Lecture 1 – Introduction to PlayWright and its features

Why use PlayWright?

Reliable!

* Playwright has an “auto-wait” function. This is for waiting for elements to load etc.
* The list of how this interacts with elements is here: <https://playwright.dev/docs/actionability>
* Unlike other tools eg. Selenium Playwright will automatically wait for the elements to be visible and stable etc before even using them.
* This prevents having to write specific code methods to do this and helps avoid flaky tests.

Cross browser compatibility

* Playwright will support testing with ALL major browsers ie. Chrome/Firefox/Safari/Opera/Edge

MultiPlatform

* Playwright works with Windows/MAC/Linux out of the box.
* Supports device emulation for Android or iOS.

Multi-Lingual

* You can create code in many programming languages ie. Java, Javascript, Python, C#

This course is specifically designed for Javascript and Typescript, this is because they work on Node.JS platforms.

Lecture 3 - Install Node.js & Visual Studio CODE for setting up Playwright Environment

* Node.JS – This is the runtime environment for executing Javascript, it is the equivalent of the Java virtual machine.
* Install Node package, based on the machine used, then update the path.
* In windows this just involves adding this line C:\Program Files\nodejs

A screenshot of a computer

Description automatically generated

Lecture 6 - Create npm Project and install Playwright dependencies for testing

* Start by creating a folder in windows and then using open directory in VSCode.
* By running the initialize command (**npm init playwright**) a new Javascript Project will be created, and the file structure will be put in place for the project.
* The PlayWright tool will also be installed and setup per the answers given.
* Playwright.config.js – this is the test runner file.
* Package.Json – this is the project packages file (like eg. POM.XML). All node projects have one of these.
* Node Modules directory – this is where all the JARS (equivalents) are stored for the project.

A screen shot of a computer program

Description automatically generated

A screen shot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

Lecture 7 - Importance of Playwright Test Annotation and async await understanding

* The requires line below is needed to access the “test” part of the module (and allow tests to actually run).
* This “test” keyword is then able to be used for all the tests.
* Because Javascript is asynchronous you need to be able to handle the ordering of the steps (otherwise they would just execute in a random order).
* The “await” keyword is used to help get around this issue.
* You need to add the async keyword to the method signature, otherwise the “await” keyword will be ignored.
* If you don’t have a name for the method then it is an anonymous function. In this case you can use the “=>”(fat pipe) to indicate it is a function and not write the keyword.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

Lecture 8 – Browser Context and Page Fixtures

* Browser is a global fixture which is available for any test annotation (comes from the @playwright/test package).
* You pass the browser object to your methods so that you can access the browser inside your tests.
* You have to wrap the browser in curly braces, to tell JS that this a playwright fixture.
* The browser context relates to eg the settings which are saved when you open Chrome (you might have your own links, plugins etc).
* Browser.newContext() opens the new context but you can tell the browser if you want to open it with some fixed properties.
* Without adding anything the new browser context is the same as opening a new incognito instance.
* A good usage of this is to log in to a site and then capture the login token, meaning you won’t have to login again and again.
* Playwright.config.js (test runner) should contain the browser which will be opened by the method.
* The page fixture will open up the browser on the specified page with no additional settings (such as cookies).

Lecture 9 – Playwright.config.js

* This file is created with a new project.
* testDir – tell the test runner where the tests are.
* Timeout – how long the test should wait before reporting a failure.
* Expect – This is for assertion timeouts.
* Reporter – What format do you want to output the test results in.
* Use – this contains the information about your test setup eg. Logs, browser
* Browsername – webkit is the safari engine, chromium is the chrome engine.
* This webkit is basically the same as Safari, so if your tests are working with webkit, they will be working on safari, and you can check this on WINDOWS.
* By default Playwright will run tests in HEADLESS mode.
* You cannot run individual tests, you have to run them in the folder.
* “npx playwright test” – execute playwright in the node\_modules directory and then run the test runner (playwright.config.js) and the tests inside the configured directory.
* The NPX keyword means that you don’t have to explicitly write the path to the playwright executable.
* To run with an opened browser you need to pass the parameter header – “npx playwright test –headed”
* You can use “test.only” to run one specific test.

Lecture 10 – Running in multiple browsers

* You can get all the assertions from here - <https://playwright.dev/docs/test-assertions>
* Expect is a Playwright assertion, which comes out of the box. This means that you don’t need to import another test framework for testing.
* You need to add “expect” to the import statement – const {test,expect} = require ('@playwright/test');
* You generally need to add the “await keyword” to every line in Playwright, to prevent unexpected results.
* To prevent having to enter hyphen hyphen headless on each command you can just add it to the “use” section of the test runner file.

Lecture 11 – Locators supported and typing into elements

* Playwright supports CSS or XPATH out of the box.
* CSS is the suggested locator strategy.
* Type and fill are the same method, for typing text into the box. Type is now deprecated, so just use fill.

Lecture 12. Extracting the text from browser and inserting valid expect assertions in test

* You do not need to have explicit waits in Playwright, it has the intelligence to wait for itself for the element to show up on the DOM.
* The wait will just adhere to the maximum wait time for the tests which is currently set in the configuration.
* This is a MAJOR improvement over Selenium, as this is the main pain point for Selenium.

Lecture 13. How to work with locators which extract multiple webelements in page

* Using password.fill(“”) can be used to clear a text box.
* The error message below will be shown if multiple elements are returned.
* There are two ways to get an element then, the **first** method and the **nth** method.

A screen shot of a computer code

Description automatically generated

Lecture 14. Understanding how wait mechanism works if list of elements are returned

* Using the allTextContents() method on a list of elements will return the text contents of each in an array.
* The allTextContents() method DOES NOT HAVE waiting capability. This is because it returns an array and the array could be empty, or have 100 results but Playwright will not know.
* The textContent method DOES have waiting capability, so you can put this in front of it ie. Get the first/last elements textcontent, which would ensure there is content on the page before getting the list.

Lecture 15. Techniques to wait dynamically for new page in Service based applications

* Most websites now use microservices to retrieve products via the API and then display them on the page.
* In Playwright we can make the automation wait until all network calls to services have been completed and the state is then idle. (this call is discouraged in the documentation as it can be flaky on some browsers).
* As an alternative to this you can directly put a waitFor on a specific element, this will ONLY wait when ONE specific element is targeted.
* You can specify one js file by adding the path to it on the test command: npx playwright test .\tests\ClientApp.spec.js

Lecture 17. Handling static Select dropdown options with Playwright

* There are multiple types of dropdown list. The first of which is a SELECT dropdown, in this case the options are already prepopulated, and the HTML tag will be SELECT.
* Select option is based on the “value” of the option element.
* Page.pause() will stop the test at completion, so that you can see the result.
* Pausing will also open the playwright inspector, which allows you to debug the test code.

Select dropdown code:

A screen shot of a computer code

Description automatically generated

Lecture 18. Selecting radio buttons, Checkboxes and implement expect assertions

* Checkboxes and radio buttons can just be clicked like buttons
* There are the isChecked and toBeChecked methods to check their status
* There is no “toBeUnchecked” but you can use “toBeFalsy” as an assertion instead.

Lecture 19. Using async await with Assertions and understand validating the attributes

* The await keyword is only required in the scope where it is performing an action.
* await expect(blinkingText).toHaveAttribute("class","blinkingText");
* This assertion can be used to check an attribute where the functionality it uses can only be verified based on the presence/lack of an attribute.

Lecture 20. Handling Child windows & Tabs with Playwright by switching browser context

* This is around handling child windows/new tabs when they are opened by the site (eg. Clicking on a link).
* When the link is opened on the same page Playwright can easily handle it, but when it is opened in a new page it is more complicated to handle.
* You have to give pw knowledge that a new page will be opened before you click the link.
* You need to use the browser fixture, so that you can have multiple pages, using a context for each page.
* You use the waitforevent method to detect when a new page has been opened, so you MUST put the check for the event BEFORE clicking the button, otherwise the event will already have taken place.
* The awaits which are used to handle the asycnhoronous nature of Javascript operate with PROMISES.
* A Promise can be either PENDING, REJECTED OR FULLFILLED
* Pending – step is currently being executed.
* Rejected – step has failed, usually because there is a missing object etc.
* Fulfilled – the step has completed and returned whatever the step is expecting.
* You cant put the wait after because the event has already been fulfilled before the step is executed and you cant put the wait before because it will not move to the next line until the promise is fulfilled.
* To get around this the two events need to be linked together as one, you do this using the Promise.all method.
* You don’t need an await in the method as BOTH promises need to be fulfilled within the method before the method will complete.
* Promise.all will iterate again and again until all promises are fulfilled.
* If any of the steps are rejected then the method will fail.
* You can add the new page to a variable, such as below and then once you use this variable it will have knowledge of the new page and you can perform operations on it.

Example of Promise.all

* Promise.all returns an array of promises.
* The first step will promise to return a newPage
* The second step will not return anything as it is just a click.

A screen shot of a computer code

Description automatically generated

Lecture 21. What is Playwright Inspector? And how to debug the playwright script

* The inspector is for debugging tests.
* You can look at each step and see in detail what it is actually doing.
* You can also then execute the steps one by one (you need to add the pause at the start of the script).
* You can also use the inspector as a tool to get locators.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

Lecture 22. Codegen tool to record & Playback with generated automation script

* You can get Playwright to automatically generate test code by using the “record” function.
* npx playwright codegen <https://rahulshettyacademy.com/loginpagePractise/>

A screenshot of a computer

Description automatically generated

Lecture 23. Detailed view of Test Traces, HTML reports, logs & Screenshots for test results

* To generate a screenshot on every automation step you need to add “screenshot: ‘on” in the test runner file.
* To generate detailed logs on every automation step you need to add “trace:’on’” in the test runner file.
* The report data will be stored in the playwright-report and test-results folder inside the project.
* The trace data shows you what the screen looks like before and after every step.
* The test-results folder contains a trace.zip file for each test which has run.
* Setting the trace value to “retain-on-failure” will ensure that the trace.zip file is only created when a test fails.

A screenshot of a computer

Description automatically generated

Lecture 25. Write the Script to dynamically find the product to buy from list of products

* Any automation tool for web support locator “chaining”, this means that you can navigate the DOM from wherever you currently are in it.
* The scope will be within the current locator.
* You can search for an element based only on the text which is inside it.
* You need to be careful when navigating pages and looking for elements if the both have elements which are the same eg. Text=ipod. You can use tags here to make sure that you don’t select the same element from the previous page (assuming they don’t have the same tag).
* There are two ways to get elements with text in Playwright. You can use CSS to get any element with the text ie. locator("text= Add To Cart"), OR you can use a Playright feature for getting text within elements of a certain type ie. isVisible().
* IsVisible is not one of the functions which Playwright automatically supports waiting for and would fail if the locator is not visible on the page immediately, in this case you can use WaitFor() to wait for an appropriate element. You don’t have to wait for the specific element itself, just another element which must be there when you want to proceed is sufficient.

A screenshot of a computer

Description automatically generated

Lecture 27. Handling Auto suggestive dropdown options with playwright – Example

* Suggestive dropdown options only work as you actually type into the field, if you use the “fill” function then it is pasted into the field and will not trigger the options.
* The “pressSequentially” option instead of fill will type the characters incrementally.

Lecture 32. Understand how GetByLabel & Playwright UI Runner works with an example

* This relates to additional locators which are available in Playwright only.
* You can use getByLabel() to grab any label based on the text within it and if it is next to a button then you can use click()/check() on it.
* You can use npx playwright test –ui to open the test runner and run tests from here!
* You can see the screenshots of each step here also (as in trace).

A screen shot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

Lecture 33. Filtering elements with GetByRole,GetByText and perform chaining methods in step

* The placeholder attribute can be used to getByPlaceholder, this can be used to enter text.
* The getByRole method lets you grab by the attribute role or by the tag eg. Button

A screen shot of a computer

Description automatically generated

Lecture 35. Strategy on handling Calendars automation using Playwright

* Example calendar - <https://rahulshettyacademy.com/seleniumPractise/#/offers>
* If an element has the “hidden” tag then it will not be detected by Playwright

Lecture 38. How to validate if element is hidden, displayed mode with Expect assertions

* This is around sites where an element is hidden and then displayed again based on the functionality.
* You have the standard browser back/forward options in pw.
* You have assertions tobevisible and tobehidden

Lecture 39. How to automate Java/JavaScript Alert popups with Playwright

* Opening a popup will block the page, you cannot proceed unless the popup has been handled.
* Javascript popups are not using HTML code, so you cannot use the standard locators to handle them.
* You have to tell pw in advance how to handle the JS event which you want.
* Page.on listens for JS events and then performs the provided action, it DOES NOT MATTER what line the code is on, it will still be detected.

Lecture 40. How to handle & Automate frames with Playwright – Example

* A frame is an additional page which is displayed on the current page.
* If pw is having trouble identifying an element then it is probably within another frame, pw can only see the original page.
* If the tag iframe or frameset appears in the HTML then it means there is a frame on the page.
* You need a new page object for using the frame.
* When getting a locator, if you add :visible to the end pw will only return elements which are visible on the page.
* Use text.split to break up a string using a delimiter.

Lecture 41. understanding the importance of API integration calls for Web tests

* As a webpage is working it makes calls to the related APIs and then stores details in the “local storage” In the browser.
* The information can then be reused within the application eg. Session Token.
* This is really useful because you can use pw to make the api call to get a token and you don’t need to log in the UI, which will take longer.

Lecture 42. Playwright request method to make API calls and grab response – Example

* It is also better to use APIs for sign in scenarios because they are more stable, no flickering tests due to the browser.
* To make API requests you need to add “request” to the import statement.
* The API context must contain the HTTP method which you want to use for the call.
* The data object is just the request body which will be pushed in.
* If you use the .json() method on the response then you can grab the json fields easily.
* Playwright can execute any Javascript and so can be used to insert the token into the browser session.
* AddInitScript takes two arguments, a function and a value.
* Using pw to insert the token into the session means you can skip all the log in steps, which should save time (especially if there are many tests).
* Pw CAN be used as an API test tool but really it is only intended to allow the API calls which are needed for the UI automation.

Lecture 44 Place order API to create order and bypass the flow in UI with mix of web/API

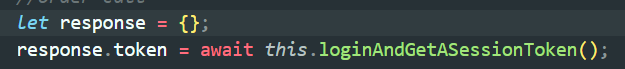
* The session token may not always be stored in the same place in the browser but it should always be possible to get the token.
* If we just want to check that orders are displayed correctly on the web then we don’t need to actually run through the steps in the ui to create one, we can just insert one via the api and then check that it is displayed.
* The industry is moving in this direction, as it is much faster and helps minimize the amount of the actual web UI steps which are needed.

Lecture 46. Refactor API calls from utils folder and isolate from Web test logic

* module.exports = {APIUtilities} – This makes the class globally available to other classes.

Lecture 47. Part 2 - Refactor API calls from utils folder and isolate from Web test logic

* As the createOrder method needs to generate a token and also return an order id a new response js object is created, this response contains both the token and order id. This response can then just be returned to the calling test class and both values will be available in the tests.
* Creating a JavaScript object like below means that other variables can just be assigned to it as properties of the object.



Lecture 48. How to save session storage using Playwright and inject into new Browser context

* This is to help in situations where logging in is a complex process and you don’t really want to have to run a log in for each of your tests.
* All the information when you sign in is logged in the browser and you can capture all this information. (Local storage, Sesssion storage, Cookies etc)
* How this works is that you log in once and the “Storage State” is output into a JSON file.
* All subsequent tests can then just “Inject” this storage state (of the browser) into the fresh browser.
* This works on the “BROWSER” context level (if you 100s of pages open the information would be stored about all of them).
* If you want to skip the log in steps for multiple tests you really need to decide whether to use this method or the API call method. The API method is very fast compared to logging in with the UI but it will still need to be executed every time a test runs, whereas the state saving solution only logs in one time and then uses the details for ALL the other tests.

A screenshot of a computer

Description automatically generated

Lecture 49. How to debug the API steps in script using Visual code debugging

* The pw inspector only works on the UI part of your tests, it will skip any API tests and open the debugger at the first web step.
* The hyphen hyphen debug line opens the inspector when running the test.
* To debug both api and ui you need to add a “scripts” section in the package.json.
* You then need to add at least one debug breakpoint
* To access the debugger in visual studio, you need to open the package.json file, do ctrl shift p and then select “Debug: debug npm script”
* This will run the debugger to the breakpoint

A screenshot of a computer program

Description automatically generated

Lecture 51. How to intercept Network response calls with Playwright route method

* The webui makes API microservices calls to populate the data eg. Products
* With pw you can intercept the network calls and then mock the data to test some scenarios which could be difficult to test otherwise.

Lecture 52. Understand the playwright route method and its parameters in intercepting – demo

* The route command is the one which tells pw that we want to route requests to one endpoint in a different way.
* The first argument will always be the URL to mock and then the function is HOW to go about this.
* We convert the page into the request and then fetch the response from the route which is specified.
* If you don’t have values in the route then it will just send the original details.

Lecture 54. How to intercept Network request calls with Playwright - Example demo

* Hackers will attempt to intercept requests eg. The request to get an order by id, and then use it to view other peoples orders.
* This is a security hole and should not be allowed.
* Pw lets you capture the outgoing request and modify them, so that you can check this scenario.
* You need to ensure pw is ready before you click the button to generate the API call
* You can add the \* to an intercept URL, so that it will match to ANY order id.
* The continue method on route will intercept the URL and say that “whenever we get this URL pattern, continue with these details instead”
* The continue method is able to capture the request based on many variables eg. URL/headers/body, what is in the curly braces is what will be captured.
* Ther curly braces are like JSON, so you can specify many different parameters.
* In VS, right click and “format document” sorts the code into pretty mode.

Lecture 56. How to abort the Network calls with Playwright – Examples

* You can use this to test what happens if the backend apis are not responding to requests.
* Any call which is made by the browser can be blocked eg. Png file
* \*\*/\* refers to any request
* You can use the curly braces to abort multiple file extensions - await page.route("\*\*/\*.{jpg,gif,jpeg}",route=>route.abort());
* You can use the page listener to output information about your tests
* page.on("request",request=>console.log(request.url()));
* page.on("response",response=>console.log(response.url(),response.status()));
* A screenshot of a computer program

  Description automatically generated

Lecture 57. Capture Screenshots with Playwright on page & partial Element level – Demo

* You can take a screenshot of the whole page or of one specific element
* You just need to provide the path for the screenshot to be downloaded to.

Lecture 58. What is visual testing & How to perform it using Playwright

* Visual testing is the process of setting a baseline image as the “correct” screenshot and then using pw to compare the layout with the baseline image every time it runs.
* If a screenshot with the configured name is not present the first time the test runs, then it will create one for you.

Lecture 60. Introduction to excelJS node module and setting up JS Project

* Npm install exceljs

Lecture 61. Traversing rows and columns of excel worksheet with excelJS library

* 62. Build Util functions to read and update excel file strategically
* 63. How to get and update the data from excel based on filter search criteria
* 64. Strategy to handle download & uploading files using Playwright
* 65. End to end example for excel edits and upload with assertions using Playwright
* 66. code download
* Skipping these ones – don’t think it will be useful for our suite to use Excel.

Lecture 67. What is page object pattern & Importance of its implementation

* The constructor is the place where the locators should be defined in a POM
* The page needs to be imported to the test class - const {LoginPage} = require("../Pages/LoginPage");

Lecture 69. Creating Page objects and action methods for end to end Script - Part 2

* The Page Object Manager class lets you create all your pages in one place, then you can just import this class and you will have all of them.