you're not already logged in.
2. On the main Jenkins UI click on "Manage Jenkins", then click on "Manage Plugins":



3. Now click on the "Available" tab, **search** and **select** the "NodeJS" plugin:

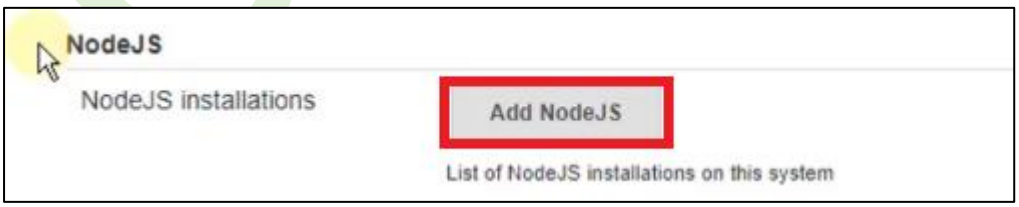


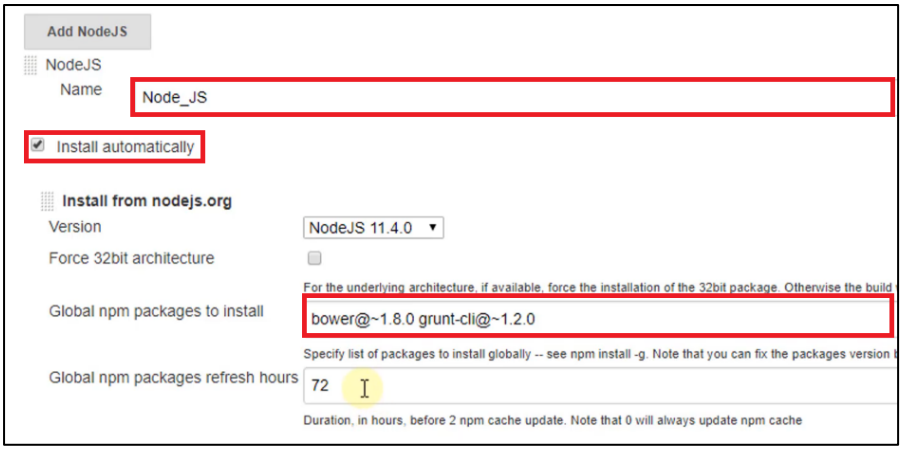
4. Click on "Install without restart".
5. Now perform steps 3 & 4 for the following plugins: **HTML Publisher** and **Post build task.**
6. The next step is to setup NodeJS, access the main Jenkins homepage by clicking on the header icon:

7. Now click on “Manage Jenkins”:

8. Now click on “Global Tool Configuration” as listed:

9. Now locate the “NodeJS” option and click on the “Add NodeJS” option and provide the following information:
   Name: **Node**
   Tick / enable the option: **“Install automatically”**
   Global npm package to install: **bower@~1.8.0 grunt-cli@~1.2.0**

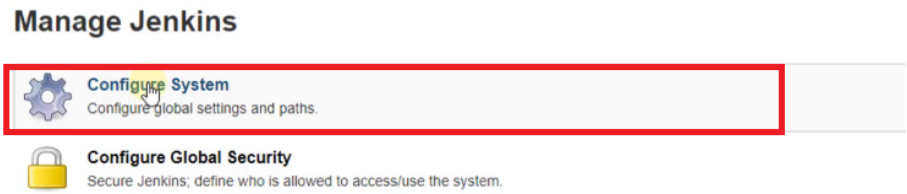




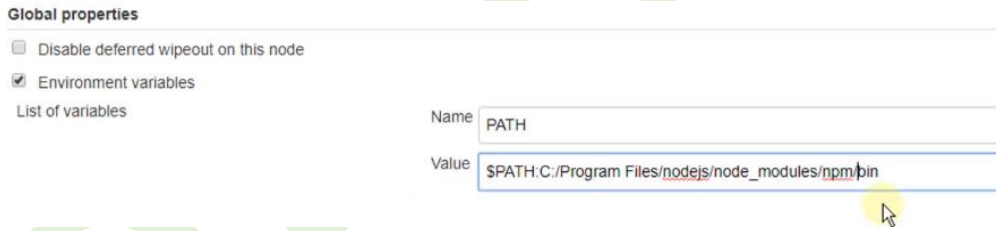
10. Now click on the “Save” option.

***Please note: Apply** will apply your settings and you will remain on the same UI page, **Save** will apply the settings and take you back to the previous page.*

11. The next step is to create a Node environment variable, on the “Manage Jenkins” page click on “Configure System” as listed below:



12. Now locate the “Global properties” section as listed below:



13. You will need to provide the following information dependent on system type (Windows or Mac):

Windows Machine:

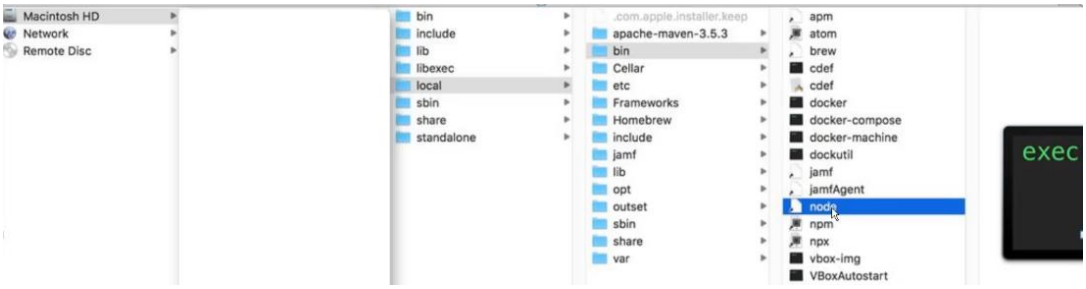
Name: **PATH**

Value: **$PATH: C/Program Files/nodejs/node_modules/npm/.bin**

MAC Machine:

Name: **PATH**

Value: Refer to the image below (**Example**: HD/local/bin/node):



14. Now click on “Save” in order to apply the recent changes.

Lecture 100 - Trigger our Tests via Jenkins

In this lecture we will take a look at how we can trigger specific tests to run within Jenkins, setup a base URL and also take a look at other settings.

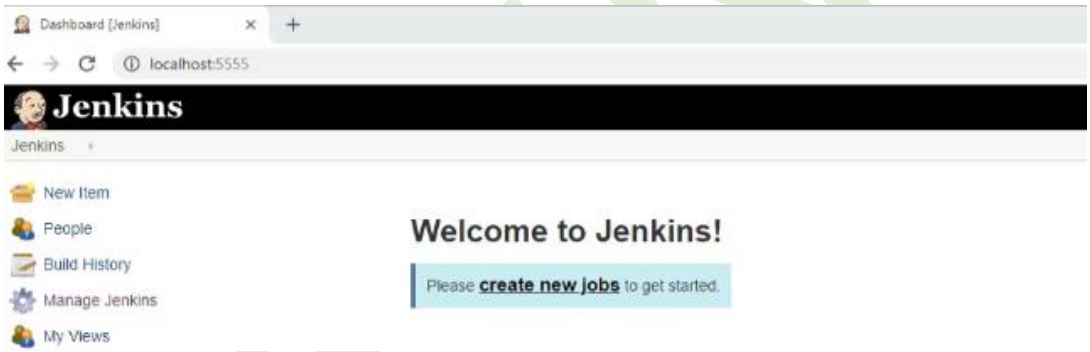
Key Points:

- Learn the difference between a **Jenkins job** and **Jenkins builds.**
- Create a new Jenkins job.
- Take a look at some of the settings which Jenkins makes available.
- Target and trigger specific tests via a series of commands.
- Setup a base URL, which is changeable within Jenkins.
- Trigger Jenkins builds and view test results of a given Jenkins build.

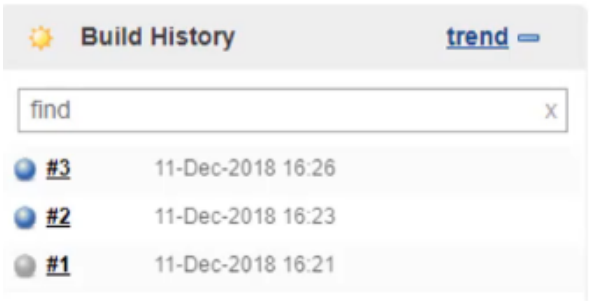
Instructions:

1. First it's important to know the differences between a Jenkins job and build:

Jenkins Job: Is simply a job to execute build(s), users can customise and alter settings via the Jenkins job UI.

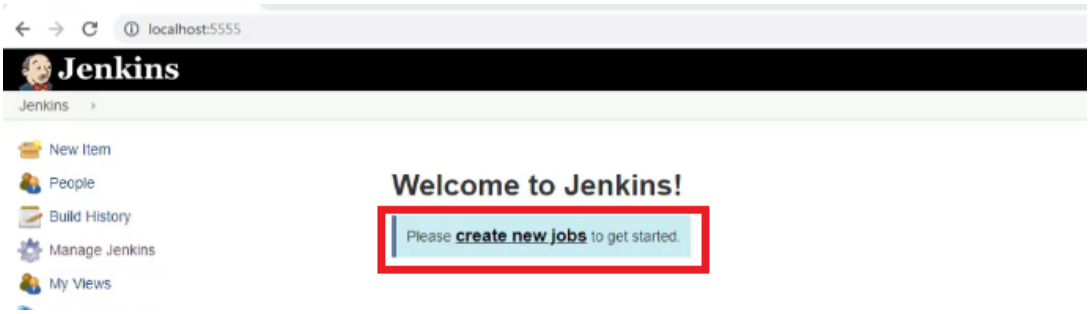


Jenkins build(s): Are executions of your Jenkins Job, take a look at the following example:

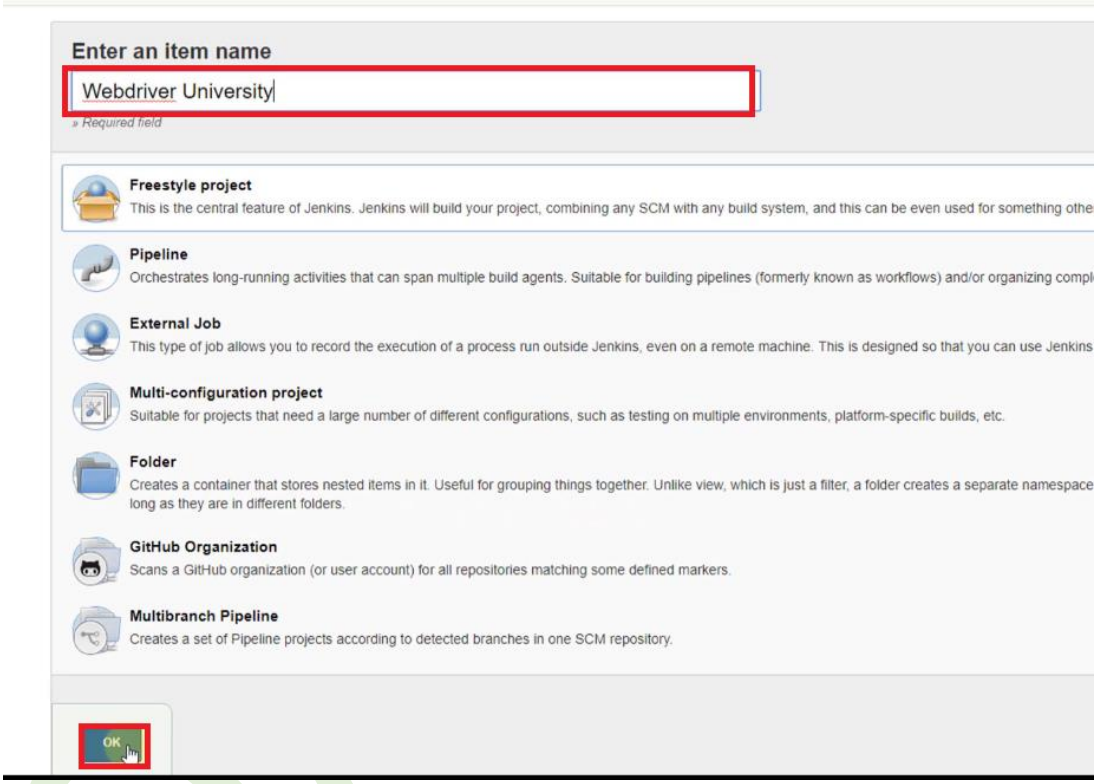


As you can see from the image above three builds have been individually executed, the latest build was triggered at: "16.25". Each build contains details of the execution for example build #1 was aborted hence why it's showing as Grey, build #3 passed and therefore showing as Blue (Default colour for passed builds within Jenkins).

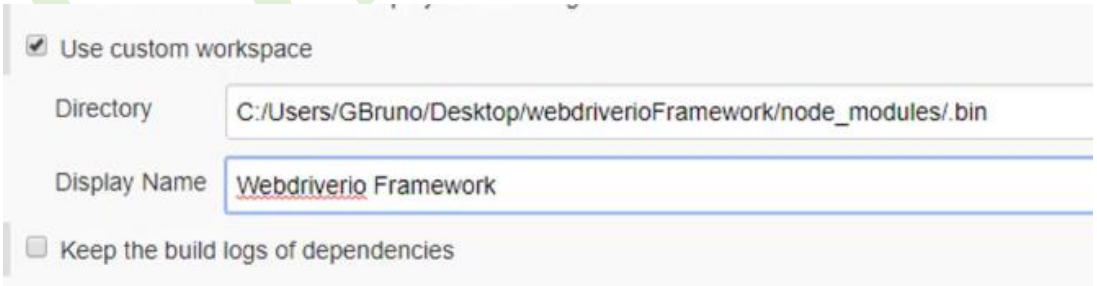
1. First we need to create a job within Jenkins, access the main Jenkins UI and click on the "create new jobs" option:



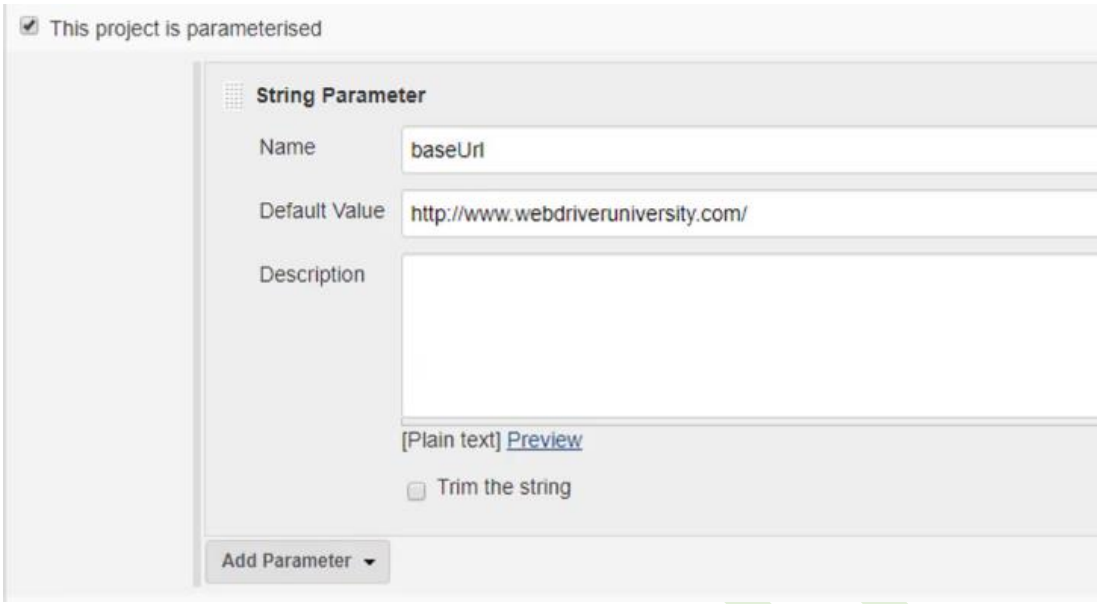
2. Now provide the Job with a name of “Webdriver University” as listed within the image below and click on “ok”.



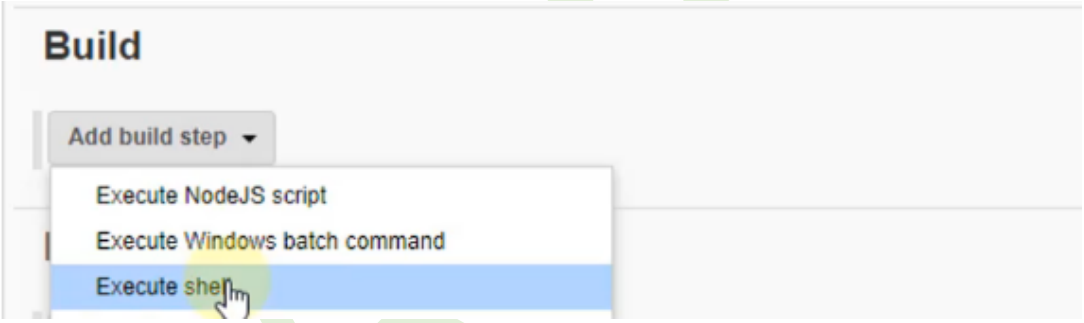
3. Next you will be presented within the Jenkins job UI, click on the option “Use custom workspace” and provide the path of your WebDriverIO test frameworks directory:



4. Now click “Apply”.
5. The next configuration change is to setup a “String Parameter” within Jenkins which will enable the user to alter the tests base URL. Click on the option / tick box to enable the option “This project is parameterised” and provide the following information:
   Name: baseUrl
   Default Value: http://www.webdriveruniversity.com/

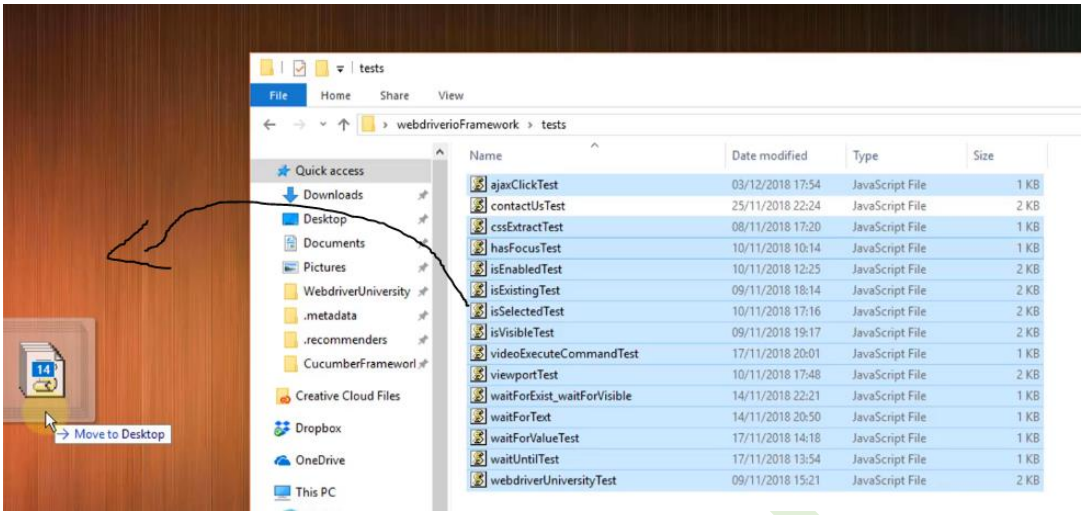


6. Again click on “Apply” to save the changes.
7. The next task to setup the script (Code) which will trigger our tests to run within Jenkins, under the “Build” section, click on “Add build step” and select “Execute shell” from the dropdown menu as listed below:

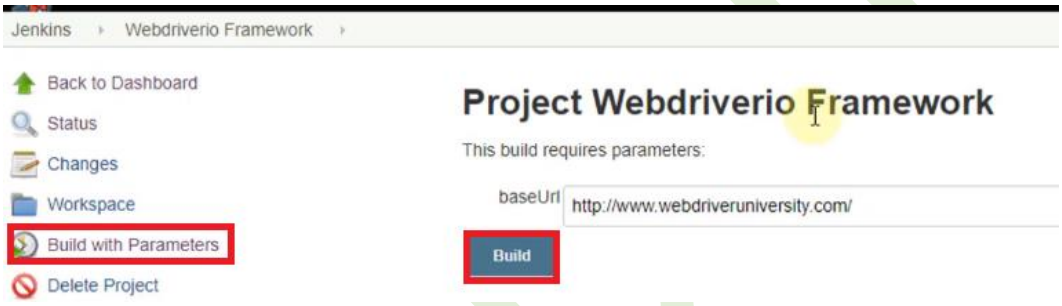


8. Now add the following commands within the text field:

9. As you can see we are using a **$** symbol which will pass the parameter (Base URL) which the user provides via Jenkins.
10. Now click on “Save” to apply our recent changes.
11. Now we will move all tests apart from “contactUsTest.js” from our tests folder onto the desktop in order to validate whether Jenkins is able to execute a specific test.
12. As you can see from the image below all tests are dragged and dropped outside of the tests folder onto the desktop apart from the “contactUsTest”:

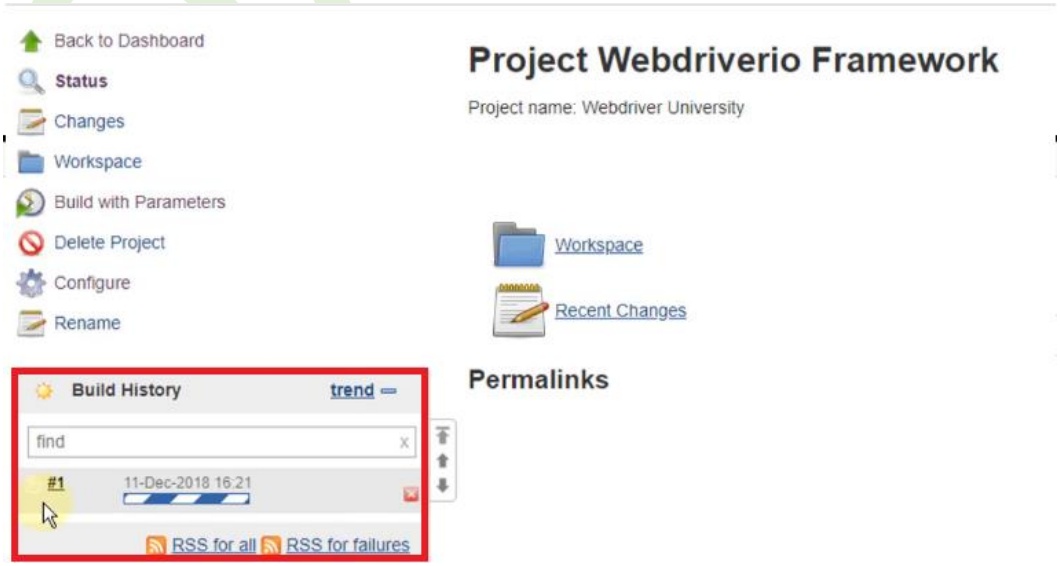


13. Now let's trigger a build via Jenkins, access the main Jenkins UI and click on "Build with Parameters" and then click on "Build" as listed below:



Please note: The default URL is: http://www.webdriveruniversity.com/ because in our previous steps listed above, we set the base URL (String parameter) with the default value of http://www.webdriveruniversity.com/ this value can be altered to the URL of your choice.

14. Now you will be able to see a Jenkins build which has been triggered via the Jenkins homepage UI.

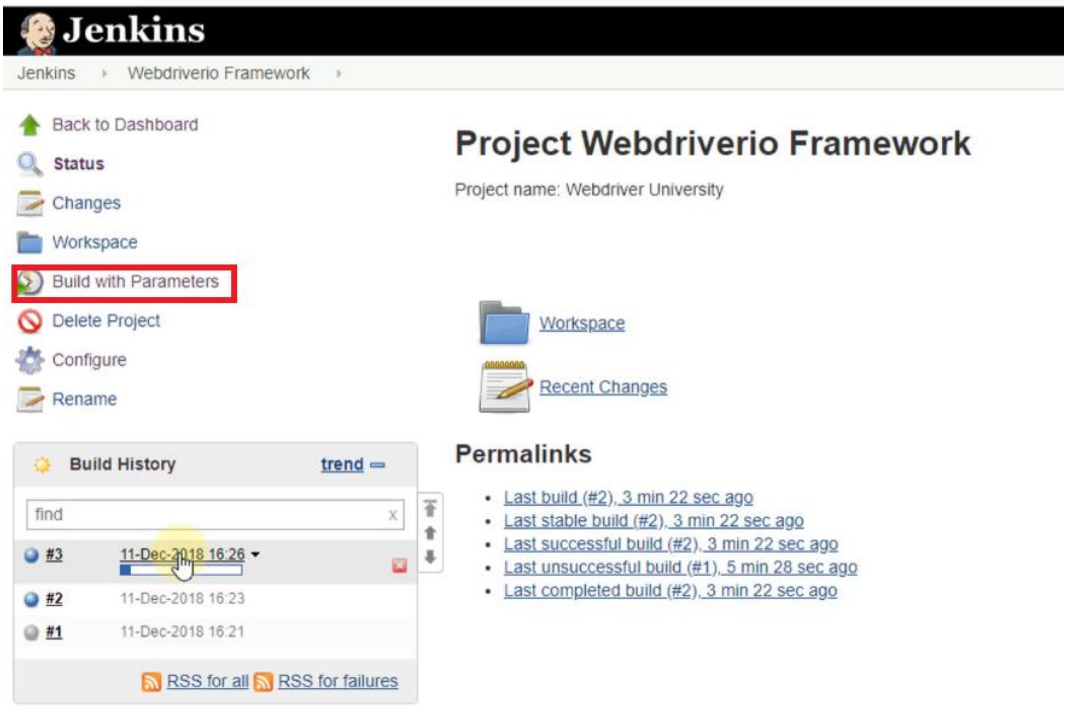


15. Click on the build #1 to be taken to the build homepage.

16. You can inspect more information about the build by clicking on the option “Console output” as listed:

17. Now you should see a new browser instance open and see the Contact Us test executing the relevant actions.
18. Once the build has fully finished you should see details of the execution, including whether the tests have passed or failed (4 passing).

19. Go ahead and trigger more builds within Jenkins, one build can only be executed at any given time, the other builds will remain within the build queue waiting to be executed.


Jenkins
Jenkins › Webdriverio Framework ›
Back to Dashboard
Status
Changes
Workspace
Build with Parameters
Delete Project
Configure
Rename
Build History
trend
find
#3 11-Dec-2018 16:26
#2 11-Dec-2018 16:23
#1 11-Dec-2018 16:21
RSS for all
RSS for failures
Project Webdriverio Framework
Project name: Webdriver University
Workspace
Recent Changes
Permalinks
Last build (#2), 3 min 22 sec ago
Last stable build (#2), 3 min 22 sec ago
Last successful build (#2), 3 min 22 sec ago
Last unsuccessful build (#1), 5 min 28 sec ago
Last completed build (#2), 3 min 22 sec ago

Lecture 101 - Jenkins Adding Parameters

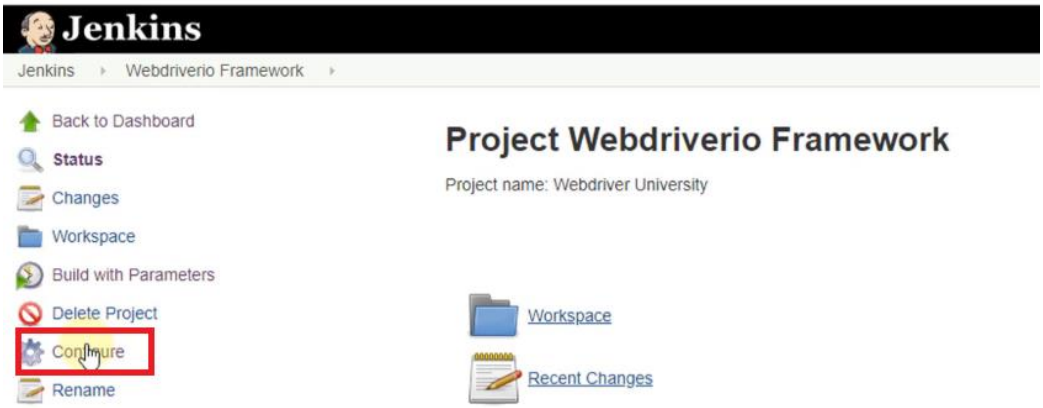
As you have previously seen, we can add parameters within Jenkins to alter specific properties of our automation framework; in this lecture we will be adding another parameter which will enable the user to target and execute the desired test of their choice.

Key Points:

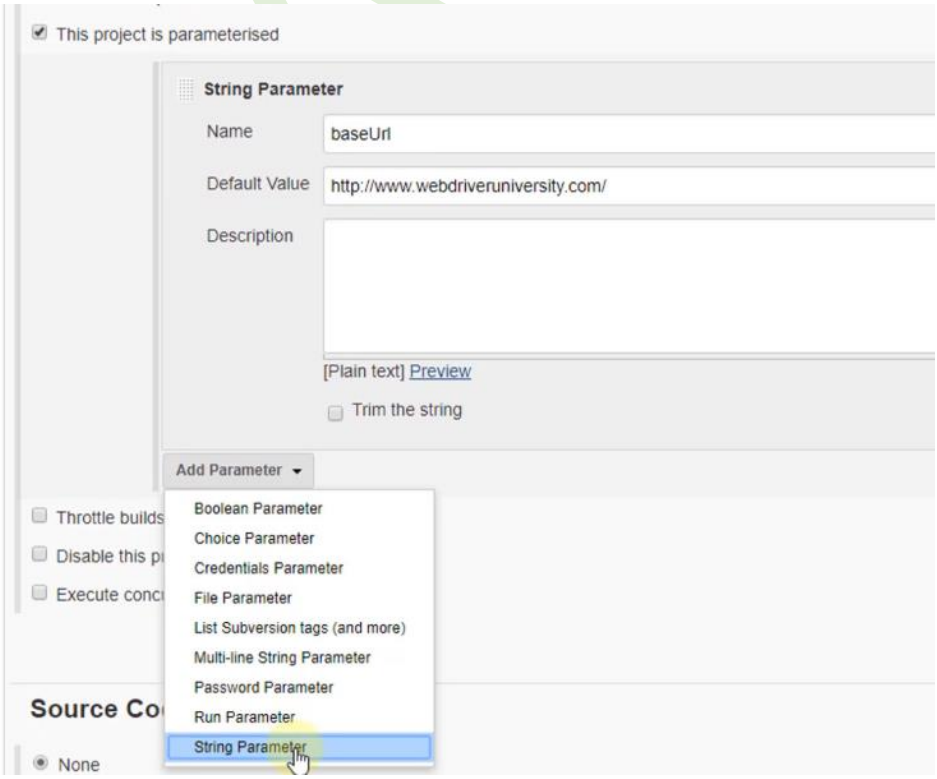
- Add a String parameter within Jenkins, enabling the user to target and execute the test of their choice.
- Make some minor tweaks to the wdio file to remove code which will no longer be used within this module.

Instructions:

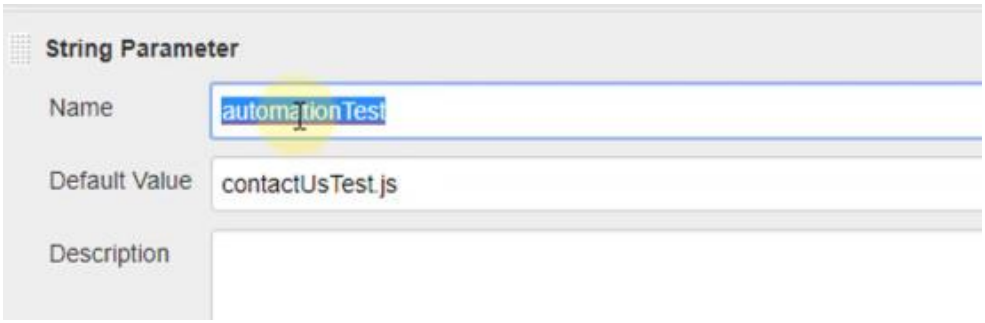
1. Click on the "Configure" option on the main Jenkins job UI:



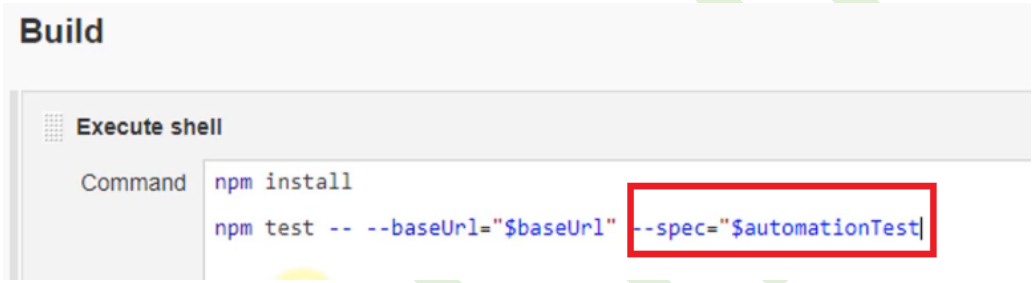
2. Now locate the "This project is parameterised" section, click on the "Add Parameter" button and select "String Parameter" from the dropdown menu:



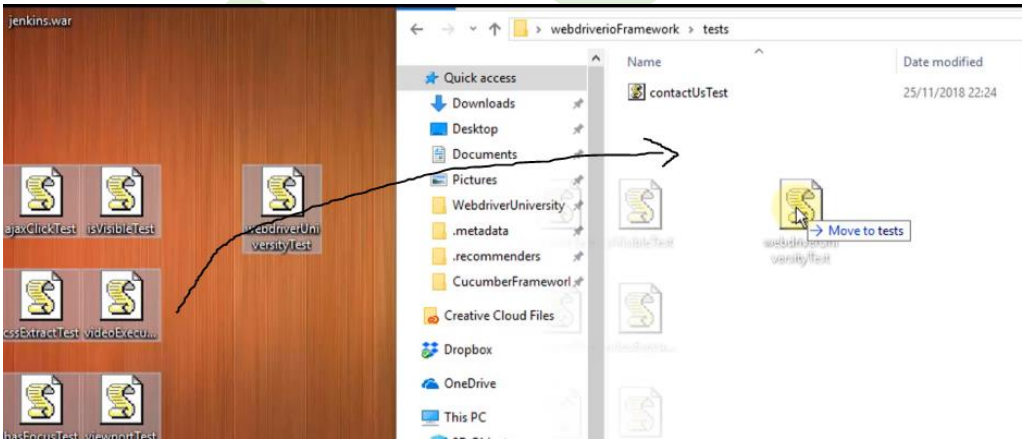
3. Now add the following values:
   Name: automationTest
   Default value: contactUsTest.js



4. Now click the “Apply” button to save the changes.
5. Next navigate to the “Build” section and append the following code to the existing command:

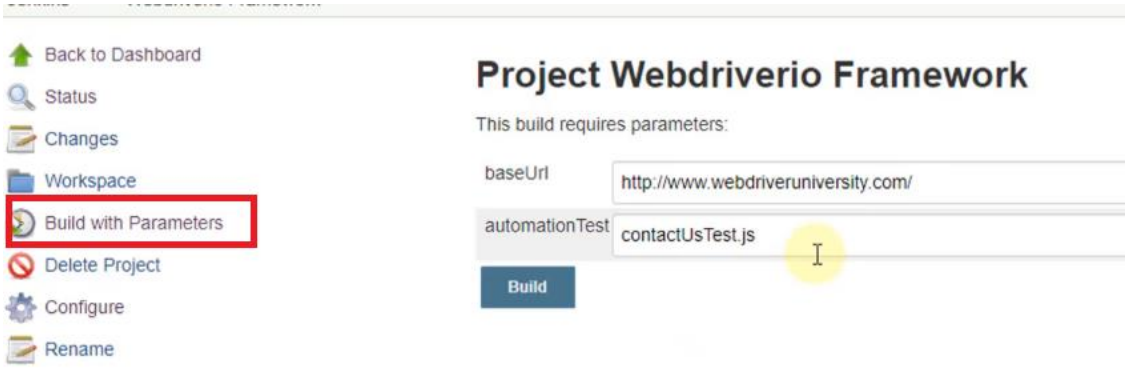


6. Now click “Save” to save the recent changes.
7. Now drop and drag the other tests which were previously on the desktop into the tests folder.



8. Now go back to Jenkins and click on the Jenkins job:

9. Click on the “Build with Parameters” option, you will see that the default value for the “automationTest” field is “contactUsTest.js” the value we previously specified to be the default value.



10. Now click on the “Build” button and you will see that the “contactUsTest.js” test will be triggered via Jenkins.
11. Inspecting the build via the “Console Output” option you will see that there are specific messages being outputted to the console window:

```
+ npm test -- --baseUrl=http://www.webdriveruniversity.com/ --spec=contactUsTest.js

> webdrivierioframework@1.0.0 test C:\Users\GBruno\Desktop\webdriverioFramework
> wdio "--baseUrl=http://www.webdriveruniversity.com/" "--spec=contactUsTest.js"

...Error: ENOENT: no such file or directory, open 'C:\Users\GBruno\Desktop\webdriverioFramework\errorShots\ERROR-
    at screenshot() - C:\Users\GBruno\Desktop\webdriverioFramework\node_modules\webdriverio\build\lib\commands\save
    at saveScreenshot("./errorShots/ERROR-chrome-1544547625790.png") - C:\Users\GBruno\Desktop\webdriverioFramework
    at elementIdText("0.5829506424454969-1") - C:\Users\GBruno\Desktop\webdriverioFramework\node_modules\webdriver
```

12. Now open the wdio file via Git Bash and comment out the following lines of code, as we will **not** being using these types of reporting within Jenkins going forward:

Before:

```
reporterOptions: {
    junit: {
        outputDir: './reports/junit-results/'
    },
    json: {
        outputDir: './reports/json-results/'
    },
    allure: {
        outputDir: './reports/allure-results/',
        disableWebdriverStepsReporting: true,
        disableWebdriverScreenshotsReporting: false,
        useCucumberStepReporter: false
    }
},
```

After:

```
reporterOptions: {
    /**
    junit: {
        outputDir: './reports/junit-results/'
    },
    json: {
        outputDir: './reports/json-results/'
    },
    **/
    allure: {
        outputDir: './reports/allure-results/',
        disableWebdriverStepsReporting: true,
        disableWebdriverScreenshotsReporting: false,
        useCucumberStepReporter: false
    }
},
```

13. Now remove the following types of reports from the reporter's section within the wdio file:

Before:

```
reporters: ['dot', 'junit', 'json', 'allure'],
```

After:

```
reporters: ['dot','allure'],
```

14. Finally comment out the following lines of code within the wdio file, as we will no longer need this code:

Before:

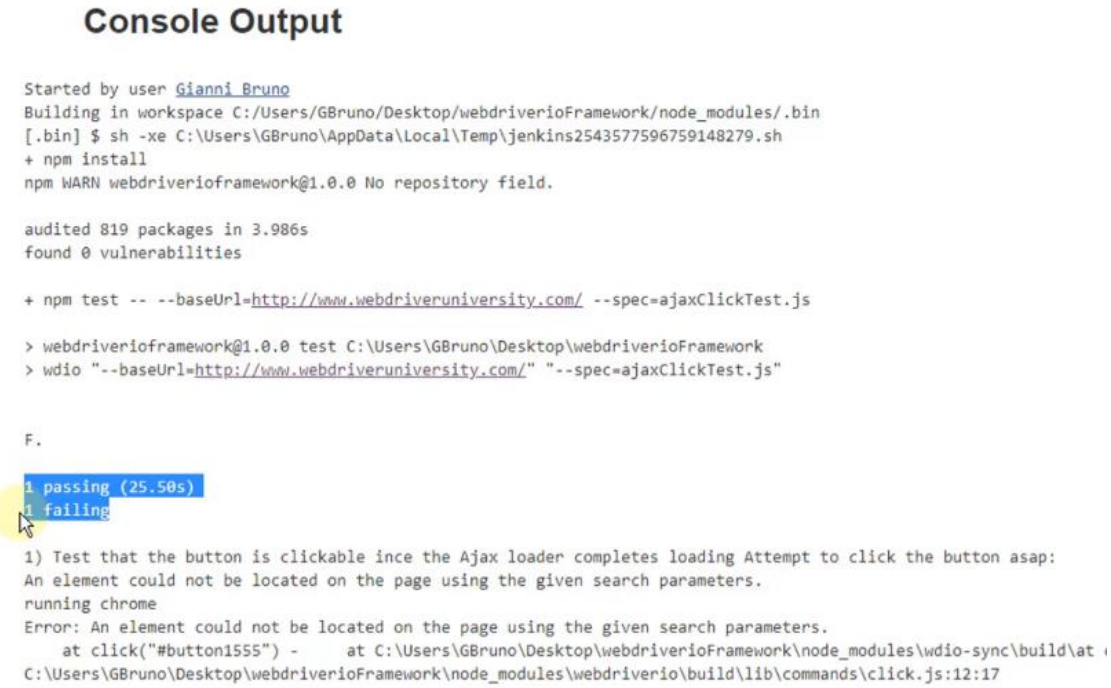
```
after: function (result, capabilities, specs) {
    var name = 'ERROR-chrome-' + Date.now()
    browser.saveScreenshot('./errorShots/' + name + '.png')
},
```

After:

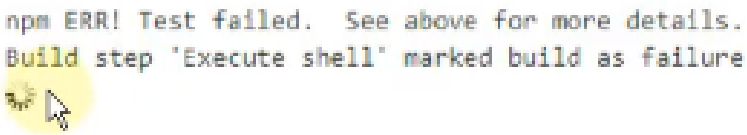
```
/**
after: function (result, capabilities, specs) {
    var name = 'ERROR-chrome-' + Date.now()
    browser.saveScreenshot('./errorShots/' + name + '.png')
},
**
```

15. Now save the wdio file.
16. Now navigate back to Jenkins and trigger a new build via the "Build with Parameters" button and change the automationTest text fields value to target the "ajaxClickTest.js":

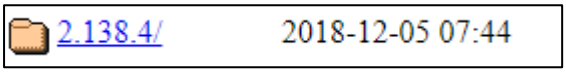
17. Now you should see the "ajaxClickTest" execute within a new browser instance, if you inspect the builds logs via the "Console Output" button, you should no longer see the previous exception messages being outputted to the console window:



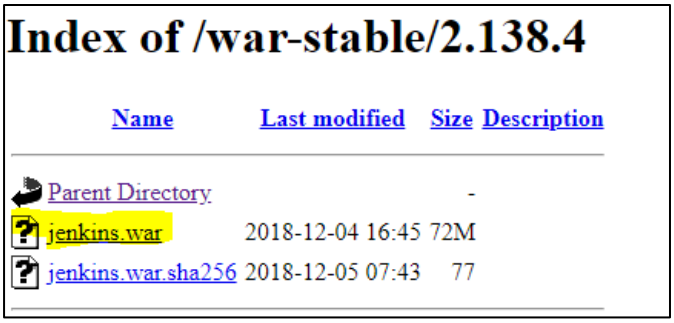
18. **IMPORTANT!** If your build doesn't seem to complete / finish and your seeing the following loader within the build "Console output":



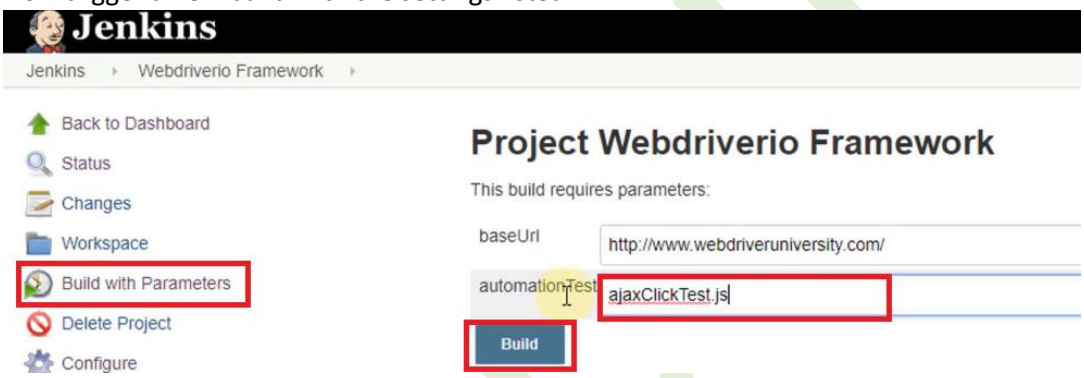
19. You may need to try and downgrade your instance of Jenkins.
20. Access the Git Bash instance which is currently running Jenkins and type: *Ctrl + C* (On a windows machine) or *Control + C* (On a Mac machine) to stop the instance of Jenkins.
21. Now delete or rename the "jenkins.war" (Don't worry you will NOT lose your settings etc.)
22. Now access the following URL: http://mirrors.jenkins.io/war-stable/
23. Navigate to an older release version such as: "2.138.1" and click on the link:



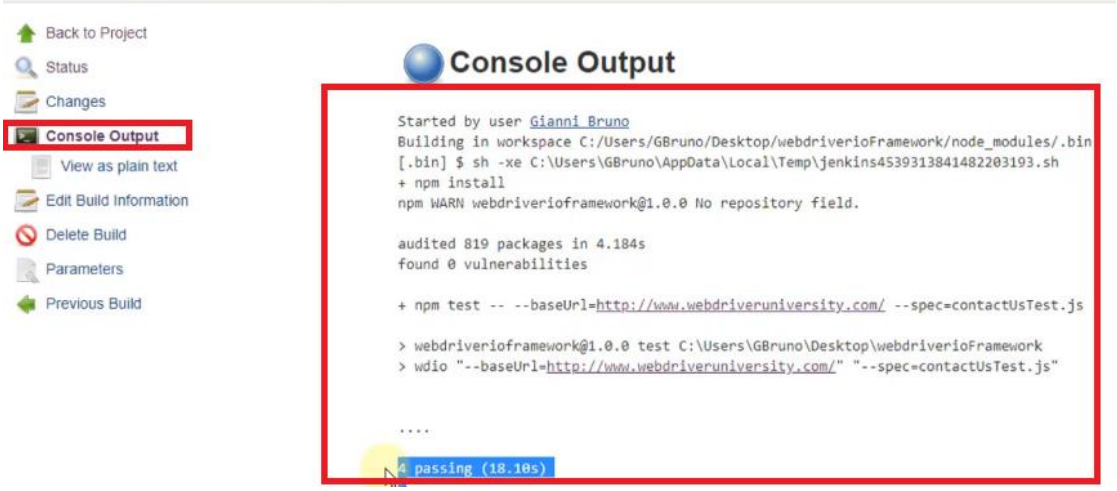
24. Now download the Jenkins.war file and drag and drop it to your desired location:



25. Now open Git Bash and navigate to the location of your jenkins.war file, run the following command (Dependent on the port your Jenkins instance is using):
    java –jar jenkins.war --httpPort=5555
26. Now you should see Jenkins start up, log into your account and you should see all the previous setting preserved.
27. Now trigger a new build with the settings listed:



28. Inspect the builds via the “Console Output”, you should no longer experience issues with the build struggling the complete / finish.



Lecture 102 - Jenkins Adding Additional Logging Information

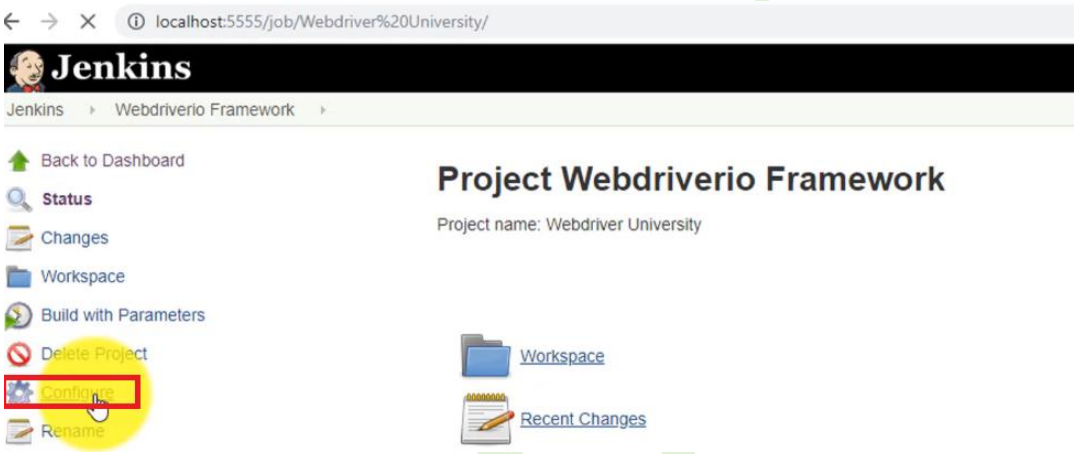
Now we are going to look at how can add additional logging data within Jenkins.

Key Points:

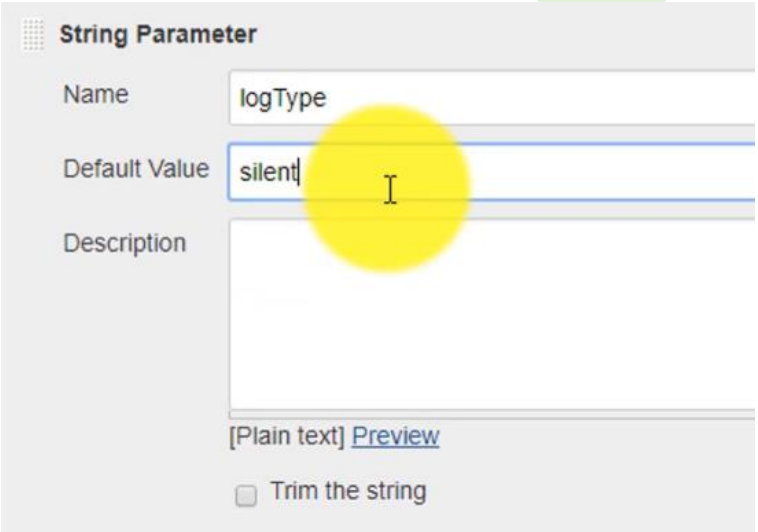
- Add a parameter within Jenkins to alter the type of logging data which will be outputted when triggering a build within Jenkins.
- Inspect various types of logging options available.

Instructions:

1. Refer to the following URL to view the types of logging options available: http://v4.webdriver.io/guide/getstarted/configuration.html
2. Access the main Jenkins job UI and click on "Configure":



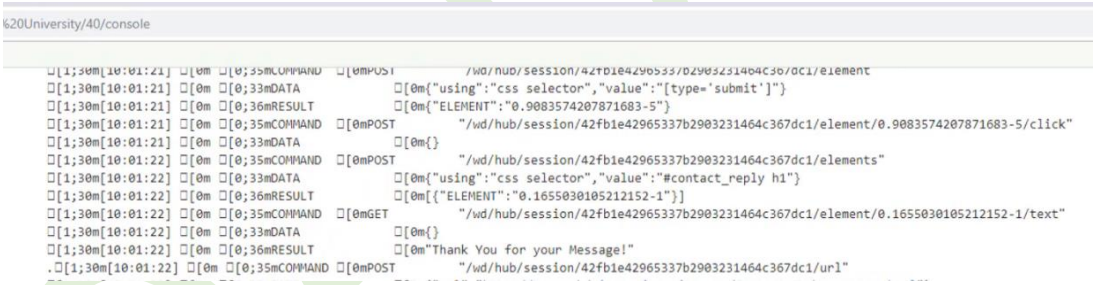
3. Add another string parameter and set the default value to "silent":



4. Click "Save" to apply the changes.
5. Now extend upon the build execute shell as listed:

6. Now click the “Save” button to apply the changes.
7. Now trigger a new build via Jenkins however alter the default value to “verbose”:

8. Once the Jenkins job begins, inspect the console output and you will see additional information being outputted to the console window:



9. Now when triggering a Jenkins build the user can alter the base URL, test to target and also the type of logging to be outputted to the console window.

Lecture 103 - Jenkins Execute Our Tests when Ever We Want

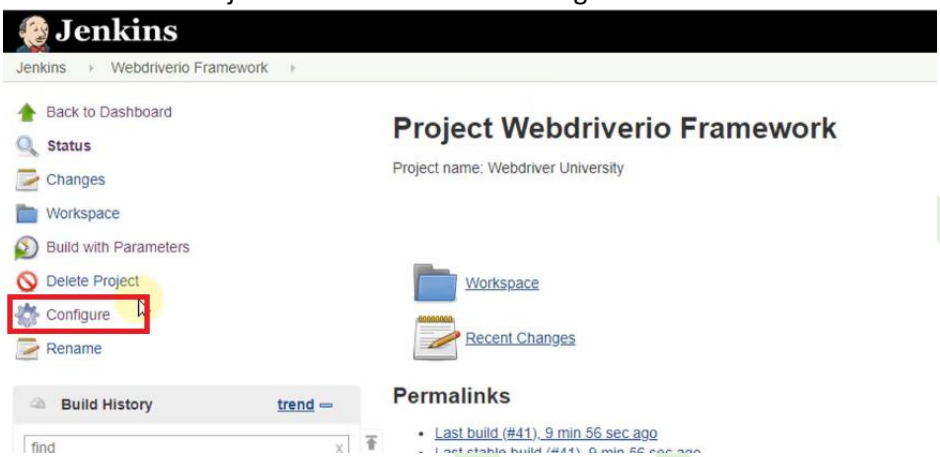
In this lecture we will learn how we can automatically trigger jobs / builds within Jenkins using the schedule jobs functionality; for example, automatically triggering jobs during specific times of given day(s).

Key Points:

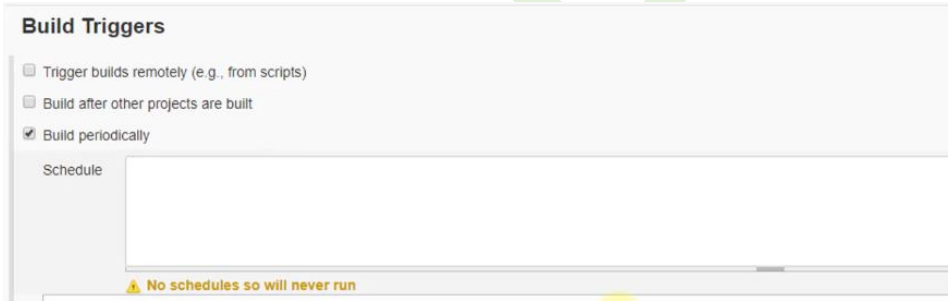
- Learn how to automatically trigger Jenkins builds for specific day(s) and time(s).
- Use cron (Time base schedule).

Instructions:

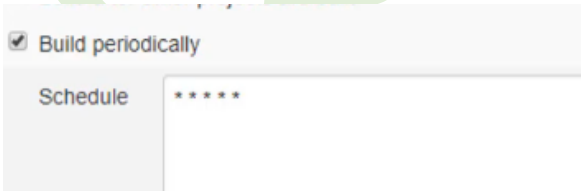
1. Access the Jenkins job UI and click on the "Configure" button:



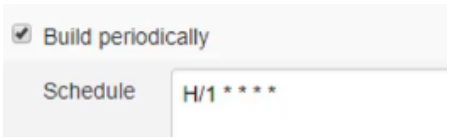
2. Now navigate to the "Build Triggers" section and enable the "Build periodically" tick box:



3. Add the following code and click "Save":



4. Now your project (Automation framework) tests will get triggered automatically and continuously; one build after the other.
5. Now alter the value to the following:



6. Now the tests will be executed automatically every hour, for example 8am, 9am, 10am….

Lecture 104 - Generating Allure Reports within Jenkins – Part 1

Lecture 105 - Generating Allure Reports within Jenkins – Part 2

(Lectures above combined into one section in this document)

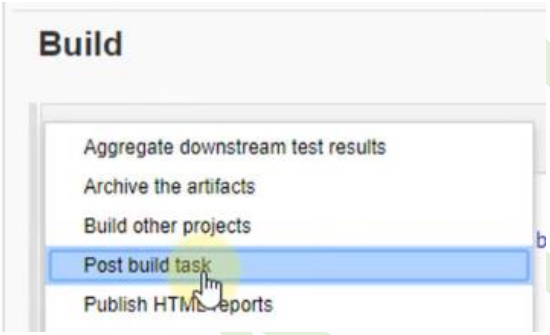
We will be using Jenkins post build tasks to create scripts; with the end of goal of backing up all test(s) reporting history generated from executed builds.

Key Points:

- Learn how to use Jenkins "Post build tasks" to perform specific actions; such as backing up tests results from a given build.
- Learn why it's important to back up your data (Test results).

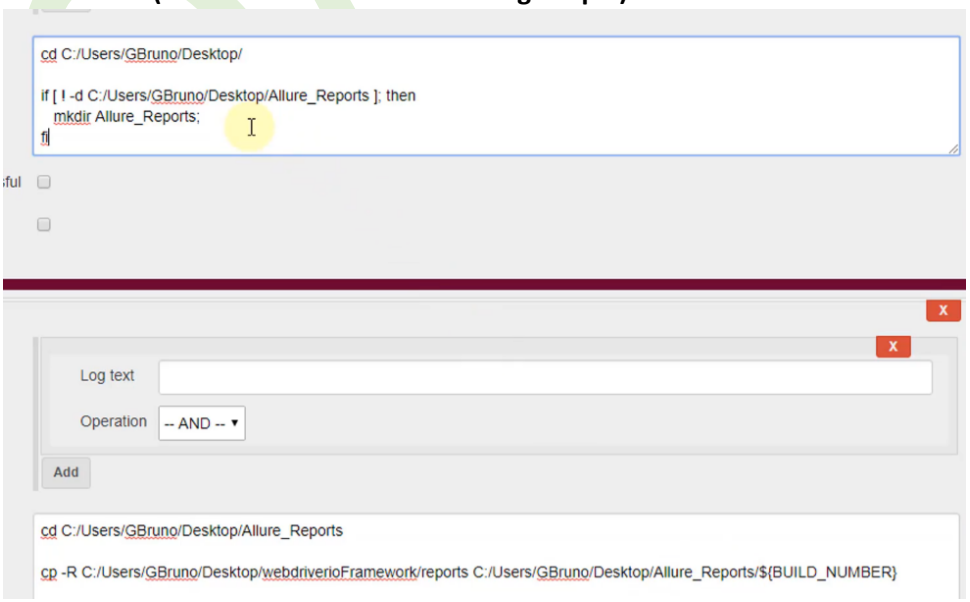
Instructions:

1. Access the current Jenkins job (Project Webdriverio Framework)
2. Click on the "Configure" button.
3. Navigate to the "Build" section and click on "Add post build action" then select the following option (Post Build Task)

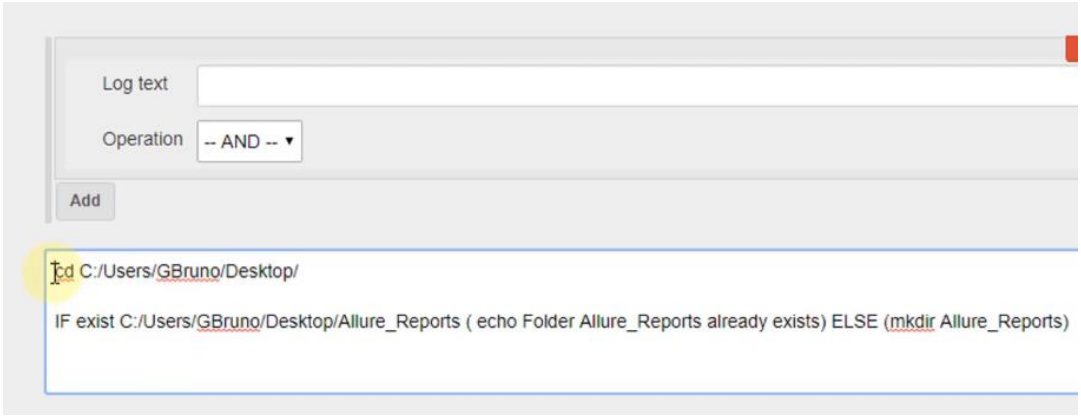


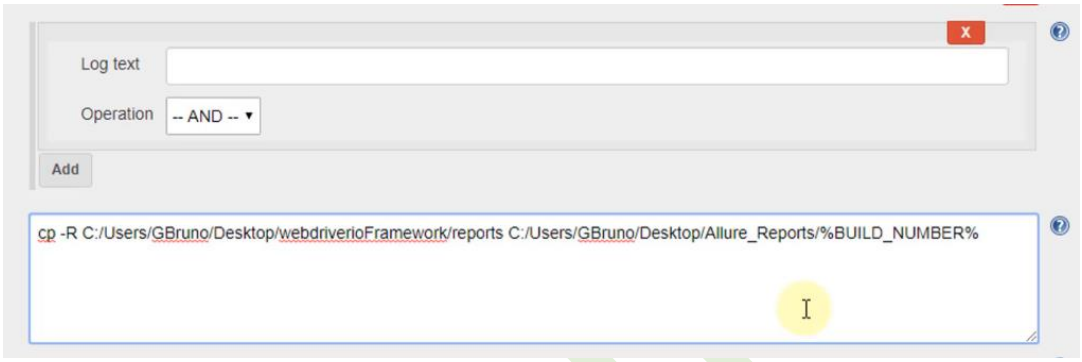
4. Make sure you add two separate tasks as listed below and add the following scripts dependent on system type (**Mac or Windows**, you can also leave the tick boxes un ticked):

MAC USERS (Will need to add the following Scripts):

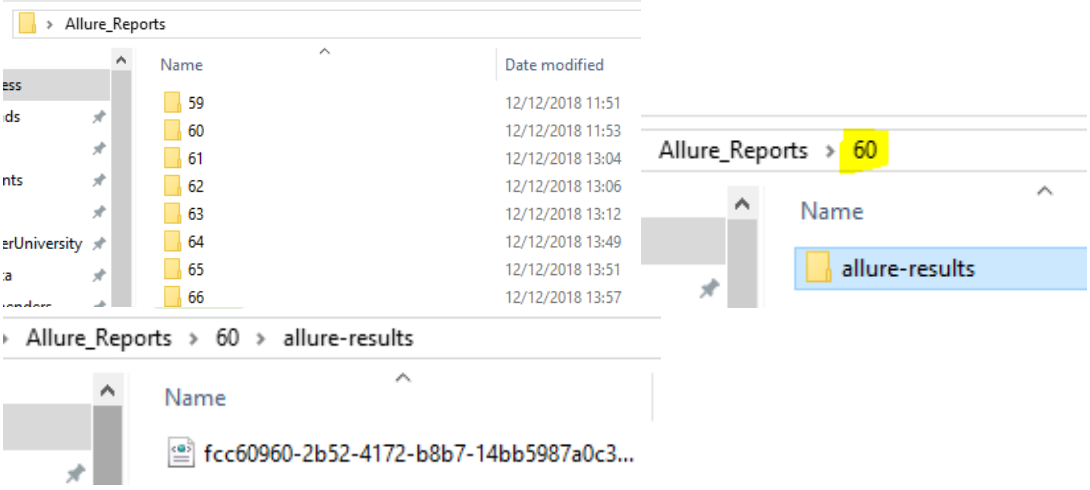


WINDOWS USERS (Will need to add the following Scripts):





5. The **first script** will check to see whether the folder "Allure_Reports" exists based upon your desired location (I'm using the desktop as my directory), if the folder does not exist then a new folder with the name of "Allure_Reports" will be created, if a folder already exists with the given name then the script will do nothing.
6. The **second script** will copy the **latest tests results folder** housed within the main framework directory (webdriverioFramework) and copy the folder and files across to the new folder in turn creating another folder based upon the "Number" of the current build as listed below:



7. As you can see, all previous test automation results have now been backed up locally, enabling the user to generate Allure test reports using the allure commands listed within the previous lectures.
8. Now trigger as many Jenkins builds as you want, each time a build completes you will see a new folder created with the name of the current build number which also contains the test results (Allure xml) relevant to that build.

Lecture 106 - Generating Allure Reports within Jenkins – Part 3

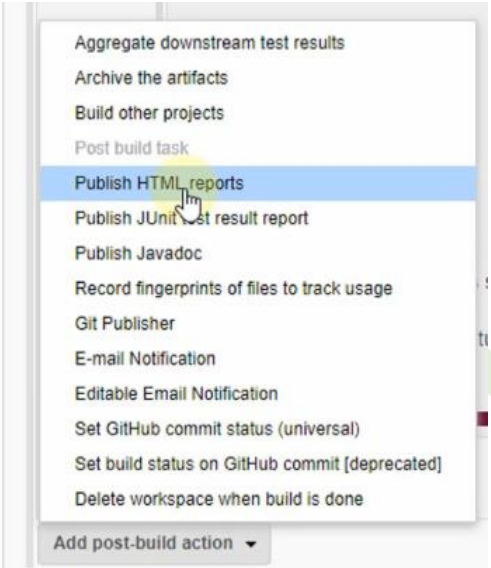
Allure reports has its own Jenkins plugin however it can be fairly cumbersome to get up and running therefore we will be using "HTML Reports" plugin to generate our Allure reports within Jenkins.

Key Points:

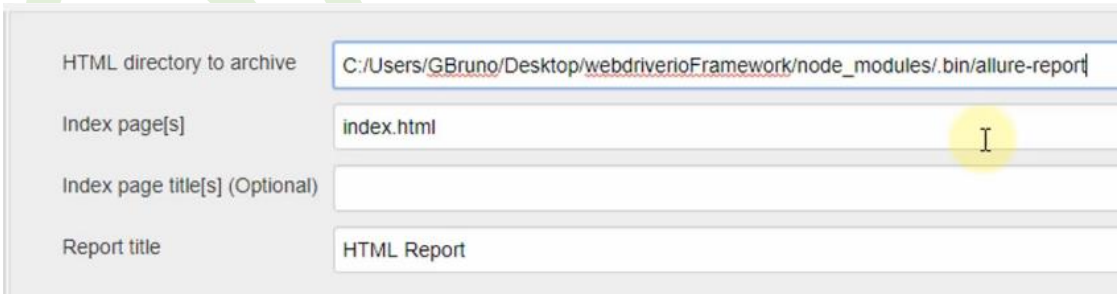
- Configure the "HTML Reports" plugin within Jenkins to generate an Allure Report based upon the most recent build.
- Modify Jenkins restrictions so that all data populates within the generated report.

Instructions:

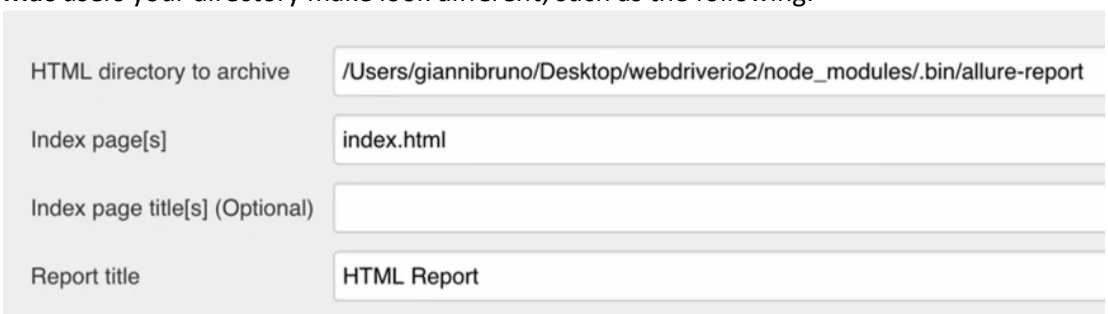
1. Access the current Jenkins job (Project Webdriverio Framework)
2. Click on the "Configure" button.
3. Navigate to the "Build" section and click on "Add post build action" then select the following option (Publish HTML reports)



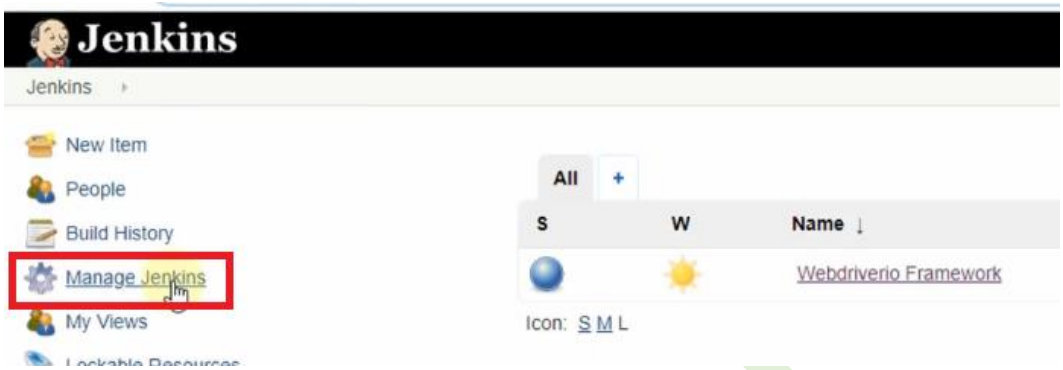
4. Now add the following information, make sure you add the location of your "node_modules/.bin/allure-report" folder (**Windows** example):



5. **Mac** users your directory make look different, such as the following:



6. Now click “Save” to apply all changes.
7. Now we will need to edit the permission of Jenkins in order to show all reporting data.
8. Click on “Manage Jenkins”:

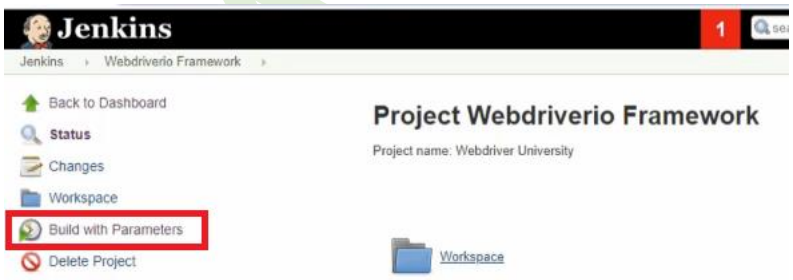


9. Then click on “Script console”:

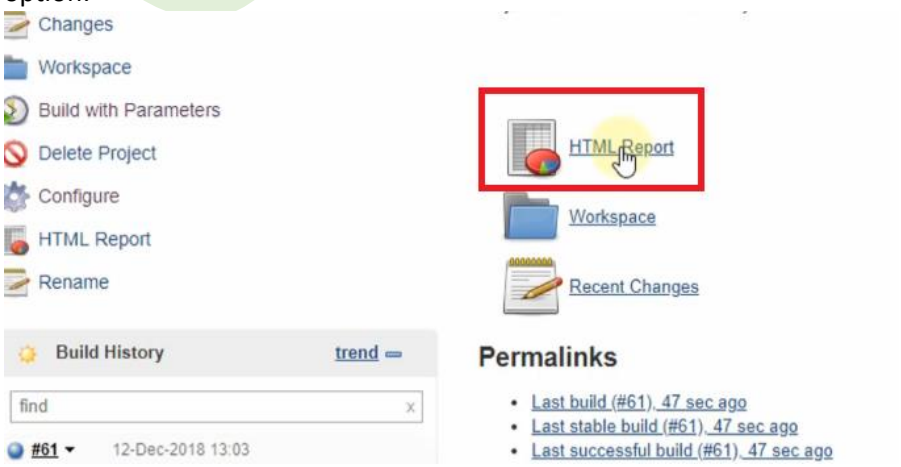
10. Within the script console text field add the following commands:

```
System.setProperty("hudson.model.DirectoryBrowserSupport.CSP", "default-src 'self'; script-src 'self'
'unsafe-inline' 'unsafe-eval'; style-src 'self' 'unsafe-inline';")

System.setProperty("jenkins.model.DirectoryBrowserSupport.CSP", "default-src 'self'; script-src 'self'
'unsafe-inline' 'unsafe-eval'; style-src 'self' 'unsafe-inline';")

System.setProperty("hudson.model.DirectoryBrowserSupport.CSP", "")
```

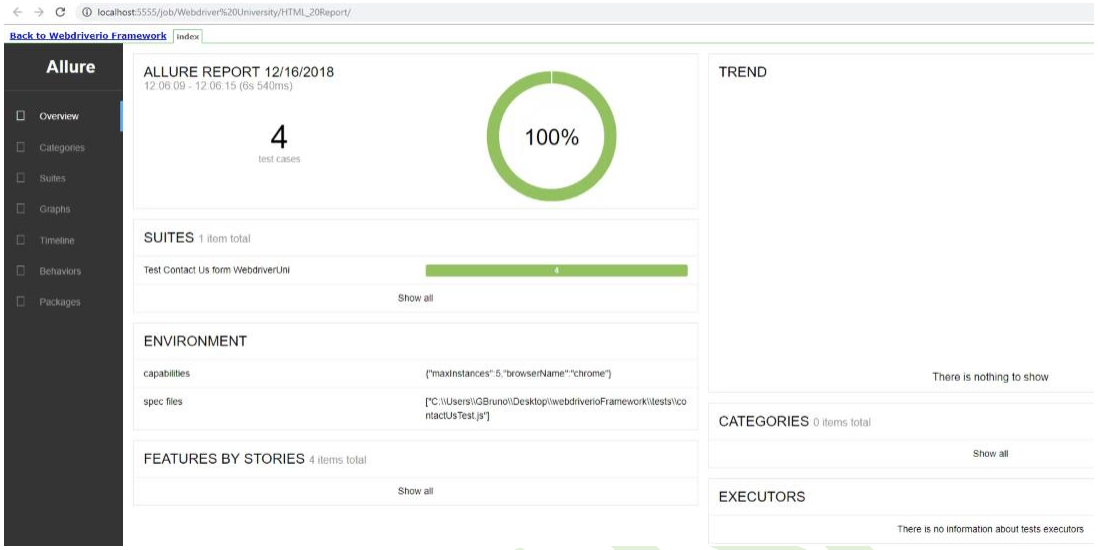
11. Then click on the “Run” button to trigger the scripts.
12. Now it’s time to test the report functionality, access the Jenkins job and trigger a new build:



13. Once the test and build completes, you should then see a “HTML Report” option, click on the option:



14. Now you will be directed to a new page (URL) which contains a report with all details of the most recent test run (Occasionally you may need to clear your browser cache, log back into Jenkins and then attempt to open the report again if you are unable to see all details of the report even after applying the previous permission settings):



15. **PLEASE NOTE**: You can only view a report of the ***most recent*** build execution via Jenkins; this is why we added additional logic within the previous steps to counter this limitation.

Lecture 107 - Generating Allure Reports within Jenkins – Part 4

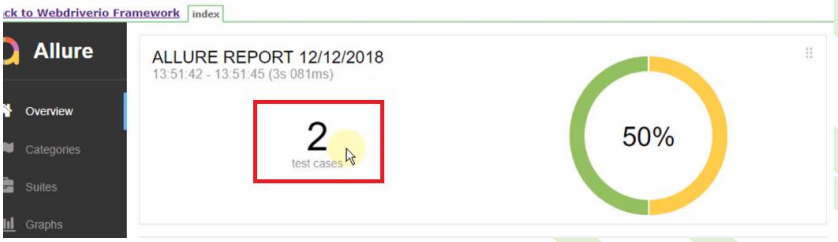
In this lecture we will be making further modifications to the "contactUsTest" to optimise the results / data recorded within the test reports. We will also inspect the data captured within the reports.

Key Points:

- Learn how to effectively structure our automation tests to optimise the data generated within the Allure reports.
- Reviews generated reports post-test / build execution.

Instructions:

1. Trigger a new build targeting the "contactUsTest" and let the tests / build complete, you will see that the total number of test cases listed within the Allure report doesn't add up to the number of tests housed within the test itself.
2. As you can see below the Allure report lists only two tests:



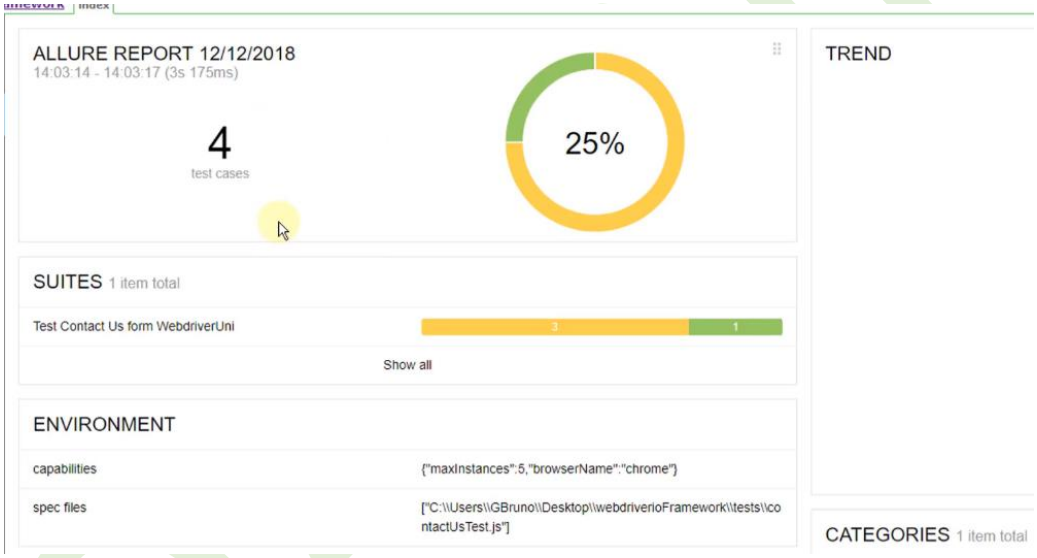
3. If we inspect our "contactUsTest" you can see that the test contains four test cases:

```
describe('Test Contact Us form WebdriverUni', function() {
  it('Should be able to submit a successful submission via contact us form', function(done) {
    ContactUs_Page.submitAllInformationViaContactUsForm('joe', 'Blogs', 'joe_blogs123@outlook.com', 'How are you?');
    });


  it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
    ContactUs_Page.setFirstName('Mike');
    ContactUs_Page.setLastName('Woods');
    ContactUs_Page.setEmailAddress('mike_woods@mail.com');
    ContactUs_Page.clickSubmitButton();
    ContactUs_Page.confirmUnsuccessfulSubmission();
    });


  it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
    ContactUs_Page.setFirstName('Sarah');
    ContactUs_Page.setEmailAddress('sarah_woods@mail.com');
    ContactUs_Page.clickSubmitButton();
    ContactUs_Page.confirmUnsuccessfulSubmission();
    });

  it('Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
    ContactUs_Page.setLastName('Jomes');
    ContactUs_Page.setEmailAddress('sarah_Jomes@mail.com');
    ContactUs_Page.clickSubmitButton();
    ContactUs_Page.confirmUnsuccessfulSubmission();
    });
});
```

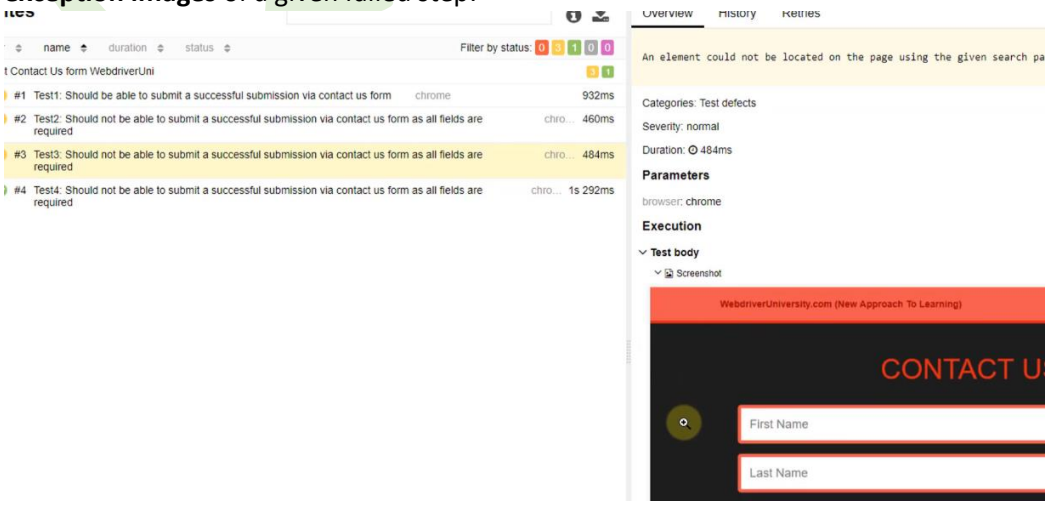
4. To ensure Allure reports reflects the correct number of tests, we need to make sure that each "it" block **description** is **unique** between tests.
5. Make the following changes to the "contactUsTest":

```
describe('Test Contact Us form WebdriverUni', function() {
  it('Test1: Should be able to submit a successful submission via contact us form', function(done) {
    ContactUs_Page.submitAllInformationViaContactUsForm('joe', 'Blogs', 'joe_blogs123@outlook.com', 'How are you?');
    });

  it('Test2: Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
    ContactUs_Page.setFirstName('Mike');
    ContactUs_Page.setLastName('Woods');
    ContactUs_Page.setEmailAddress('mike_woods@mail.com');
    ContactUs_Page.clickSubmitButton();
    ContactUs_Page.confirmUnsuccessfulSubmission();
    });

  it('Test3: Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
    ContactUs_Page.setFirstName('Sarah');
    ContactUs_Page.setEmailAddress('sarah_woods@mail.com');
    ContactUs_Page.clickSubmitButton();
    ContactUs_Page.confirmUnsuccessfulSubmission();
    });

  it('Test4: Should not be able to submit a successful submission via contact us form as all fields are required', function(done) {
    ContactUs_Page.setLastName('Jomes');
    ContactUs_Page.setEmailAddress('sarah_Jomes@mail.com');
    ContactUs_Page.clickSubmitButton();
    ContactUs_Page.confirmUnsuccessfulSubmission();
    });
});
```

6. As you can see from the image above, each “it” test description is unique.
7. Now trigger a new build via Jenkins, targeting the “contactUsTest”, let the build and tests complete and open the latest Allure Report:



8. Now you can see that the correct numbers of tests are listed within the report.
9. If you drill into the step(s) details you can even view additional information such as **exception images** of a given failed step:



Lecture 108 - Generating Allure Reports within Jenkins – Part 5

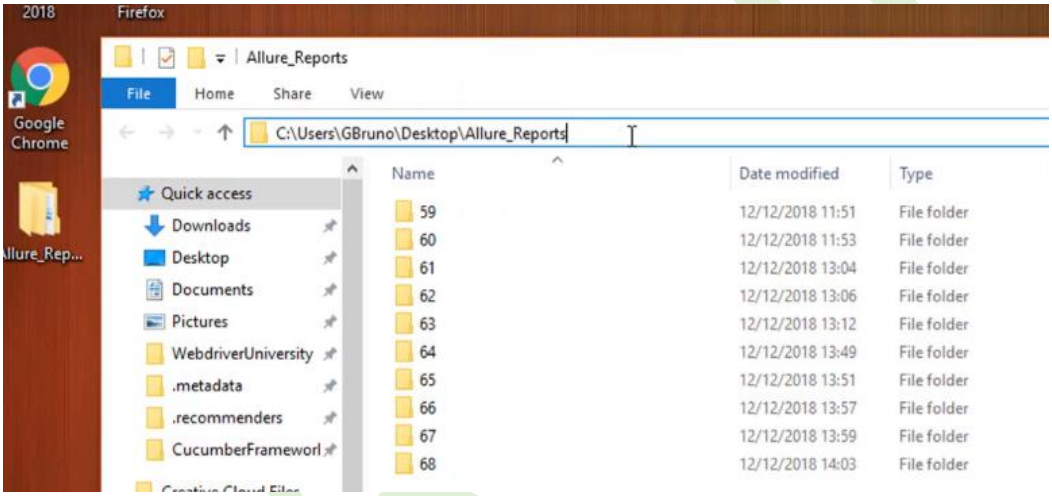
So, you're probably thinking to yourself great I've reached the end of the course and I can only view reports of the most recent test run (Build execution)! In previous lectures we added logic to back up all tests results locally, now we will look at how we can generate test reports based on previous test suite executions (Jenkins builds).

Key Points:

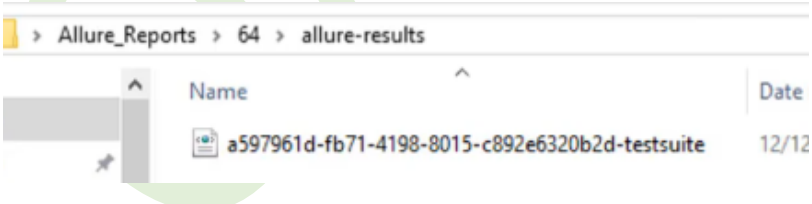
- Understand the reasons behind backing up all test results locally.
- Learn how we can generate Allure reports based on test runs (Build executions).

Instructions:

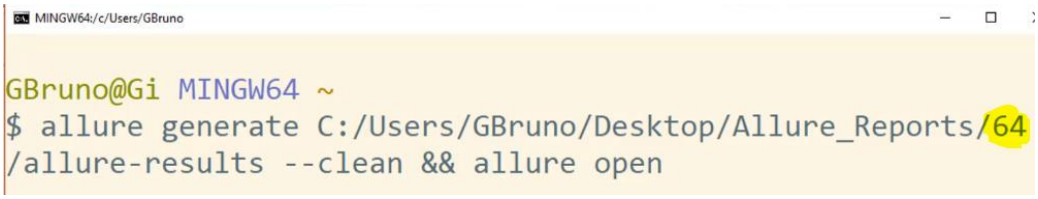
1. If you have implemented the logic from the previous steps correctly, you should see all previous test results saved locally to the location of your desired choice.
2. As you can see below my desktop contains the folder "Allure Reports" (Left hand side), inside that folder you will see subfolders named according to the relevant build number:



3. Now if you open a specific subfolder (For example: 64) you will see that the folder contains an Allure xml file (May include images if there were any test exceptions):



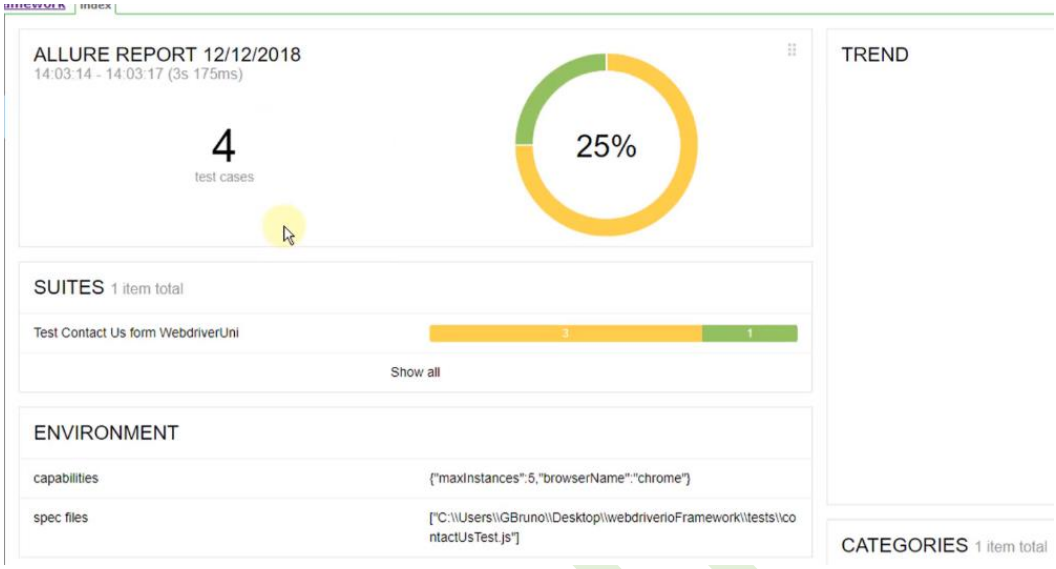
4. Now run the following command to target one of the Allure build xml files using Git Bash:



5. Wait until you see the following message to validate the generating of the report was successful.

```
GBruno@Gi MINGW64 ~
$ allure generate C:/Users/GBruno/Desktop/Allure_Reports/64
/allure-results --clean && allure open
Report successfully generated to allure-report
```

6. Now you should see a new browser tab open which contains the Allure report and test data relating to the Jenkins build; in my case Build 64:



7. Now you are able to generate Allure reports based on all tests / builds which were previously triggered via Jenkins.