Back tick character (`) in JavaScript is used **to define template literals**, which are a convenient way to include expressions and multi-line strings in your JavaScript code.

A placeholder is represented by ${}, with anything within the curly brackets treated as JavaScript and anything outside the brackets treated as a string.

BY default, Playwright runs test cases in parallel. BY default it starts **5 workers/processes**. We can change that in Config file.

Test cases that are within a file are run sequentially. If we want to run them in parallel, then use: **test.describe.configure(mode: {‘parallel});** //put this statement at the top of the test cases file after the import statements.

Grouping the test cases can be done using @**tagname** inside the test case name. When running use: npx playwright test - - grep **tagname**.

We can add scripts in package.json file to run specific test cases. When we do that, we don’t have to run ‘npx playwright test’, instead, we can do **‘npm run nameofthescript’**

"scripts": {

    "test": "echo \"Error: no test specified\" && exit 1",

    "regressionTest": "npx playwright test --grep @Regression",

    "smokeTest": "npx playwright test --grep @Smoke"

  },

In the above script, we can do ‘npm run smokeTest’. It will only run test cases that are tagged as ‘@Smoke’ in the test case name.

CI/CD: Jenkins: needs Java 11 or 17.

**Allure Report:**

Reporting: we can use html report, also **Allure report**. Just download its plugin using:

Installation: npm i -D @playwright/test allure-playwright

DO this: npx playwright test - - grep @tagname --reporter=line, allure-playwright

//we have to run the line report first, line report will be input for Allure report.

After the tests are run, run this command to generate the report:

Generate Allure Report:

allure generate my-allure-results -o allure-report --clean

Open Allure Report:

allure open allure-report

npx playwright test --grep Parameterized\_test\_2 --config playwright.config1.js --reporter=line,allure-playwright

Chropath or SelectorHub extension in chrome.

Playwright suggest using CSS selectors for locators: Do not use Xpath if possible.

Below, tagname is optional if you think the tagname is unique, you don’t have to write it.

If *Id* is present: CSS => tagname#id or #id

If *class* attribute is present: CSS => tagname.class or .class

Write CSS based on any attribute: CSS => [attribute = ‘value’]

Write CSS with traversing from parent to child: CSS => parenttagname >> childtagname

Write CSS locator based on text: CSS => text = ‘ ‘

What kind of wait mechanism to implement depends on the applications.

If it is a **service based application** (network calls are made), then we can use the function to wait until network state is idle.

await this.page.waitForLoadState("networkidle");

This isn’t the only function for the wait. Refer to official website for more functions like the above.

For **non-service based app** where the application gets the data from the server directly:

Use ‘Race condition’:

Await Promise.all (

[

Page.waitForNavigation(), //telling playwright that after you click on signIn, wait for the page to navigate/load.

signIn.click(),

]

);

**How to wait for element if there is no autowait implemented by the playwright.** Eg: autowait is not implemented with **isVisible().** We can use the below method.

*waitFor()* method = wait until items are loaded.

**Drop downs:**

Select Dropdowns: static dropdown, values are already populated.

**Child Window handling**:

1. create a new browser context.
2. Use ***Promise.all***() to make playwright wait for the new page to open.

Const[newPage1, newPage2] = await Promise.all([

Context.waitForEvent(‘page’), //before clicking on a link in new page (newPage1 or newPage2), wait for an event. The event is an opening of new page or pages (newPage1, newPage2 and so on). We can have many new child pages open.

Element.click();

])

Switch back to parent page. Just use ‘page’ object rather than ‘newPage’ object.

**How to open** ***Playwright Inspector***:

await **page.pause();** //can be use at any step in the script.

You can also run the test in **‘- - debug’** mode. This will also open the inspector window.

**Record and playback** – codegen:

NPX playwright ***codegen*** <https://google.com>

**Locator chaining**: start searching locator from the current location. Locator chaining is a way to find an element relative to another one. This means using the current element as the starting point and navigating to its relative parent/child elements.

**Handling popups:**

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

Page.on(‘dialog’, dialog => dialog.dismiss());

**Handling Frames:**

Tagname is iFrame or frameset

Create a new framepage object:

Const framePage = page.frameLocator(“#id”)

Await framePage.locator(“#id”).click();

**How modern day browsers render page to the user:**

For example login process: user sends username and password to the end point url (application url). Username and password are payload. The API responds back with a session token. That token is saved as session cookie (application tab in dev). When a user opens a new tab on browser that is already open, the same session cookie is shared, hence the user does not have to provide the same login info to the server again. In incognito mode, the browser opens with all cookie and cache deleted. I could take the session token from the normal browser and use the same token in the incognito mode to login as long as the session token is not expired. The server authenticates the user with that token.

**Playwright supports API testing:** Weshould be using API calls into our framework to make the framework robust and efficient.

*If you have 50 test cases that needs to go through the login process and perform some action on the homepage, then we can use an API call to login, get the session token, inject that token into the browser and we will land on homepage. Making API call is much faster than a normal login process from the UI. Ask the team developer how they are storing the session information for the login. Based on that, implement the login bypass. It could be stored in the ‘Local Storage’ or Cookies.*

**API Testing:**

Use ‘**beforeAll’** annotation, make the API call to get the token and use that token in the login test.

We can inject the token into the window using the follow methods:

Page.addInitScript(value => {

Window.localStorage.setItem(‘token’, value); //token is the key, value is the token that is passed as 2nd argument below.

}, token );

//now just use normal login

Await page.goto(<https://rahulshettyacademy.com/client/>);

**Browser Context:** This is the browser level state. Cookies and session information are stored at the browser level. I can have multiple tabs (page) on a particular browser context. They all will have the same session information. So, when creating a page, we create a Browser context first, then we create a page on the browser context object**.**

**VVI: You cannot debug the API calls using Playwright Inspector.**

But you can do it using ‘Package.json’ file. Inside this file, add the “tests” object inside the “Scripts’ object. The value of the test object will be the command to run and debug the test.

This way, we can debug from the CLI on the VSCode. Enter Cntrl+Shift+P

We can also use ‘Trace’ functionality to debug. It lets you see all API calls made, all response, etc. The trace file is a zipped folder. Use **Https://www.race.playwright.dev** to open the zipped files.

We can also track every network call and response using Playwright *event listener*: **page.on()**

**Page.on(‘request’, request => console.log(request.url())); //**On every ‘request’ event, take the request and get the url of the request and print it on the console.

**Page.on(‘response’, response => console.log(response.url(), response.status()); //**On every ‘response event, take the response and get the url and status of the response and print it on the console.

This will help in debugging.

**Security Testing:**

**How to intercept network RESPONSE calls with Playwright *ROUTE* method.**

Basically when we get a response from a certain end point, we should intercept the response and alter the body part of the response to our desired fake data and pass it to the browser for it to display on the UI.

**How to intercept network REQUEST calls with Playwright *ROUTE* .continue() method.**

**How to ABORT the network calls with Playwright *ROUTE.abort()*.**

**Route any file with particular extension:**

Page.route(‘\*\*/\*.{jpg, png, jpeg}’, route => route.about()); //to abort any url with jpg, jpeg, png extension. You can do the following to not load .CSS files.

Page.route(‘\*\*/\*.css’, route => route.about()); //to abort any url with jpg, jpeg, png extension.

**‘\*\*/\*’** **:** is regular expression for an url.

**Visual testing with Playwright:**

Basically compares screenshot of an element or page to a screenshot taken previously of the same items. It compares pixel by pixel.

**Page Object Model:**

**Drive data from external JSON files to playwright tests:**

Create a JSON file for each test file. JSON file will contain all test data (eg: username, password, url, etc.) that each test scripts need.

Create JSON files in UTIL folder or TestData folder. A common practice is to name the test data file same as the ‘test script file’ concatenated with ‘TestData’.

When importing JSON file into the test script file: convert JSON into String and then into JS Object:

JSON -> String -> JS Object

const {**dataset**} = JSON.parse(JSON.stringify(require(‘../utils/testPageDataSet’)));

This is one way of doing it. Below, we learn more ways.

**Parameterization in running tests:**

If we need to drive our tests with multiple sets of data, then put the JSON objects inside an array inside the JSON files. Write a FOR loop inside the test scripts OUTSIDE OF TEST FUNCTION to iterate over the array object to drive the test for each test data set. The ‘for loop’ should nest the test function because you want to run the test function for each test data (item) in the array.

JSON file test data:

[

    {

    "username": "hemraj.shahi90@gmail.com",

    "password": "Testing123",

    "productName": "zara coat 3"

},

{

    "username": "anshika@gmail.com",

    "password": "Iamking@000",

    "productName": "adidas original"

}

]

In test script do the following:

For (const data of **dataset)**

**{**

**Test function {}**

**}**

And to have a **unique test case name** for each test data, we can do the following addition to the name of the test case:

const {**dataset**} = JSON.parse(JSON.stringify(require(‘../utils/testPageDataSet’)));

For(const data of dataset)

{

Test(`Client App login for ${data.productName}, async ({page}) =>

{

….test scripts;

}

}

**How to pass test data as fixture by ‘extend’ test annotation behavior:**

We can also pass the data to the test as test fixture but this way can only support one set of test data. In other words, we **cannot parameterize** with this method.

‘page’, ‘browser’ are fixtures the Playwright provides by default. We can also provide our custom made fixtures.

Create a new JS file where we will save our custom test data set (test fixture). Import that file to the test and pass the test fixture to the test function along with the page.

## Creating a fixture[​](https://playwright.dev/docs/test-fixtures#creating-a-fixture): To create your own fixture, use [test.extend()](https://playwright.dev/docs/api/class-test" \l "test-extend) to create a new test object that will include it.