Quick start with Playwright

By,

Kunal Singh

ks1912

Index

1. What is playwright?

Playwright is a Node.js library for automating web browsers. It allows you to write scripts in JavaScript or TypeScript to simulate user interaction with web pages, such as clicking buttons, filling out forms, and navigating between pages. Playwright supports multiple browsers, including Chromium, Firefox, and WebKit, and provides a unified API for interacting with them. This makes it easy to write cross-browser tests and automate tasks that involve interacting with web pages. Playwright is developed and maintained by Microsoft and is open source. Playwright's architecture is more intricate than this simplified overview, as it involves various communication protocols, concurrency management, and browser-specific intricacies. It supports Chromium (Google Chrome), Firefox, and WebKit (Safari) browsers. Here's a simplified representation of Playwright's architecture:

- **Browser Instances:**

Playwright creates separate browser instances for different browser engines (Chromium, Firefox, WebKit). These browser instances are isolated from each other, and you can launch multiple instances simultaneously.

- **Browser Contexts:**

Each browser instance contains one or more browser contexts. A browser context represents an isolated environment that includes a separate set of cookies, local storage, cache, and other resources.

- **Page Instances:**

Inside a browser context, you can create multiple page instances. A page instance represents a single browser tab or window. Pages are where you interact with web content and perform actions.

- **Network Interception:**

Playwright provides a powerful network interception mechanism that allows you to intercept and modify network requests and responses. This is useful for various testing and mocking scenarios.

- **Event Listeners and Emitters:**

Playwright uses an event-driven architecture. You can attach event listeners to browser instances, pages, and other components to listen for various events like navigation, requests, and more.

- **Selectors and Automation:**

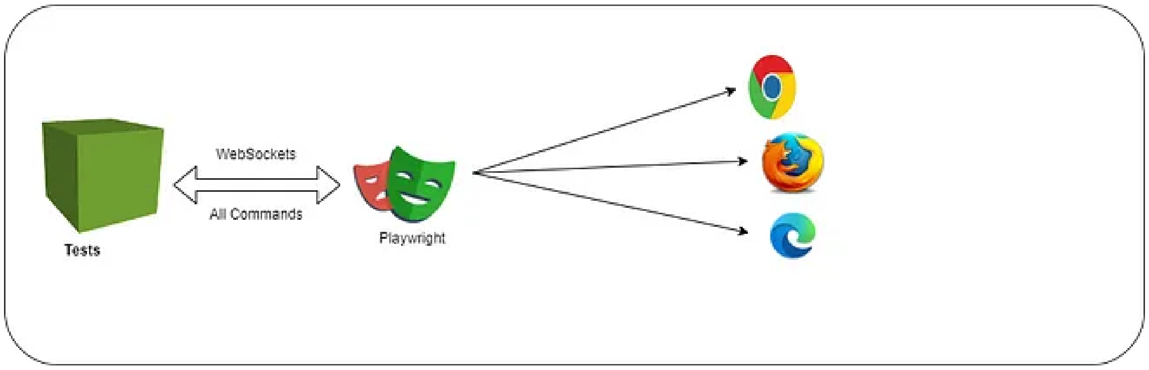
Playwright offers APIs to interact with the page's DOM, select elements using various selectors (CSS, XPath), and automate interactions such as clicking, typing, scrolling, etc.

- **Test Runners:**

While not a core part of Playwright, it can be integrated with various test runners like Mocha, Jest, and others to manage and run tests written using Playwright.

- **Chromium DevTools Protocol:**

Playwright utilizes the DevTools Protocol to communicate with Chromium browsers. This enables advanced control and inspection of browser behavior.



a. What is meant by the Playwright Test?

Playwright is a Node.js library for automating web browsers, while **Playwright/test is a test runner and test framework built on top of Playwrigh**t.

Playwright provides a set of APIs for automating web browsers, such as launching a browser, navigating to a page, interacting with elements on the page, and handling browser events. Playwright also supports multiple browser types, such as Chromium, Firefox, and WebKit.

On the other hand, Playwright/test is a test runner and test framework built on top of Playwright. It provides a set of APIs for writing and running tests using Playwright, including the ability to define test fixtures, group tests into test suites, and run tests in parallel.

Playwright Test was created specifically to accommodate the needs of end-to-end testing. Playwright is a framework for Web Testing and Automation. It allows testing Chromium, Firefox and WebKit with a single API. It is built to enable cross-browser web automation that is evergreen, capable, reliable, and fast. It was built by the team behind Puppeteer, which is a headless testing framework.

Playwright works with some of the most popular programming languages, including JavaScript, Python, Java, and C#.Playwright is a Node. js library to automate Chromium, Firefox, and WebKit with a single API.

2. Getting started with playwright.

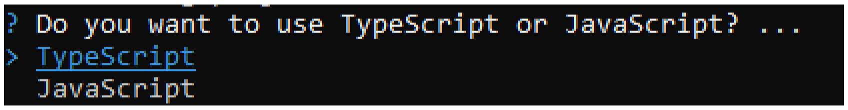
a. Installation of playwright

To get started with Playwright, you can install it using npm or yarn. Here's an example of how to install Playwright using npm:

```
npm init playwright@latest
```

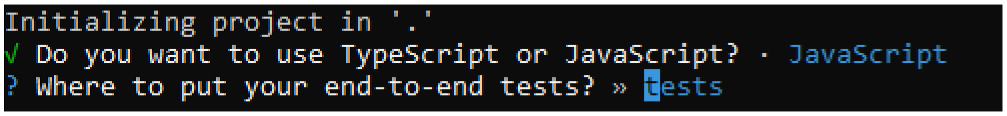
This command will download and install the latest version of Playwright and its dependencies. Once you've installed Playwright, you can start using it in your Node.js projects.

After running the test command you have to choose between the language you will be using typescript or javascript.



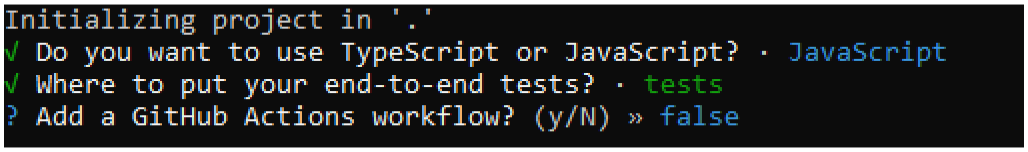
```
? Do you want to use TypeScript or JavaScript? ...
> TypeScript
  JavaScript
```

Once language is selected you have to define a test folder name where your end to end test files will be present. Default name for the folder will be tests. You can press enter if you are satisfied with the default value.



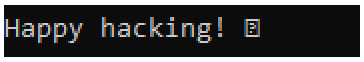
```
Initializing project in '.'
√ Do you want to use TypeScript or JavaScript? · JavaScript
? Where to put your end-to-end tests? » tests
```

After that you will be asked to select a git action workflow. It will be marked as false by default. You can press enter if you are satisfied with the default value.



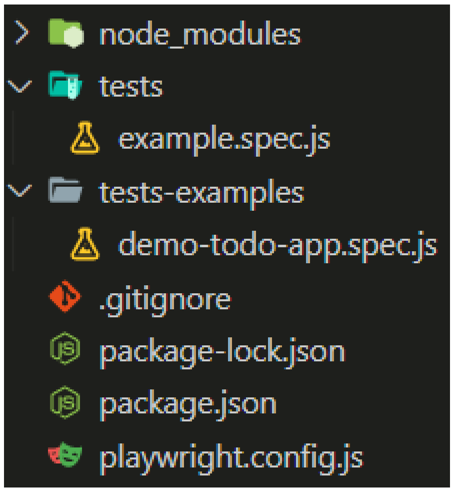
```
Initializing project in '.'
√ Do you want to use TypeScript or JavaScript? · JavaScript
√ Where to put your end-to-end tests? · tests
? Add a GitHub Actions workflow? (y/N) » false
```

Once the installation is completed you will get the message like “happy hacking”.



```
Happy hacking! 
```

We will be getting a folder structure like below.



```
> node_modules
v tests
    example.spec.js
v tests-examples
    demo-todo-app.spec.js
  .gitignore
  package-lock.json
  package.json
  playwright.config.js
```

The playwright.config is where you can add configuration for Playwright including modifying which browsers you would like to run Playwright on. If you are running tests inside an already existing project then dependencies will be added directly to your package.json.

b. Running a test case

By default tests will be run on all 3 browsers, chromium, firefox and webkit using 3 workers. By default the script will execute in headless mode.

```
npx playwright test
```

We can use the below command to execute the script in headed mode.

```
npx playwright test -headed
```

After running the following command we will be seeing something like this.

```
Running 3 tests using 3 workers
[3/3] [webkit] › example.spec.js:4:1 › homepage has Playwright in title and get started link linking to the intro page
```

c. Playwright config

To set a timeout we can use.

```
/* Maximum time one test can run for. */
timeout: 2 *30 * 1000,
```

For browsers selection we can comment/remove browsers name from the projects array. By default we will be running our tests on chrome, firefox and edge.

```
// {
//   name: "firefox",
//   use: {
//     ...devices["Desktop Firefox"],
//   },
// },
```

For executing multiple scripts at a time we can use the below command.

```
/* Run tests in files in parallel */
fullyParallel: true,
```

For generating reports we can use.

```
/* Reporter to use. See https://playwright.dev/docs/test-reporters */
reporter: process.env.REPORTER || 'html',
```

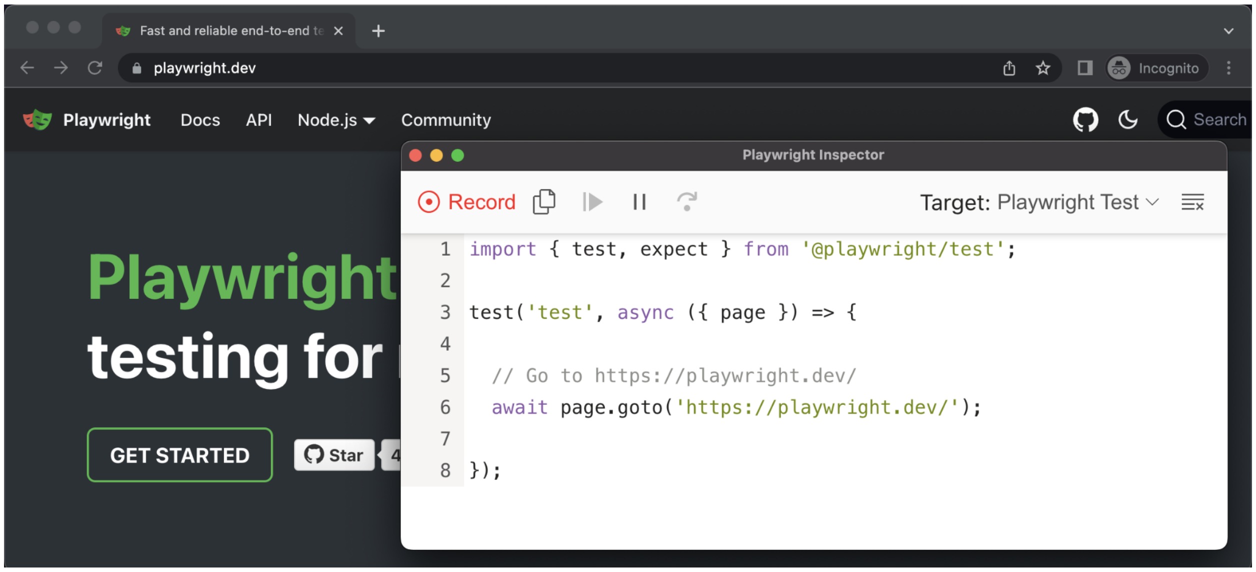
3. Recording a script.

Playwright comes with the ability to generate tests out of the box and is a great way to quickly get started with testing. It will open two windows, a browser window where you interact with the website you wish to test and the Playwright Inspector window where you can record your tests, copy the tests, clear your tests as well as change the language of your tests.

Syntax:- `npx playwright codegen   <URL>`

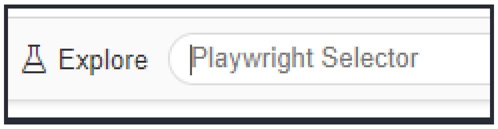
```
npx playwright codegen playwright.dev
```

Here playwright.dev can be changed with the required website for which we want to generate the code.

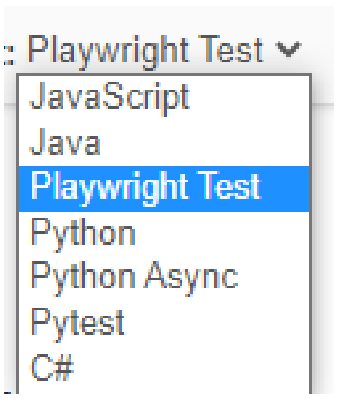


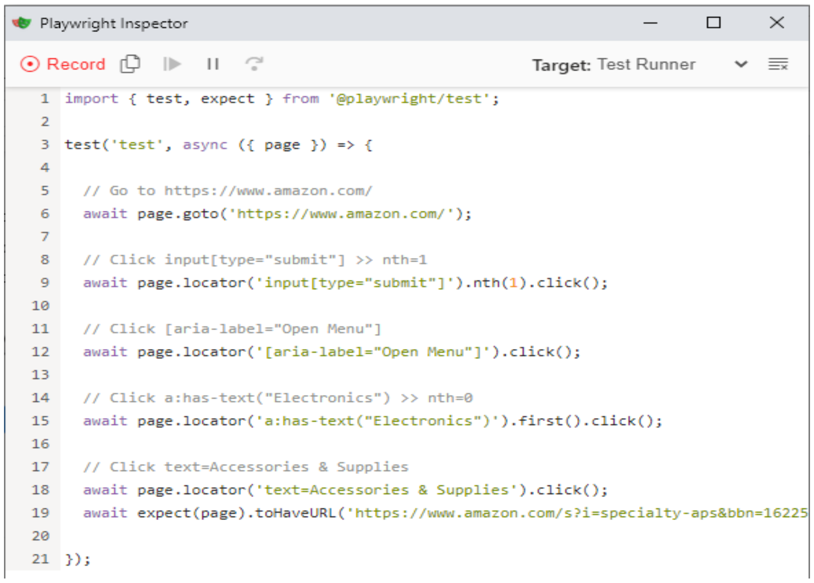
```
import { test, expect } from '@playwright/test';

test('test', async ({ page }) => {

  // Go to https://playwright.dev/
  await page.goto('https://playwright.dev/');

});
```

To find a locator we have to first click on the record option then an explore popup will come and when we click to explore we can go and select a particular element on the web and our locator will be appeared next to explore in playwright selector also we can write a selector there and check weather it is correct or not if correct it will highlight same on webpage.

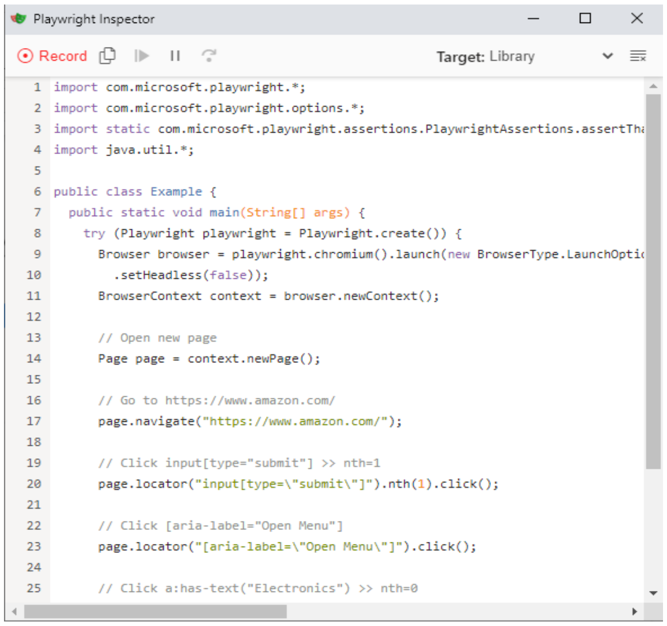


Once the recorder is on we can perform whatever task we have to perform on the web and in the backend, steps will be getting added in the recorder section and once we complete our execution we can copy and paste the code. Also we can change our code on biases on the language.

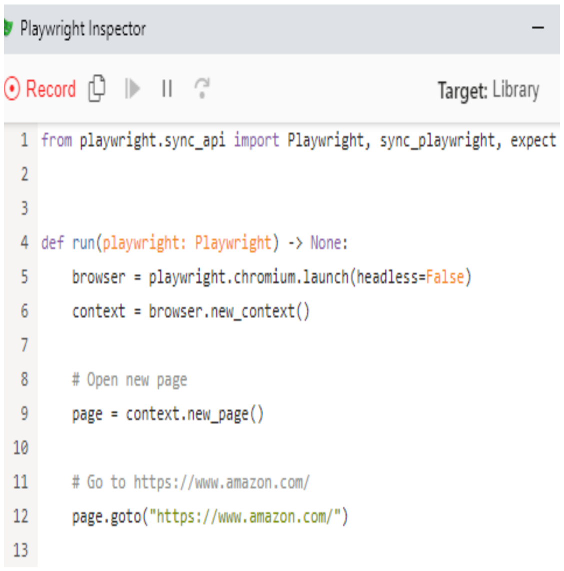




```
import { test, expect } from '@playwright/test';

test('test', async ({ page }) => {

  // Go to https://www.amazon.com/
  await page.goto('https://www.amazon.com/');

  // Click input[type="submit"] >> nth=1
  await page.locator('input[type="submit"]').nth(1).click();

  // Click [aria-label="Open Menu"]
  await page.locator('[aria-label="Open Menu"]').click();

  // Click a:has-text("Electronics") >> nth=0
  await page.locator('a:has-text("Electronics")').first().click();

  // Click text=Accessories & Supplies
  await page.locator('text=Accessories & Supplies').click();
  await expect(page).toHaveURL('https://www.amazon.com/s?i=specialty-aps&bbn=16225

});
```



```
import com.microsoft.playwright.*;
import com.microsoft.playwright.options.*;
import static com.microsoft.playwright.assertions.PlaywrightAssertions.assertTha
import java.util.*;

public class Example {
  public static void main(String[] args) {
    try (Playwright playwright = Playwright.create()) {
      Browser browser = playwright.chromium().launch(new BrowserType.LaunchOptic
        .setHeadless(false));
      BrowserContext context = browser.newContext();

      // Open new page
      Page page = context.newPage();

      // Go to https://www.amazon.com/
      page.navigate("https://www.amazon.com/");

      // Click input[type="submit"] >> nth=1
      page.locator("input[type=\"submit\"]").nth(1).click();

      // Click [aria-label="Open Menu"]
      page.locator("[aria-label=\"Open Menu\"]").click();

      // Click a:has-text("Electronics") >> nth=0
```



```
from playwright.sync_api import Playwright, sync_playwright, expect


def run(playwright: Playwright) -> None:
    browser = playwright.chromium.launch(headless=False)
    context = browser.new_context()

    # Open new page
    page = context.new_page()

    # Go to https://www.amazon.com/
    page.goto("https://www.amazon.com/")

```

4. Working with different browsers.

Each version of Playwright needs specific versions of browser binaries to operate. Depending on the language you use, Playwright will either download these browsers at package install time for you, or you will need to use Playwright CLI to install these browsers.

With every release, Playwright updates the versions of the browsers it supports, so that the latest Playwright would support the latest browsers at any moment. It means that every time you update playwright, you might need to re-run the install CLI command.

In newer versions browsers will be installed automatically with the other files but in older versions we have to explicitly mention the browser command to install all the browsers.

To install playwright browsers separately we have to use the below command:

```
npx playwright install
```

Playwright support:

- Chrome
- Webkit
- Firefox

For testing in mobile web browsers we have to use.

- Mobile chrome
- Mobile safari

5. Page Object Method (POM) with Playwright

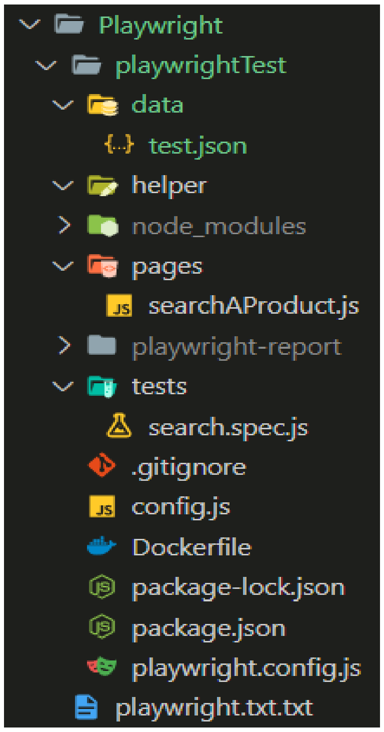
POM is a page of a design pattern that creates a repository for storing all web elements. It is useful in reducing code duplication and improves test script maintenance.

In Page Object Model, consider each web page of an application as a separate class file. Each class file will contain only corresponding web page elements. Using these elements, testers can perform operations on the website under test.

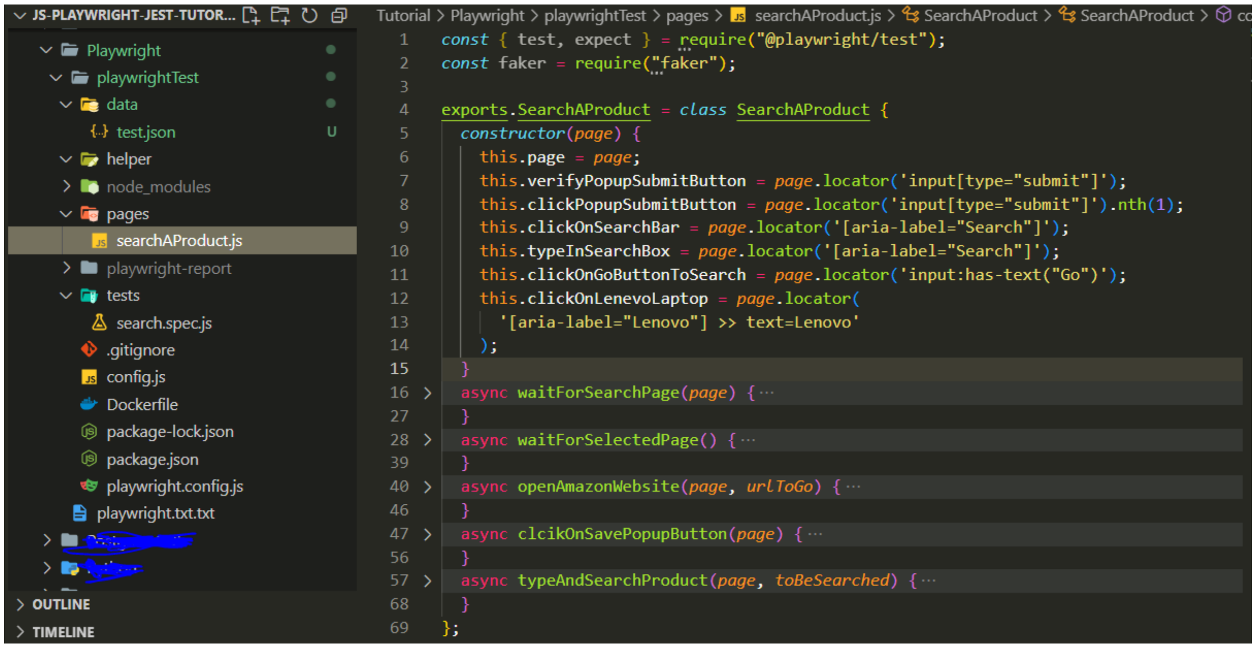
Advantage of page object method:

- Easy Maintenance
- Increased Reusability
- Readability

To write a POM we have to first create a directory where we can write our page methods and export it for storing any data files. We can create a separate repository and in last to write test we can write those under test files. A glimpse of folder structure has been provided below.



```
Playwright
  playwrightTest
    data
      test.json
    helper
    node_modules
    pages
      searchAProduct.js
    playwright-report
    tests
      search.spec.js
    .gitignore
    config.js
    Dockerfile
    package-lock.json
    package.json
    playwright.config.js
  playwright.txt.txt
```

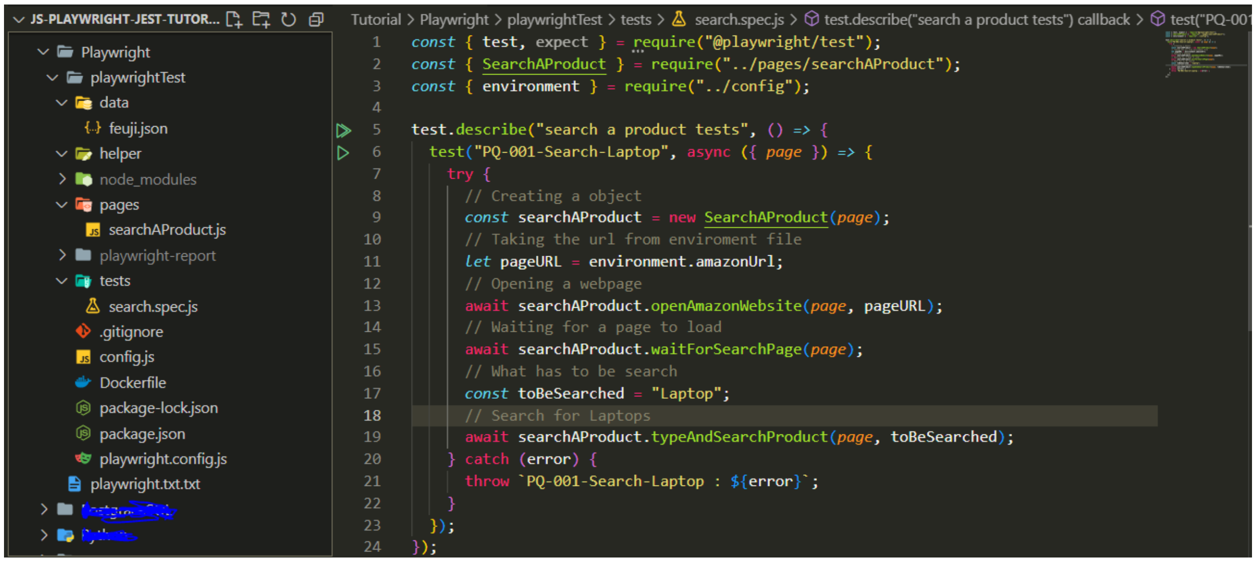


```
JS-PLAYWRIGHT-JEST-TUTOR...
  Playwright
    playwrightTest
      data
        test.json                U
      helper
      node_modules
      pages
        searchAProduct.js
      playwright-report
      tests
        search.spec.js
      .gitignore
      config.js
      Dockerfile
      package-lock.json
      package.json
      playwright.config.js
    playwright.txt.txt
  [redacted]
  [redacted]
OUTLINE
TIMELINE
```

Tutorial > Playwright > playwrightTest > pages > searchAProduct.js > SearchAProduct > SearchAProduct > co

```javascript
const { test, expect } = require("@playwright/test");
const faker = require("faker");

exports.SearchAProduct = class SearchAProduct {
  constructor(page) {
    this.page = page;
    this.verifyPopupSubmitButton = page.locator('input[type="submit"]');
    this.clickPopupSubmitButton = page.locator('input[type="submit"]').nth(1);
    this.clickOnSearchBar = page.locator('[aria-label="Search"]');
    this.typeInSearchBox = page.locator('[aria-label="Search"]');
    this.clickOnGoButtonToSearch = page.locator('input:has-text("Go")');
    this.clickOnLenevoLaptop = page.locator(
      '[aria-label="Lenovo"] >> text=Lenovo'
    );
  }
  async waitForSearchPage(page) {…
  }
  async waitForSelectedPage() {…
  }
  async openAmazonWebsite(page, urlToGo) {…
  }
  async clcikOnSavePopupButton(page) {…
  }
  async typeAndSearchProduct(page, toBeSearched) {…
  }
};
```

And once written we can import them in our required script file to perform a particular operation and also we can implement a try catch block for error handling.



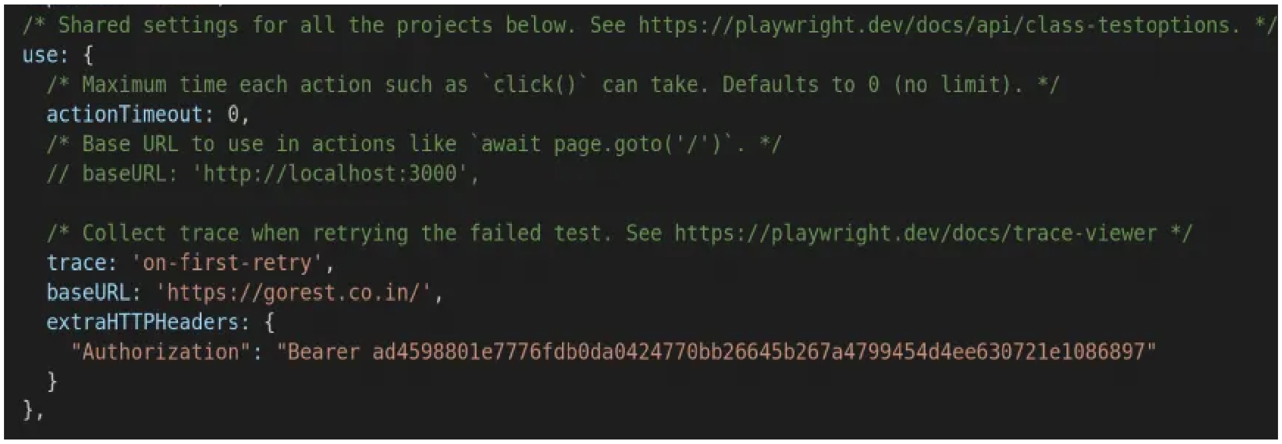
```
const { test, expect } = require("@playwright/test");
const { SearchAProduct } = require("../pages/searchAProduct");
const { environment } = require("../config");

test.describe("search a product tests", () => {
  test("PQ-001-Search-Laptop", async ({ page }) => {
    try {
      // Creating a object
      const searchAProduct = new SearchAProduct(page);
      // Taking the url from enviroment file
      let pageURL = environment.amazonUrl;
      // Opening a webpage
      await searchAProduct.openAmazonWebsite(page, pageURL);
      // Waiting for a page to load
      await searchAProduct.waitForSearchPage(page);
      // What has to be search
      const toBeSearched = "Laptop";
      // Search for Laptops
      await searchAProduct.typeAndSearchProduct(page, toBeSearched);
    } catch (error) {
      throw `PQ-001-Search-Laptop : ${error}`;
    }
  });
});
```

6. API testing using playwright.

API testing is a type of software testing that analyzes an application program interface (API) to verify it fulfills its expected functionality, security, performance and reliability. The tests are performed either directly on the API or as part of integration testing. An API is middleware code that enables two software programs to communicate with each other. The code also specifies the way an application requests services from the operating system (OS) or other applications.

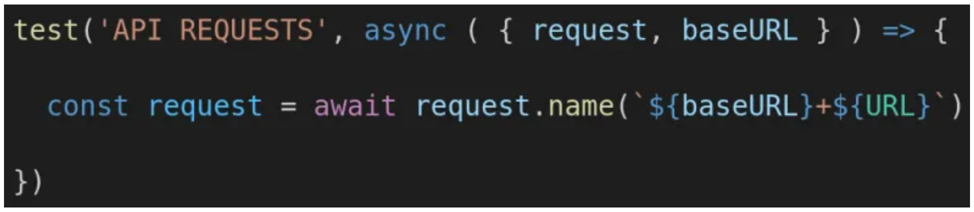
Applications frequently have three layers: a data layer, a service layer -- the API layer -- and a presentation layer -- the user interface (UI) layer. The business logic of the application -- the guide to how users can interact with the services, functions and data held within the app -- is in the API layer. API testing focuses on analyzing the business logic as well as the security of the application and data responses. An API test is generally performed by making requests to one or more API endpoints and comparing the response with expected results.

To perform api automation testing using playwright we have to configure the playwright.config first. We have to add the baseURL, trace and authentication if needed. (It's not mandatory to write like these. We can mention directly in the code we are using as it is known as best practice).



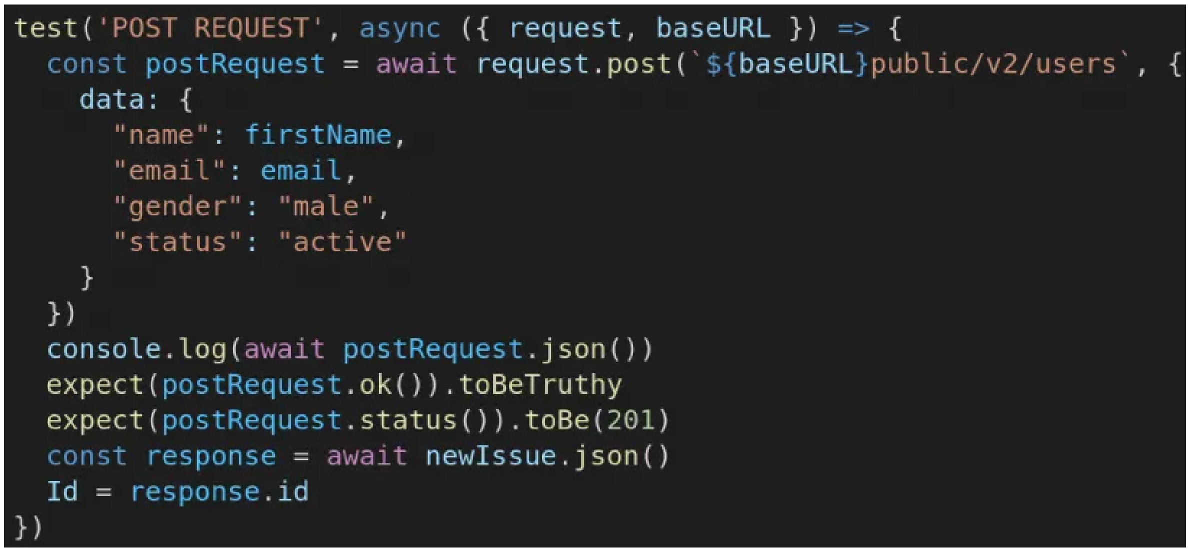
```
/* Shared settings for all the projects below. See https://playwright.dev/docs/api/class-testoptions. */
use: {
  /* Maximum time each action such as `click()` can take. Defaults to 0 (no limit). */
  actionTimeout: 0,
  /* Base URL to use in actions like `await page.goto('/')`. */
  // baseURL: 'http://localhost:3000',

  /* Collect trace when retrying the failed test. See https://playwright.dev/docs/trace-viewer */
  trace: 'on-first-retry',
  baseURL: 'https://gorest.co.in/',
  extraHTTPHeaders: {
    "Authorization": "Bearer ad4598801e7776fdb0da0424770bb26645b267a4799454d4ee630721e1086897"
  }
},
```

Now first we have to create our endpoint where we want to hit the target.



```
test('API REQUESTS', async ( { request, baseURL } ) => {

  const request = await request.name(`${baseURL}+${URL}`)

})
```

Once done we will be now creating different scripts for different kinds of operations (CURD). For post requests there is an example below.



```
test('POST REQUEST', async ({ request, baseURL }) => {
  const postRequest = await request.post(`${baseURL}public/v2/users`, {
    data: {
      "name": firstName,
      "email": email,
      "gender": "male",
      "status": "active"
    }
  })
  console.log(await postRequest.json())
  expect(postRequest.ok()).toBeTruthy
  expect(postRequest.status()).toBe(201)
  const response = await newIssue.json()
  Id = response.id
})
```

7. Route in playwright

Route allows you to intercept and mock network requests made by a web page. It provides us with 5 simple methods which are listed below.

- abort
    - route.abort();
    - It will abort the request that is triggered by the browser and it will send a response that will have an error message.

```
await page.route("**/api/data", (route) => {

      route.abort();

});
```

- continue
    - route.continue();
    - This method is used within a route callback to allow a network request to proceed to the actual network in other words we can say as route's request with optional overrides.

```
await page.route("**/*", (route, request) => {

      // Override headers

      const headers = {

            ...request.headers(),

            foo: "foo-value", // set "foo" header

            bar: undefined, // remove "bar" header

      };

      route.continue({ headers });

});
```

- fallback
    - route.fallback();
    - This method is used within a route callback to allow an intercepted network request to fall back to the default network behavior if no custom response is provided. This is useful when you want to selectively intercept and modify certain requests while letting others proceed without interception. In other words we can say when several routes match the given pattern, they run in the order opposite to their registration. That way the last registered route can always override all the previous ones.

```
// Intercept network request and provide a custom response, or fallback to network

await page.route('**/api/data', (route) => {

  if (/* condition to provide a custom response */) {

    route.fulfill({

      status: 200,

      contentType: 'application/json',

      body: JSON.stringify({ message: 'Custom data' }),

    });

  } else {

    route.fallback(); // Allow the request to proceed to the network

  }

});
```

- fetch
    - route.fetch();
    - Performs the request and fetches the result without fulfilling it, so that the response could be modified and then fulfilled.

```
await page.route('https://dog.ceo/api/breeds/list/all', async route => {
  const response = await route.fetch();
  const json = await response.json();
  json.message['big_red_dog'] = [];
  await route.fulfill({ response, json });
});
```

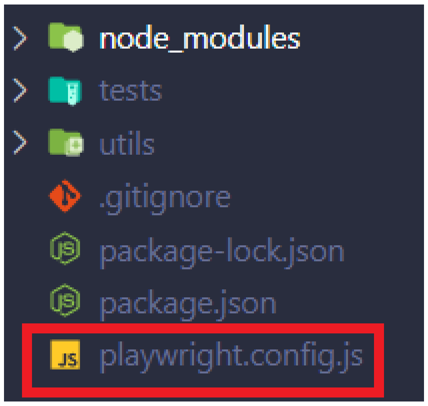
- fulfill
  - route.fulfill();
  - Performs the request and fetches the result without fulfilling it, so that the response could be modified and then fulfilled.

```
await page.route('**/*', route => {
  route.fulfill({
    status: 404,
    contentType: 'text/plain',
    body: 'Not Found!'
  });
});
```

- request
  - route.request();

8. How to configure playwright to select test-id from UI?

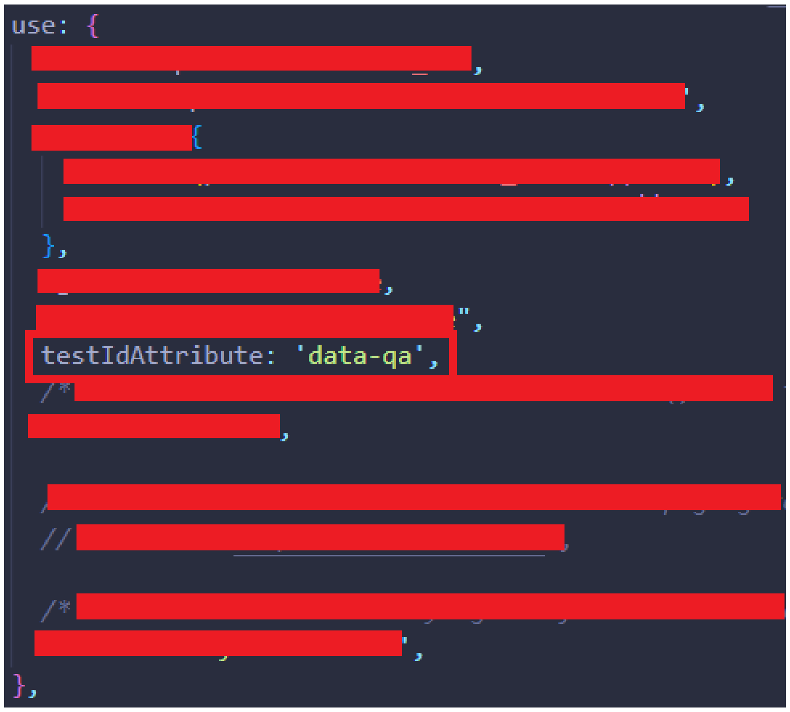
Step 1: First we have to open our playwright config file.



```
> node_modules
> tests
> utils
  .gitignore
  package-lock.json
  package.json
  playwright.config.js
```

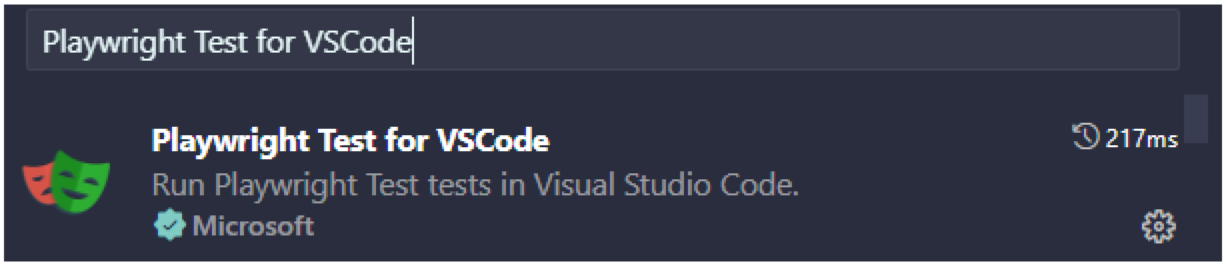
Step 2: We have to go to the “use” section if it is not there we have to create a use section.

Step 3: Add testIdAttribute: ‘attribute_name’.



```
use: {
  [redacted],
  [redacted]',
  [redacted]{
    [redacted],
    [redacted]
  },
  [redacted],
  [redacted]",
  testIdAttribute: 'data-qa',
  /*[redacted]
  [redacted],

  /[redacted]
  //[redacted],

  /*[redacted]
  [redacted]',
},
```

Step 4: Install an extension in our VS_Code “Playwright Test for VSCode”.

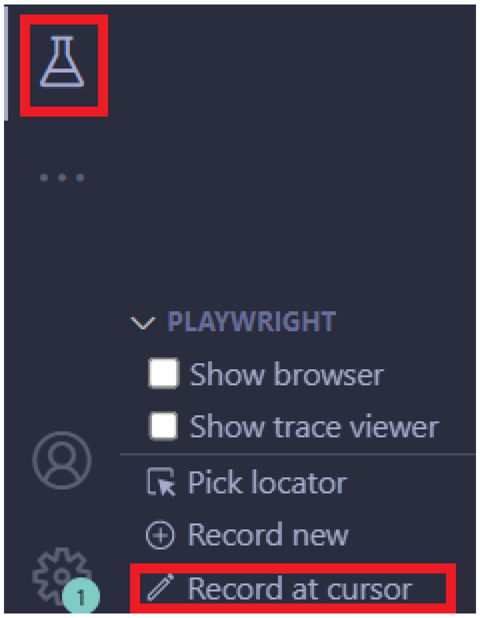

Playwright Test for VSCode

Playwright Test for VSCode

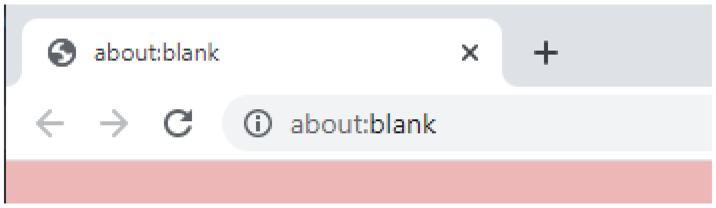
Run Playwright Test tests in Visual Studio Code.

Microsoft

Step 5: A new Tab will be visible named testing. Open the tab and then open a script where you want to record.

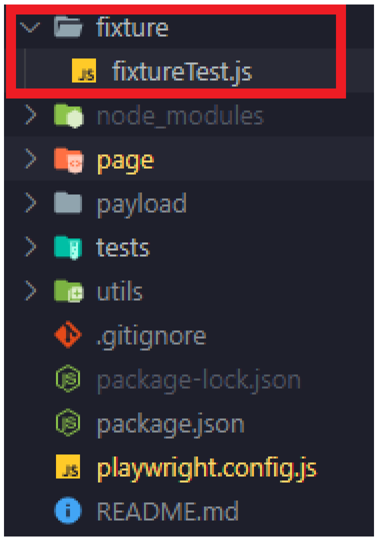


Step 6: Select Record a cursor and start using the generated script.



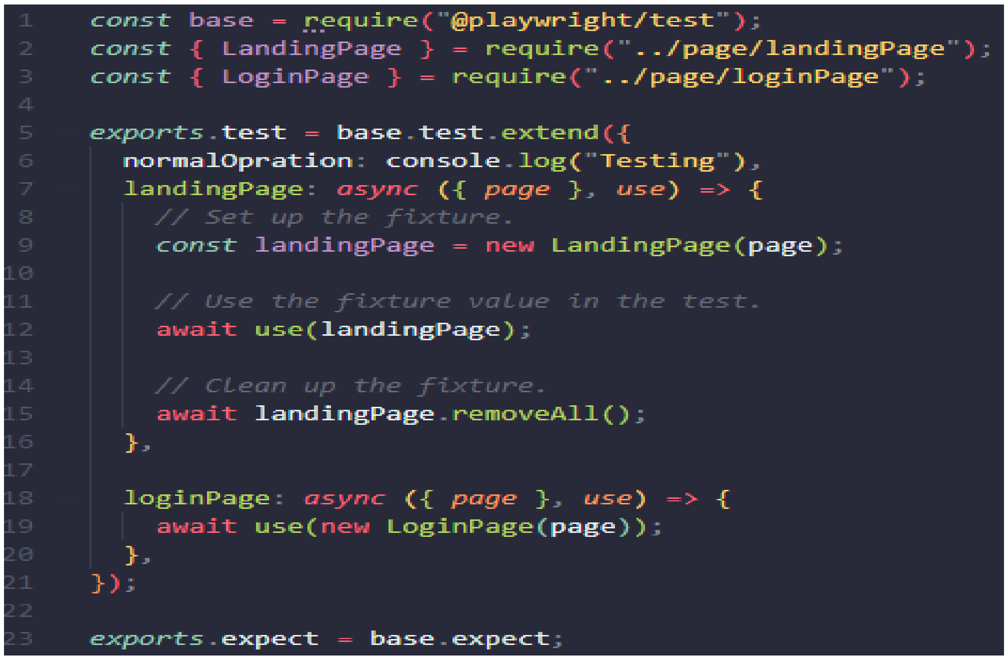
9. Fixtures in playwright.

Playwright Test is based on the concept of test fixtures. Test fixtures are used to establish an environment for each test, giving the test everything it needs and nothing else. Test fixtures are isolated between tests. With fixtures, you can group tests based on their meaning, instead of their common setup. Here is an example for you. Let's look at the file structure.



```
fixture
  fixtureTest.js
node_modules
page
payload
tests
utils
.gitignore
package-lock.json
package.json
playwright.config.js
README.md
```

So we will create a separate folder for fixtures and we can keep all our fixture files in that. Do we need to create a separate folder for fixtures actually no but it will help to differentiate the file easily.



```javascript
const base = require("@playwright/test");
const { LandingPage } = require("../page/landingPage");
const { LoginPage } = require("../page/loginPage");

exports.test = base.test.extend({
  normalOpration: console.log("Testing"),
  landingPage: async ({ page }, use) => {
    // Set up the fixture.
    const landingPage = new LandingPage(page);

    // Use the fixture value in the test.
    await use(landingPage);

    // Clean up the fixture.
    await landingPage.removeAll();
  },

  loginPage: async ({ page }, use) => {
    await use(new LoginPage(page));
  },
});

exports.expect = base.expect;
```

As we can see in the above image we have to import the relevant files

10. References.

a. Playwright documentation https://playwright.dev/

b. Github https://github.com/microsoft/playwright

c. Microsoft Learn https://learn.microsoft.com/en-us/microsoft-edge/playwright/

d. Wikipedia https://en.wikipedia.org/wiki/Playwright

e. BrowserStack https://www.browserstack.com/guide/playwright-tutorial

f. LambdaTest https://www.lambdatest.com/playwright