Playwright

**Why Playwright:**

It is a open source tool for web browser automation from Microsoft. It has several advantages compared to selenium like it has in-build auto wait concept so we need not to write separate method for wait (but the timeout is based on the config parameter we gave globally), then it is best for modern web application (i..e, built using angular js), also supports cross browser testing.

**Features:**

A screenshot of a computer

Description automatically generated

Tracing and Debugging – Capture screenshot automatically with before and after steps, records video of execution, logs all info, displays piece of code from script that corresponds to the screenshot which enables the user to identify the failure more easily etc.,

Browser context management – allows the user to capture the cookies and pass it to the subsequent steps to bypass the repetitive steps like we can bypass login for all test script by capturing the cookies and inject into to the next test scripts.

Codegen tool – it is used to auto generate the code based on the user action on browser.

A screenshot of a computer

Description automatically generated

Official Doc - <https://playwright.dev/docs/videos>. Node can be downloaded from google and install it. Then add the variable path and check in command prompt using node --version.

Similarly install visual studio code from google which is the best editor for JS and TS language.

**Practise URL:**

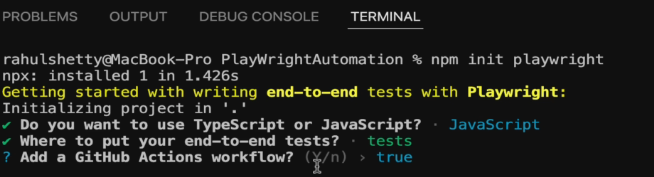
<https://rahulshettyacademy.com/loginpagePractise/>

<https://rahulshettyacademy.com/client/>

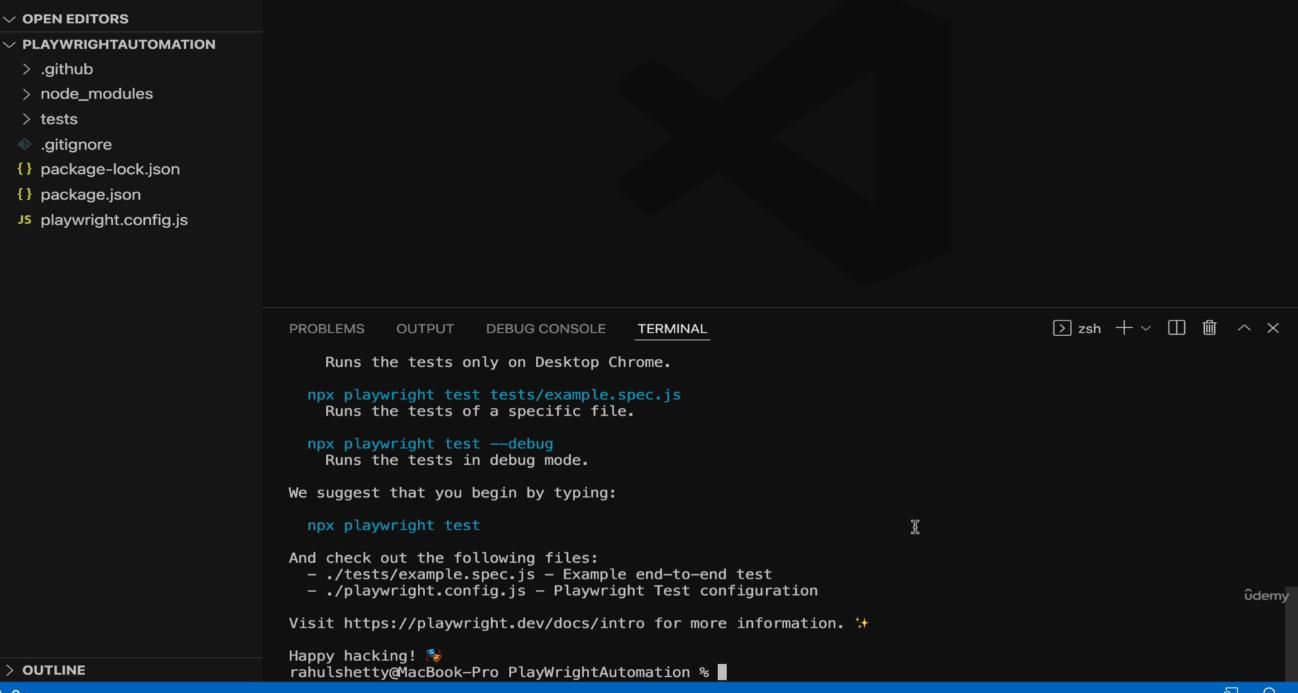
**How to install playwright:**

Open Visual Studio code and open an empty folder from the IDE from desired location say D drive. Then open the terminal using the encircled cross mark at the left bottom of the IDE.

Command 🡪 npm init playwright and then accept the default for remaining questions.



After some time, you will see the playwright framework template on the empty folder you have opened. It is not just the dependency but capable of building the framework template in order to work with playwright.



Each folder / file on the framework template we see is important, but the most important is “Tests” folder where we create the spec file (like class in Java) and “playwright.config.js” is a configuration file which acts as test runner for our project.

A screenshot of a computer

Description automatically generated

**How to automate using playwright:**

Create a new file under “Tests” folder and save it with “.spec.js” extension. To automate first we need to import one annotation from playwright module (which is indirectly from comes from node\_modules) named @playwright/test (it comes from the jars in “node\_modules” folder). It should be assigned to a variable as shown below.

**How to write the basic test template:**

Const {**test**} = require(‘@playwright/test’)

**test**(‘test case name’, function(){

})

Line 2 and 3 is the template for test case. Anything user writes between these two lines, is treated as test case. It takes two parameter 1. String (to mention the TC name) and 2. function

**Things to remember:**

Since JS is asynchronous, we need to add a keyword ‘await’ before every line so that the subsequent step will wait till the previous step execution is completed. Since we add the keyword ‘await’ before every line, we need to declare the function as async. It is like a rule, when you use await keyword then it is must to make the function as async. Also, when we don’t have a name for the function, it is called anonymous function. As per the recent update, when a fn is anonymous then it we can write it as ()=>. Considering these points, the above function can be written as

test(‘test case name’, async ()=>{

})

**Fixtures** – like global variables once you declare them as a parameter in TC function it is available throughout the test script. Fixtures are available by default to your test cases and that is the reason we are not declaring them at the global level like we did for const {test}. The fixtures will be enclosed in curly braces so that my TC understands it as playwright fixtures.

It is approx. of 4 types. Let us see the first 2.

Browser – used to open a browser with “browser.netContext()” which means an empty browser instance with no cookies, proxies and plugins. We need to assign it to a variable say “const browser”.

Page – on the opened browser, open a new page using the browser variable say browser.newpage() and assign it to a variable say const page = browser.newpage()

**To open a browser** with a specified URL, we need to start using fixtures (like global variables) inside the function brackets as shown below.

To open a new context (used to pass cookies, plugins or proxy) or an instance of a browser, write the below line and assign it to a variable.

**const context =await browser.newContext();**

**Note**: newContext() 🡪 will open an fresh browser instance with no cookies and plugins but new Context method will take parameter which will be used to pass cookies, plugins etc.

To open an empty page, write the below line on the variable created for context.

**const page = await context.newPage();** //this actually opens a tab / page to enter URL etc

**Now to open a browser with URL**

await page.goto(“https://www.google.com”);

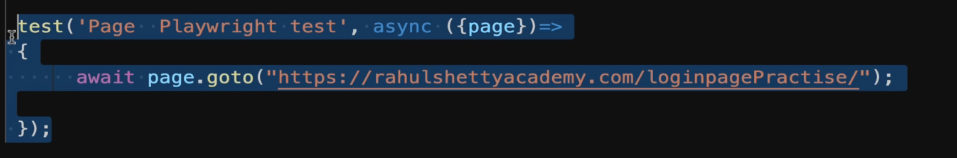
await page.goBack(); //to click browser back button

await page.goForward(); //to click browser forward button



Since we are going to open a new instance with no plugins, proxies and cookies with an empty page, the same line can be rewritten as shown below i.e., playwright helps us to achieve the above with one line.

**Remember** – browser.newContext() is required only when you want to open your browser with default cookies, plugins etc. if it is going to be a plain page, then just add “page” fixtures which makes playwright to understand that user wanted to open a plain browser and remove those 2 lines.



**Test Runner file:**

In playwright the test runner file is nothing but “playwright.config.js” i.e. the trigger point of your test cases. Update the file to have the following keys 1. Const config ={}. This is the place where we need to mention the default wait time, assertion wait time, which browser we want to use, how to take screen shot etc.,

Remember – we can’t directly triggers the test cases but only through config file.

**How to run the test scripts using command from terminal**

First, we need to prepare the “playwright.config.js” file because that is the starting point / triggering point for all our tests. Use the “config” object to set the directory from which file we need to run the test using “testDir” key. Then set the timeout key.

Also, the “use” object is the key things to mention which browser we want to use, to run in headless or head mode. Playwright by default runs the test scripts in headless mode.

**Note**: use “browsername : chromium” to run in chrome and “browsername : webkit” to run in safari (default for playwright).

**Command**

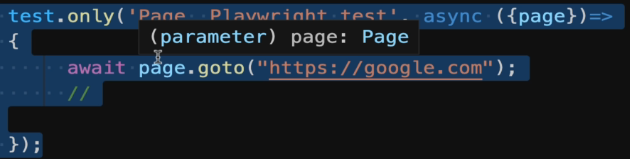
**npx playwright test** // used to run the test scripts in headless mode. npx denotes the path of playwright module in node\_modules dependencies folder.

**npx playwright test --headed** // used to run the test scripts in headed mode.

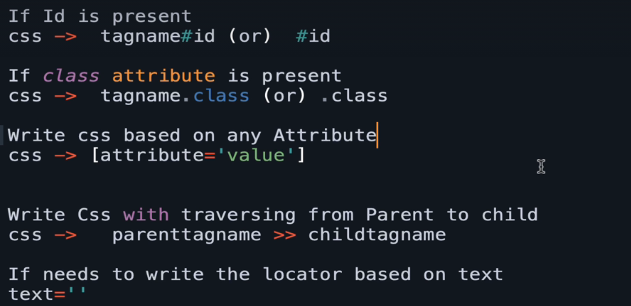
**npx playwright test tests/testscriptfilename.spec.js** //to run specific test file when test folder has more than one test file.

**Note**: to avoid using headed in the command, we can mention that under the “use” object in config file where we specify the headless and set it to false.

By default, playwright will run all the test cases which are started with “test” under the test case file. If user wants to run a specific test case inside the test case file, then make the test case read as “test.only”. “Test.only” is applicable at test file level. When more than one test file present under test directory, then change the command as above.



**How to find elements with playwright / Locator techniques:**



CSS and Xpath are the commonly used techniques and CSS will be used more.

**Using id** - > Tagname#id or #id //first one will give you unique value than second one.

Page.locator(“#name”).fill(“”);

**Using class** -> Tagname.class or .class

Page.locator(“.name”).fill(“”);

**Using attribute** - > [attributeName = ‘value of attribute’] or [attributeName\* = ‘value’]

Page.locator(“[attributeName =’value’]”);

**Note:** when locator tech is [attributeName = ‘value’], but the value of the attribute is too long then use \* after the attribute name and enter partial value of the attribute value i.e., [attributeName\* = ‘partial value of the attribute text’].

**Using Tag and attribute** - > tagname[attributename = ‘value of attribute’]

Page.locator(“tagName[attributeName=’value’]”);

**Using text**:

Method 1 = await page.locator(“**text = Zara Coat 4**”).textContent();

Method 2 = await page.locator(“**h3:has-text(‘Zara Coat 4’)**”) //will be used when you want to combine with text and tagname.

**E.g.,** to traverse from **parent to child** using CSS

A screenshot of a website

Description automatically generated

In this example if user wants to grab text of the first product, then start your CSS from parent node. Here when hovers the mouse over class=card-body, the entire tile gets selected for shoe. So, the locator should be like

Await page.locator(“.card-body b”).first().textContent();

// where “.card-body” is class attribute value and “b” is the tagname of shoe.

A computer screen shot of a program code

Description automatically generated

**Interaction with web elements:**

**To type into a text box use fill()**

await page.locator(“#id”).**fill(“Ragav”);** // type and fill method was used earlier but type is deprecated now.

**To perform a click operation use click()**

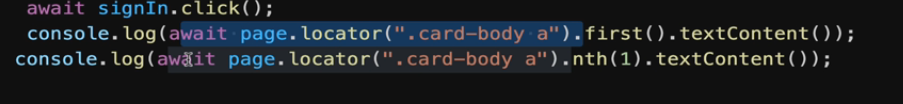
await page.locator(“#id”).**click();**

**To get the text from web page use textContent()**

await page.locator(“[attribute=’value’]”).**textContent();**

**To get all text from web page when element locator technique founds more than 1 match then use allTextContents()**

await page.locator([attribute=’value’]).**all**T**extContents();**



nth(0) - > give first element, nth(3) 🡪 4th element.

First() and last() 🡪 give first and last element respectively.

**Difference between textContent() and allTextContent():**

textContent() 🡪 gets the text from the webpage for the given web element and will wait by default. Return type is single web element of type string.

allTextContents() 🡪 gets all text from the webpage related to the given web element but won’t wait by default. Return type is multiple web element of type array of strings.

Refer 🡪 <https://playwright.dev/docs/actionability> to understand the default wait time for different methods provided by playwright.

**Tips:**

1. Let us say user needs to erase the entered value in the field username and type a new value. In that case, you can fill method with blank arguments, so that the existing value will get cleared off.
2. Also, the locator tech can be stored to a variable, and we can call that variable name further in our scripts.

const username = page.locator(“#username”);

username.fill(“”); // to clear the pre-existing text

username.fill("Ragav”);

**Validation:**

To assert anything, use “expect” function. For example, to assert a title use the following line.

**To get the web page Title:**

await expect(page).toHaveTitle(“Google”)

**Note**: The expect keyword must be added under the first line of code. It is like importing package in Java.



**To validate the text that we get from web page is matching partially:**

await expect(page.locator([style\*=’block’])).toContainText(“Incorrect”);

**Note**: here we are checking whether error message contains the given word. Upon witnessing the partial match make the test case pass.ui

**To check a radio button and checkbox is checked:**

await expect(page.locator(“”)).toBeChecked();

**To check a radio button is unchecked:**  no separate method is available. Alternatively

await page.locator(“#term”).uncheck(); //to uncheck the checkbox

await expect(page.locator(“#term”).isChecked()).toBeFalsy(); // the highlighted part returns false because we unchecked it already and checking whether it returns false. Similarly there is something called toBeTruthy().

**To check an attribute has the given value: (this is useful when something is blinking on the page)**

await expect(page.locator(“”)).toHaveAttribute(“attributeName”,”attributeValue”);

**To check whether an element is visible or hidden:**

await expect(page.locator(“”)).toBeVisible();

await expect(page.locator(“”)).toBeHidden();

**Wait Mechanism for lazy loading:**

1. **Scenario - Get web page content in ecommerce where there are more results for search:**

In general, when you try to fetch the content from the web page, it will take some time (in seconds) because to load the entire web content related to your search. During that time, if user uses allTextContent(), then it will return empty list / array without waiting as per their official docs. To handle lazy loading

await page.waitForLoadState(‘networkidle’); //not working for few and discouraged

await page.locator(“.card-body b”).allTextContents();

**Note**: <https://playwright.dev/docs/actionability> Refer the link to see playwright will give auto wait time. If any method is not there then we need to add wait mechanism manually using waitFor(). Also, when multiple elements are found and if we use waitFor(), it will throw error. So use First() or last() along with waitFor() when there are multiple elements returned for the given locator.

**Alternate solution**

When the above method is not working, add this line before you use allTextContents(), so that it will wait for the particular element and then user can use allTextContents().

await page.locator(“.card-body b”).first().waitFor(); // we need to call first() or last() before waitfor() to make this line to work.

1. **Hard stop the code like Thread.sleep in java**

await page.pause(); // this will stop the code execution.

When user uses pause() method, then playwright opens an playwright inspector in a separate screen. Unless user clicks the play button to resume the execution, the script won’t go the next step.

**Handling drop-down, radio button and check box:**

Drop down with select tags or static dropdown.

<https://rahulshettyacademy.com/loginpagePractise/>

Identify the element with locator and use selectOption(“pass the value of value attribute”) method to select a value.

A close-up of text

Description automatically generated

await page.locator(“select.form-control”).selectOption(“consult”) //to select consultant.

**Radio button:**

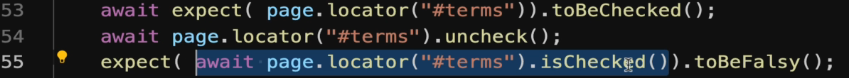
await page.locator(“#user”).click(); //to click a radio button.

**Checkboxes:**

await page.locator(“#user”).click(); // to check a checkbox

await page.locator(“#user”).unCheck(); // to uncheck a checkbox

**Usage of await:**



In general, await will be used as the first word in each line but look at the difference between line numbers 53 and 55 “await” is used in different places.

So, when we are going to perform an action then await should be the first word like. In line number 53 and 54 where we are checking something and performing an action whereas in line number 55, await is used inside the expect statement because there we are checking is it checked or not.

**Window and Tab handling:**

Switching between windows / tabs is quite challenging with playwright. It is because JS is asynchronous, so we use await in every line.

To get the new window id, we need to make use of the context that we create at the first line.

const context = await browser.newContext(); //use this variable to listen to new window.

To check whether the new window is opened, then.

Context.waitForEvent(‘page’);

This line must be written after we click the link which opens a new page, but the problem is when we use await on the before line where we perform the click operation on the link, this line will not execute and when this line is into focus, the event has already happened i.e., page has already opened. So let write the line above performing the click operation.

Now the second issue is, if we write the “waitforevent” line before clicking action, then the test case will get skipped because we are using await.

So to overcome this issue, we need to run these two lines in parallel i.e., asynchronously. To do that use promise.all() method whose return type is list.

Const [newPage] = await Promise.all(

[

context.waitForEvent(‘page’),

page.locator(“documentlinklocator”).click(),

]);

Now we need to start using the new page handle, then.

const text = newPage.locator(“”).textContent();

Note: //create const[newpage, newpage1, newpage2]based on number of new pages opened.

**How to get a specific text from a sentence:**

String = Please email us at mentor@rahulshettyacademy.com with below template to receive response.

Now in this case I want to grab rahulshettyacademy.com then use split method.

const textfromNewPage = await newPage.locator(".red").textContent();

const newText = textfromNewPage.split(“@”);

newText[1].split(“ “)[0];

**Playwright Inspector:**

It is an inbuilt engine / functionality provided by playwright to debug the script step by step. This gets popped up when user add page.pause() or run the test case with the command.

npx playwright test /tests/testscriptname --debug

A screenshot of a computer

Description automatically generated

Use the resume button to complete the execution in regular mode or step over button to see the test execution step by step.

Also, use the Explore option at the bottom of the playwright to create a locator / verify the locator used is right.

**Codegen:**

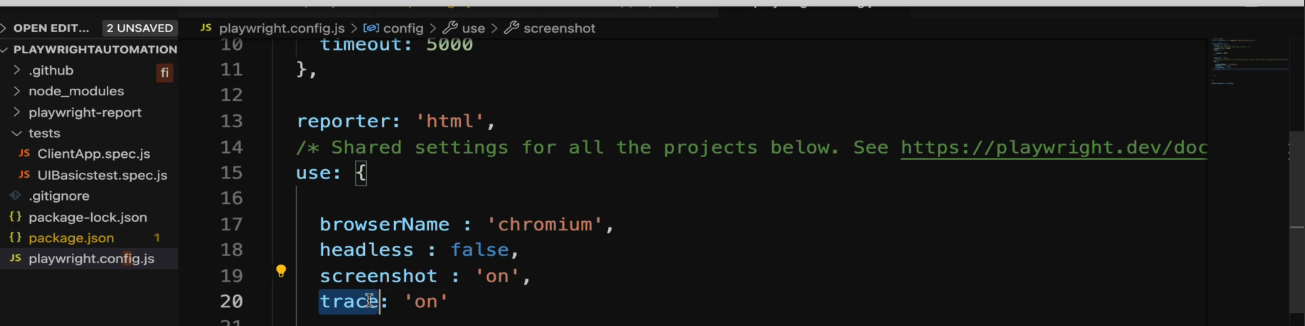
Playwright helps the user to generate code automatically based on the user’s action performed manually on a browser. To achieve this, use the command as below.

npx playwright codegen <http://www.google.com>

As soon as user presses enter after the typing command, then playwright will open the “Playwright Inspector” and start creating code for each manual action user’s perform on browser.

Note: The browser was mentioned in the config file.

**How to take screenshot and log traces:**



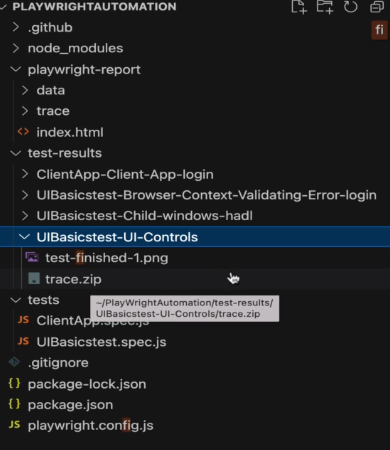
Add the attribute and set the value of “screenshot” to “on” and “trace” to “on” to capture all test cases or “only-on-failure”(screenshot) and “retain-on-failure”(trace) to capture only failed test case under “use” key in config file. These two lines will be responsible to capture the screen shots as well as capture all log information.

**To view the reports in html format,** refresh the project, then on the project explorer, open the “playwright-report” folder and open the index.html which will show step wise details along with number of lines of code it took to perform that action. Copy the path of index.html and paste it in browser. Also, the test case of each report will have screenshots, traces too. Click and view traces.

A screenshot of a computer

Description automatically generated

To view the traces separately, expand test-results folder (each test case will be captured as one sub folder) and download the zip folder(just a click on the sub-folder) and put the zipped folder “trace.playwright.dev” link. It will show how the screen looks like before and after performing each step. User can also upload the file by navigating to the project root directory and open the zip file.



**To check whether the element is visible in the page:**

await page.locator(“h3:has-text[‘Zara Coat 4’]”).isVisible();

**Note**: when isVisisble() is used in code, then we need to explicitly add a wait mechanism using first(). waitFor(), since playwright does not provide auto-wait as per their docs. You can use just waitfor() when there is only one match

**Locator Techniques:**

**GetByLabel** -> when a field has the tag name ‘label”, then we can locate the element using the method getByLabel. This is preferred when we need to perform click or select a dropdown.

await page.getByLabel(“enter the field name”).click(); //both click() and check() works same

await page.getByLabel(“enter the field name”).check();

await page.getByLabel(“enter the field name”).selectOption(“Female”);

**GetByPlaceholder 🡪** This is applicable if the field type is of textbox, or the field type has the attribute “placeholder” in it.

await page.getByPlaceholder(“Password”).fill(“Ragav”);

**GetByRole 🡪** When you want to click a button or a hyperlink, then we can use GetByRole method. We can apply this either on button tag or class name has value as “btn”.

await page.getByRole(“button”,{name:”typeTheTextOnButton”}).click();

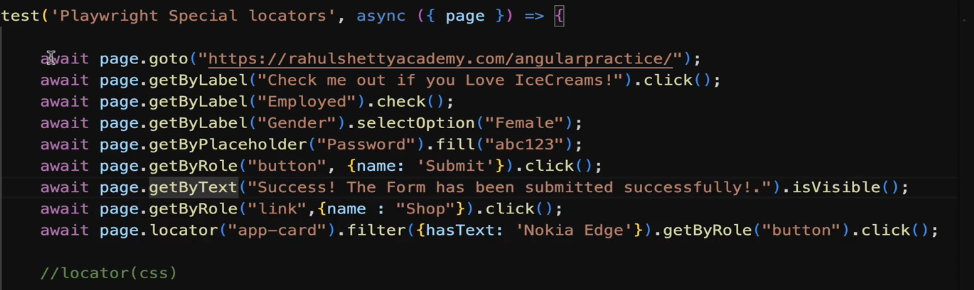
await page.getByRole(“link”,{name:”typeTheName/TextOnLink”}).click();

**Scenario:**

When user wants to click a particular item from multiple matching result for a given locator technique, then use filter method followed by getByRole.

await page.locator(“app-card”).filter(hasText : {“Blackberry”}).getByRole(“button”).click();

**Note**: here in getByRole we didn’t pass the second argument has we already applied filter on the locator which gives us only one matching result hence no need to add the second arguments. The second arguments is helpful when there are more than one matching result.



**To open a Ui Test runner:**

In general, we run the test case using the command npx playwright test /test/scriptname.spec.js. Instead, we can open a UI editor or test runner to run specific or all test cases.

Syntax to open the Ui test runner 🡪 **npx playwright test --UI**.

**How to handle alert:**

When an app displays a pop up, we must have a line in our script to handle it by keeps watching for the event using “on” method.

Page.on(‘dialog’, dialog => dialog.accept()); // to accept the dialog

Page.on(‘dialog’, dialog => dialog.dismiss()); // to cancel the dialog

**How to handle mouse hover action:**

await page.locator(“”).hover();

**How to handle frames:**

First find the frames using name or id and assign it to a variable. Then perform the next action like interacting with a web element inside the frames using the frame variable created in the first step.

const frames = page.frameLocator(“nameofTheframe); //await is not since it will not do any action

frames.locator(“”).click();

**How to handle invisible elements:**

There are sometimes for a given locator, more than one match would be found but app highlights only one element because the other element is invisible. To handle this scenario,

await page.locator(“#id:visible”).click();

**How to work with API and Web UI:**

When we interact with web pages, we use the fixture page but to work with API we need to add “request” at the very first line.

const {**test**, **request**} = require(‘@playwright/test’)

**test** (‘api’, async()=>{

const payload = {userEmail:’raghavdce@gmail.com’, userPassword:’Rahulshetty@123’}

const apiContext = await **request**.newContext();

const loginResponse = await apiContext.post(“https://rahulshettyacademy.com/api/ecom/auth/login”,

{

data:payload

}

})

expect(loginResonse.ok()).toBeTruthy(); //validating the status code is success

const jsonResponse = await loginResponse.json(); //getting the response in json format

const token = jsonResponse.token; //parsing the json to get the token from it.

**How to pass the token to web app to avoid login screen:**

test(“webapi”, async({page})=>{

page.addInitScript(value => { //addinitscript method takes 2args, 1. Fn and 2. value

window.localStorage.setItem(‘token’,value);

},token);

await page.goto(“https://rahulshettyacademy.com/api/ecom/auth/login”);

})

**How to create utils files:**

Now in the above case we can move the function login to utils folder and call the method inside the test case to avoid code duplication.

Create a folder under “Test” named “Utils” and create a file “APIUtils.JS”. Add the following class and method to handle the code duplication.

Class APIUtils{

constructor(**apiContext**, **loginPayload**){

*this.apiContext* = **apiContext**;

*this.loginPayload* = **loginPayload**

}

async getToken(){

const login = await *this.apiContext*.post(“URL”,

{

data:*this.loginPayLoad*

}

Const loginJson = await login.json()

Const token = loginJson.token;

Return token;

}}

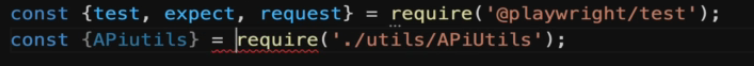
A screen shot of a computer program

Description automatically generated

Note: Pass the argument orderPayload in createOrder().

Module.export = {APIUtils};

In the actual test class where we are going to create order using API, import the utils class and call the getToken method by creating the object of APIUtils class.



Const apiutils = new APIUtils(apiContext,loginPayload) //parameters are defined in the test class

Const createOrder = createOrder(orderPayload);

A screen shot of a computer code

Description automatically generated

**How to take screenshot:**

Page.screenshot({page:’google.png’}) //to take screenshot at page level

Page.locator(“#id”).screenshot({page:’google.png’}) //element level screenshot

**How to compare 2 screenshots:**

Expect(await page.screenshot()).toMatchSnapshot(‘google.png’)

**How to inject session storage into a browser:**

After login to the web application use storageState method to capture all the session storage like token. This method takes the argument to store the path of the json file which has the details about storage session of a browser. We need to use this method on context level and not at page level.

Then in the test case, have the following steps 1. Launch the url and 2. Create a new page using the webcontext.newPage() where webcontext is defined globally.

await context.storageState({path: ‘state.json’})

A screen shot of a computer code

Description automatically generated

**How to read from excel:**

To work with excel and json file, user need to install exceljs dependency in the project level.

**Syntax to install exceljs dependency:** npm install exceljs

Once user install the exceljs, import the package at the first line using.

const excelJS = require(‘exceljs’);

async function readExcel(){

to get the workbook, we need to create an object.

const workbook = new excelJS.workbook();

await workbook.xlsx.readFile(‘path of the file”);

const worksheet = workbook.getWorksheet(“sheet1”) //to get the sheet

to iterate through the row

worksheet.eachrow( (row, rowNumber) =>{ //row holds all row and rownumber holds current row

row.eachcell( (cell, colNumber) =>{

console.log(cell.value);

})

})

}

readExcel();

**To get the co-ordinates of the particular cell value:**

const excelJS = require(‘exceljs’);

async function readExcel(){

to get the workbook, we need to create an object.

const workbook = new excelJS.workbook();

await workbook.xlsx.readFile(‘path of the file”);

const worksheet = workbook.getWorksheet(“sheet1”) //to get the sheet

to iterate through the row

worksheet.eachrow( (row, rowNumber) =>{ //row holds all row and rownumber holds current row

row.eachcell( (cell, colNumber) =>{

//write the logic here

If(cell.value === “V Ragavendran”){

Console.log(rowNumber);

Console.log(colNumber);

}

})

})

}

readExcel();

**To update the value of a particular cell:**

const excelJS = require(‘exceljs’);

async function readExcel(){

let output = (row:-1, column:-1}; //creating an obj to get the row and col value out of for loop

to get the workbook, we need to create an object.

const workbook = new excelJS.workbook();

await workbook.xlsx.readFile(‘path of the file”);

const worksheet = workbook.getWorksheet(“sheet1”) //to get the sheet

to iterate through the row

worksheet.eachrow( (row, rowNumber) =>{ //row holds all row and rownumber holds current row

row.eachcell( (cell, colNumber) =>{

//write the logic here

If(cell.value === “V Ragavendran”){

Output.row = rowNumber;

Output.column = colNumber;

}

})

})

Const cell = worksheet.getCell(outpu.row, output.column); // variable cell holds row and col num

Cell.value = “Rumble” //setting the value for the variable cell

Workbook.xlsx.writeFile(“path of the excel”); //writing it to the excel.

}

readExcel();

**To upload a file:**

page.locator(“#id”).setInputFiles(“path of the file”);

**Note**: This will work only when the web element has the attribute type=”file”

**To download a file:**

Downloading a file and performing an action on it is not straightforward. Since we need to put some manual wait using wait for event method before clicking download button.

Const download = page.waitForEvent(‘download');

await page.locator(‘#id”).click(); //clicking download button

await download;

**To find a particular value from a web table:**

1. First, we need to go the row we are interested in using the locator and assign it to variable.

Page.goto(<https://rahulshettyacademy.com/upload-download-test/index.html>);

Const searchText = page.getByText(“Mango”);

const desiredvalue = page.getByRole("row").filter({has:textvalue});

expect(desiredvalue.locator("#cell-4-undefined")).toContainText()

A screen shot of a computer code

Description automatically generated

**Page Object Model**



Create a separate folder named PageObject and have the separate class with .js files for each page with constructor that takes “page” as parameter and define all the locators inside it so that it can get initialized when user creates a new object for the class inside the test case. Then the last statement of the class should be module.exports = {classname}. Also, define separate method inside the class to perform the desired operation.

A screen shot of a computer program

Description automatically generated

Now to use the class inside the test case, first import the package using a variable

Const {LoginPage} = require(‘../PageObject/LoginPage’} where pageobject is the foldername and loginpage is the class name.

To use it inside the test case, create an object

Const **loginpage** = new LoginPage(page) //**loginpage** is a variable name and page is fixture used in test class.

To call the methods defined inside the class file, use the variable

loginPage.loginToApp(email,password); //email and password are variable with values in the test case.

Similarly use the same for all pages and have them imported.

**Code Optimization technique:**

1. Assume if we are automating an app which has more pages, then the test case will have more number of imports and more number of object creation. To fix this, create a class named POManager, which should have all page object inside the constructor. Then create only one import in test class and call the respective methods from POmanager class and each method will return the respective objects

A screen shot of a computer program

Description automatically generated

Inside the test class, import the PO manager class and upon creating the object and call the respective methods, in turn it will return the page objects as shown below.

A screen shot of a computer program

Description automatically generated

1. **Test data from external file**

Create a folder named utility and create a file test data as .json extension. Have the test data in json format and to use it inside the test class, add the following line

Const dataset = JSON.parse(JSON.stringify(require(‘../Utility/testdata.json’)

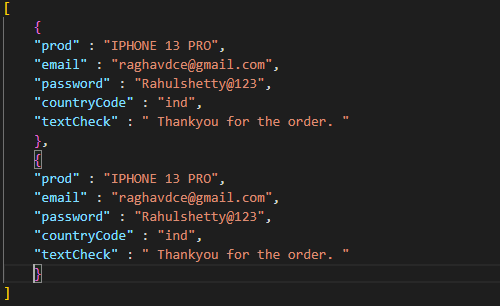
Now wherever user passes the test data, just name is as dataset.variablename or keyname used in json file.

A screenshot of a computer

Description automatically generated

1. **How to run the same test case with different set of test data.**

Modify the json file to array instead of one json set.



Now to give knowledge to test case that it needs to run based on the length of the array, then add a for loop as shown below (like for each loop)

A screen shot of a computer program

Description automatically generated

Also change the dataset to data for eg., dataset.email to data.email. Also, to make the report unique to identify which test data was run change the test case title as

`endToEndTesting for ${data}` //note **`** is in tilt button and it is not single quote.

**How to pass test data from fixtures:**

A computer screen with text

Description automatically generated

A screen shot of a computer screen

Description automatically generated

**How to use custom config file rather than using default config file:**

1. Copy paste the config file named playwright.config.js and rename it to playwright.config1.js
2. While running the test case use the command as npx playwright test /tests/specfilename –config playwright.config1.js

**How to use the same config file but to run the test case in different browsers:**

1. Copy paste the config file named playwright.config.js and rename it to playwright.config1.js
2. While running the test case use the command as npx playwright test /tests/specfilename –config playwright.config1.js
3. Create a new key named projects:[]
4. Inside the array give a name attribute to identify the test case and then paste the “use” attribute as shown below

A screenshot of a computer program

Description automatically generated

1. If user simply runs the test case with command and mention just about the config then the test will run for both chrome and firefox
2. If user wants to run for a desired browser ie., for a particular project (because both firefox and chrome are added as 2 different projects) then use

Npx playwright test /tests/endToEndTestingPO --config playwright.config1.js project=chrome

-or-

Npx playwright test /tests/endToEndTestingPO --config playwright.config1.js project=firefox

**How to run in customize width of the browser:**

Add a new property under use key in config

Viewport : {width:720, height:720}

**How to run in specific mobile models:**

Add a new property under use key in config

…device[‘iphone11’]

**Note**: as soon as you put single quote, playwright will give you the list of available devices if we use this property under browser : webkit //default for safari

To run in Android then

…device[‘nexus5’]

**Note**: this has to be used under browser:chromium

A screenshot of a computer

Description automatically generated

**How to handle ssl certificate error:**

ignoreHTTPSError:true

**How to handle google wants to access your location pop up in browser:**

Permissions:[‘geolocation’]

**To record video of test execution:**

Video : ‘on’ // add this in config file to record the video and it is available in reports.

A screenshot of a computer program

Description automatically generated

Similarly, trace is used to capture logs and screenshot is used to capture screenshots.

**How to rerun the failed test cases:**

In config file, user can add a key under **global parameter** named “retries” with value as 1 as shown below.

Note: This retries parameter should not be added in “use” parameter because if user enters under “use” then it is applicable only for that test case only. The retried test case will be available under “flaky” tab (if it gets passed in 2nd attempt).

A screenshot of a computer program

Description automatically generated

**How to run test files in parallel:**

By default playwright run the test files (.spec.js files and not test cases inside in it) in parallel and the default count is 5. We can control the numbers using the parameter workers under global parameter

workers : 3 //if we set it to 6, then 6 workers will be assigned

**How to run test cases inside test files in parallel:**

We need to add a line at the test case file level at the top of the test name as shown below

Test.describe.configure({mode:’parallel’}) //to run all test case in parallel

A screen shot of a computer code

Description automatically generated

**How to run in sequence:**

By default test cases will run in sequence but it will run all the rest of the test cases even if the previous test case is failed. For eg., you have a scenario wherein user want to try a series of test cases 1.login, 2.validate home page, 3. Order something, 4. Checkout and each part will have it own screen locator and methods.

But for some reason, test case gets failed at home page, then there is no point in running 3 and 4 because obviously those are going to get failed. Instead we can skip them if we mark the serial attribute . It can be verified in report.

Test.describe.configure({mode:’serial’}) //to run in sequence when there are depends on methods

**How to group test cases:**

We might need to run test cases based on tags like smoke, functional and regression etc. Now to achieve this, we need to add @smoke or @web or @API in the test case. For e.g.,

Test(‘@smoke login test case’, async () =>

{

})

Now to run test case related to smoke then the command should be

Npx playwright tests --grep @smoke

**How to generate allure reports:**

Install the dependencies

npm i -D @playwright/test allure-playwright

The following 3 lines are used to generate the allure reports.

1. To create a line report which collects all pass / fail status in non-readable format and stored in allure-results folder

npx playwright test --grep @smoke --reporter=line,allure-playwright

1. To create the report and store it inside a allure-report folder

allure generate ./allure-results --clean //line to create allure-report folder

**Note**: by default VS code runs all the command in powershell. Hence S.No 2 will throw error and won’t allow to create the allure-report folder. To achieve this, change the drop down from powershell to command prompt.

A screenshot of a computer program

Description automatically generated

1. To open the report in web browser

allure open ./allure-report // opens the report in html

**How to create custom scripts to trigger test from package.json:**

Need to add the command and map it to a key under scripts as shown below.

A screen shot of a computer program

Description automatically generated

To run the test, then simply say npm run WebTest

**Jenkins:**

Install Jenkins or run via war file and open Jenkins in localhost:8080 and login.

Create a new free style project.

Add the path of the project under general tab under use custom workspace.

Add all keys under scripts in “package.json” file in “This project is parameterized” option with a name say “Script” (without quotes)

A close-up of a line

Description automatically generated

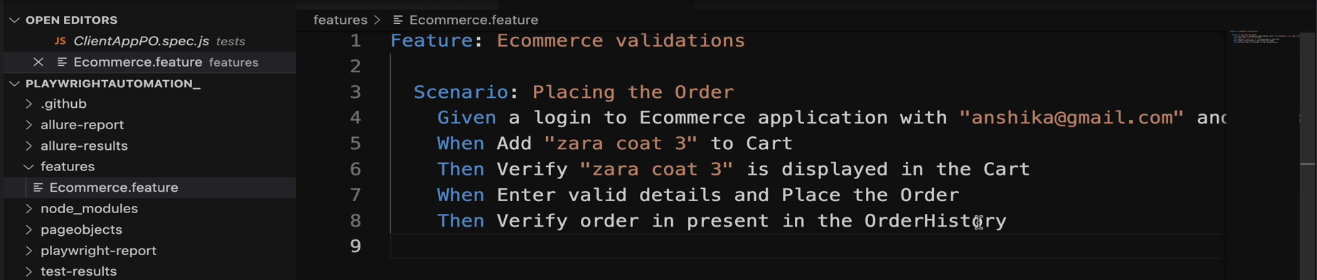
Update trigger build under command prompt with command as “npm run “%Script%”

Save the project and click build with parameter. Select the option from dropdown.

**Playwright with cucumber:**

1. First install the cucumber with the command **npm install @cucumber/cucumber**
2. Install the package from Extensions icon in VS code and search for cucumber.
3. Restart the machine so that the package and the dependencies will get attached to the project.

Create a folder named features and add a file with .feature extension and write the scenario using gherkin language. Anything comes in double quotes are nothing but values which will be passed to the respective steps file (where we will map each step with real code).



**How to add steps for the gherkin lines:**

Create a subfolder named step\_definitions and create a file called steps.js. To add the respective methods for each line written in feature file, we need to run the test case so that it can give us the template what or how to write the template for step files.

To run cucumber files: **npx cucumber-js --exit**

Once the above command is run, the console will list out that functions are not implemented with proper template and user can simply copy paste it inside the steps.js file.

**Note:** We need to import “given, when, then” as the first line. Also, there can be multiple feature files but we can write code in one step\_definition file

A screen shot of a computer screen

Description automatically generated

Now we need to start copying the actual code from end-to-end testing spec file. There are few things that need to be added like page, browser etc.,

First import the POManager package, playwright/test package in the cucumber.

Now to give knowledge about page, browser to cucumber add one more variable in playwright/test package.

const{expect } = require(‘@playwright/test’);

const{playwright} = require(‘@playwright/test’);

const browser = async playwright.chromium.launch();

const context = async browser.newcontext();

const page = context.newPage();

A screen shot of a computer program

Description automatically generated

By default, cucumber will run in headless mode. To run it in head mode, just modify the first line inside login.

Const browser = await playwright.chromium.launch(

{headless:false})

**World Constructor in cucumber:**

In general when we create any new object in the block of code, then the scope of that object lies within the block and can’t be used outside of the block. In this case, we create

Const poManager = new POmanager(Page) inside the login block but how can the rest of the code can access it.

To overcome this, we can make use of world constructor concept. If we declare the same line as

This.poManager = new POManager(page), then this.pomanager is reusable across the entire scenario.

**Wait time:**

In general, all methods like give(), when(), then() will take only 5 seconds to execute. If user wants to give manual wait time, then in the method add an argument as shown below

A computer screen with text on it

Description automatically generated

Every time when user run the cucumber test case, user will see the below error

A screen shot of a computer

Description automatically generated

To avoid this error, create a file at project level and name it as cucumber.js as given in the screen shot and add the line module.exports={default: ‘—publish-quiet’}

**Hooks:**

Hooks are generally to set the before and after condition at scenario level, step level or at suite level.

For e.g., Before can be used to open the browser and afterstep can be used to take screenshots. User need to import the package.

The hooks should be implemented under the subfolder named support with file name as hooks.js

A screen shot of a computer

Description automatically generated

**If user has more than one feature file, then how to run specific feature file.**

Npx cucumber-js features/EcommerceErrorValidation.feature –exit

**Tags:**

Add @Regression or @smoke at the top of the scenario in feature file

To run test cases, npx cucumber-js –tags “@Regression” –exit

Tags can be mixed with hooks by giving the tag name in the before or after hooks so that the preconditions mentioned in before or after hooks are executed only before the scenario related with those tags. The tag name mentioned in both scenario and hooks should be same.

A computer code with red and black text

Description automatically generated

Note: A scenario can have one or more tags. We can use and ( to combine more than tag)or or(to check either or) inside the hooks.

**Scenario Outline:**

It is used to parameterize the test case i.e., the test case can run for multiple test data.

It can be achieved with the use of Examples keyword with headers as shown below.

A screenshot of a computer

Description automatically generated

The test case will run based on the number of rows under examples. We need to carefully pass the column header in place of the data. Add more column based on the number of test data in the test case.

**How to run test case in parallel:**

By default, cucumber will run scenario in parallel and not the features files(limitation with cucumber).

To run all the scenario in one feature file in parallel, then just use the following command.

npx cucumber-js feature/ecommerceValidation.feature --parallel 10 –exit

feature – folder name

ecommercevalidation.feature – feature file name

parallel – attribute helps to run the scenarios in feature file parallel

10 – assume if the feature file has 10 scenarios, then all those 10 will run in parallel

A screenshot of a computer screen

Description automatically generated

**How to generate HTML report:**

npx cucumber-js features/EcommerceValidation.feature --parallel 2 --format html:cucumberreport.html --exit

After the execution is completed, check the root directory for the name cucumberreport.html and copy the path and paste it in browser.

**How to rerun the failed test case using the keyword retry in the command:**

npx cucumber-js –tags “@Regression” –retry 1 --format html:cucumberreport.html --exit

Note: We can even add this a key value pair under scripts under package.json

A computer screen with text on it

Description automatically generated

IT incharge – Joe

Context -

New hardware and software requirement

Connectivity required

Sign-off person –

A close-up of a list of information

Description automatically generated

Angulare – front end – on one server

Middleware – node -on a separate server

POC – will not be available over the internet (end user) hence no public IP is not needed.

Benefits to have 2 diff server – angular where we hosting the web – it will have public ip – available to public – security

What is reverse proxy?

Server for point 2 – is it separate from DB – yes

Jenkins is used as internal tool for Ci/CD

Nodes to connect to which DB or cust – that info will be stored at –

A screenshot of a project

Description automatically generated