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.

For non-service based app where the applcaton 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**: create a new browser context. 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 a 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. That 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

**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: We** should 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 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. User **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.

/\*\*

Playwright has its own test runner for end-to-end tests called Playwright Test. Playwright Test provides a test function to declare tests, and an expect function

to write assertions.

\*/

npm i -D @playwright/test

/\*\*

Install the supported browsers.

\*/

npx playwright install

// run the test in headless mode

npx playwright test

//Run all the tests

npx playwright test

//Run a single test file

npx playwright test tests/example.spec.js

npx playwright test --headed tests/example.spec.js

//Run a set of test files

npx playwright test tests/todo-page/ tests/landing-page/

//Run tests in a particular browser (config-less mode)

npx playwright test --browser=webkit --headed tests/example.spec.js

//Run tests in all browsers (config-less mode)

npx playwright test --browser=all --headed tests/example.spec.js

//Run in headed browser instead.

npx playwright test --headed

//to open the report in html

npx playwright test --reporter=html --headed

//If you put your configuration file in a different place (other than 'tests' folder), pass it with — config option.

//for some reason, putting config file in 'tests' folder does not work for me. I have to put it inside the main project folder.

npx playwright test --config=tests/playwright.config.js