**[PLAYWRIGHT-TS-UI-API-FW](https://github.com/build-vtest/PLAYWRIGHT-TS-UI-API-FW)**

**Table of Content**

**Title Page No.**

1. [**Introduction**](#Introduction)**------------------------------------------------------------------ 2**
2. [**Prerequisite**](#Prerequisites)**------------------------------------------------------------------ 2**
3. [**Getting Started**](#Getting_Started)**-------------------------------------------------------------- 3**
4. [**Page Objects**](#Page_Objects)**----------------------------------------------------------------- 4**
5. [**Test Scripts**](#Test_Scripts)**------------------------------------------------------------------- 7**
6. [**Assertions and Verifications**](#Assesrtions)**--------------------------------------------- 8**
7. [**Test data management**](#Test_Data_Management)**---------------------------------------------------- 10**
8. [**Parallel Execution and Cross Browser Execution Reporting**](#Parallel_Execution)**----- 14**
9. [**Reporting**](#Reporting)**---------------------------------------------------------------------- 16**
10. [**Continuous Integration**](#CI_CD)**--------------------------------------------------- 19**
11. [**Execution in different Environments**](#Environments)**---------------------------------- 22**
12. [**Debugging and Troubleshooting**](#Debugging)**--------------------------------------- 24**
13. [**Conclusion**](#Conclusion)**-------------------------------------------------------------------- 27**
14. **Introduction:**

The Playwright with TypeScript, Page Object Model framework is a powerful solution for automating the testing of web applications. It simplifies the process of creating, maintaining, and executing test scripts, resulting in faster and more reliable testing.

Our framework leverages Playwright, a powerful browser automation library, to interact with web applications across different browsers. TypeScript is used for writing test scripts, offering type safety and improved code maintainability.

1. **Prerequisites:**

Before setting up a Playwright with TypeScript project, you'll need to ensure that you have certain prerequisites in place. Here are the key prerequisites for such a project:

**1. Node.js:** You need to have Node.js installed on your system. Playwright and many of its dependencies are Node.js packages. You can download and install Node.js from the official website:

**[Node.js Downloads]** [**https:/nodejs.org/en/download/)**](https://nodejs.org/en/download/))

**2. Package Manager (NPM):** Node.js comes with npm (Node Package Manager) by default. This package manager installed. To check if you have npm installed, run:

**`npm -v`** in your terminal

**3. Code Editor:** You'll need a code editor for writing and managing your TypeScript and test scripts. Popular choices include Visual Studio Code (VS Code), WebStorm, and others. Choose the one you're most comfortable with.

**- Download and install Visual Studio Code: [VS Code] (**[**https://code.visualstudio.com/download**](https://code.visualstudio.com/download)**)**

**4. Git:** While not strictly required, having Git installed can be useful for version control and collaboration with others.

**- Download and install Git: [Git Downloads] (**[**https://git-scm.com/downloads**](https://git-scm.com/downloads)**)**

**5. Playwright:** You'll need to install Playwright as a development dependency in your project. This can be done using npm. To install Playwright with npm, run:

**6. TypeScript:** Since you're creating a Playwright project with TypeScript, you'll need TypeScript installed.

**npm install playwright**

- To install TypeScript with npm, run: **`npm install typescript`**

**7. Test Runner:** Playwright works with various test runners like Jest, Mocha, and Playwright Test. We have chosen *‘Playwright test*’ test runner.

**npm install -D @playwright/test**

1. **Getting Started**
2. Clone the repo using below URL

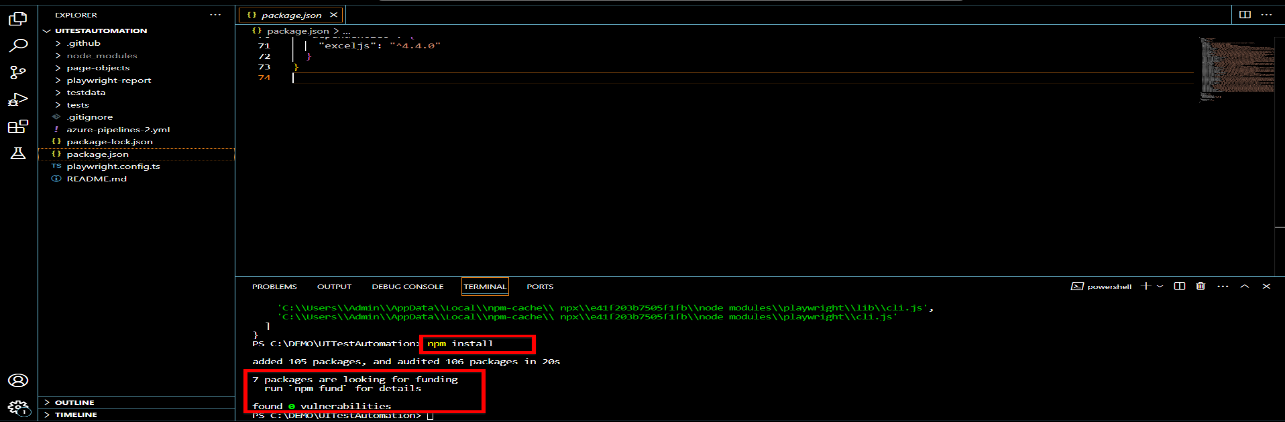
[build-vtest/PLAYWRIGHT-TS-UI-API-FW (github.com)](https://github.com/build-vtest/PLAYWRIGHT-TS-UI-API-FW)

Enter following command in command prompt to clone the repository:

git clone "https://github.com/build-vtest/Playwright-TS-Cucumber-BDD-Framework.git"

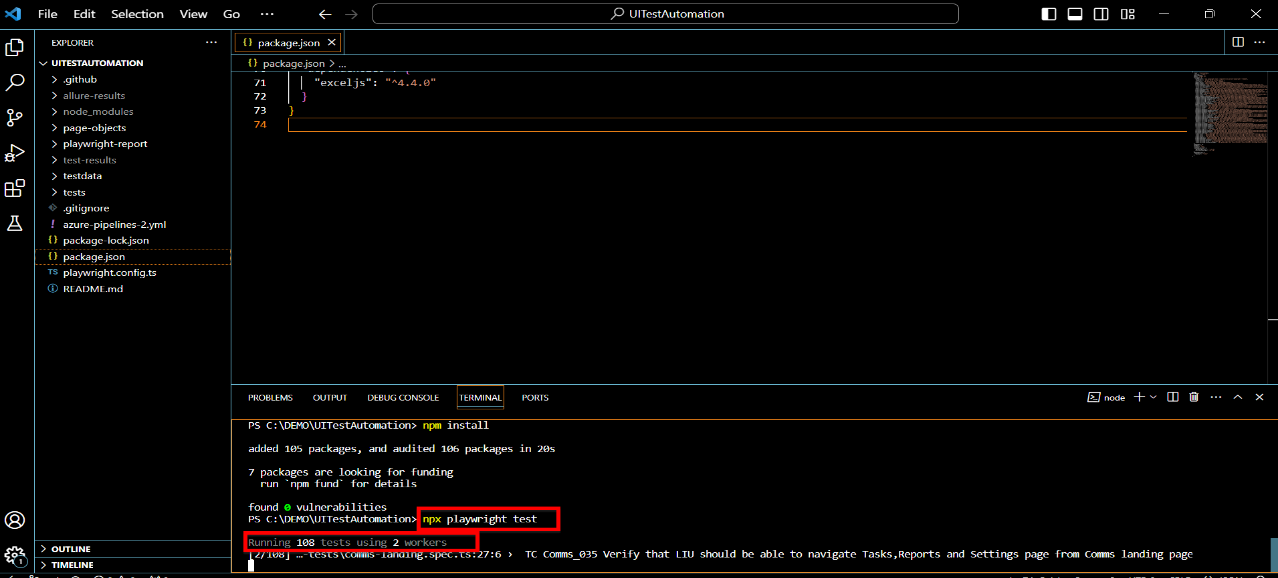
1. Navigate to folder and install npm packages using:

**npm install**



1. All the dependencies now been downloaded, you all set to run the test cases. Use following command to run your test cases on your local machine. Once tests are running successfully you can delete the existing test cases, pages and test data and add your own tests, pages and test data as guided in following sections.

**npx playwright test**



1. **Page Objects**

The Page Object Model (POM) is a design pattern that enhances the organization and maintainability of automated test scripts. It achieves this by separating the definition of web pages and their elements from the test logic, resulting in more readable and maintainable code.

**POM Components:**

The POM consists of three main components:

* 1. Page Objects
  2. Element Locators
  3. Test Scripts.

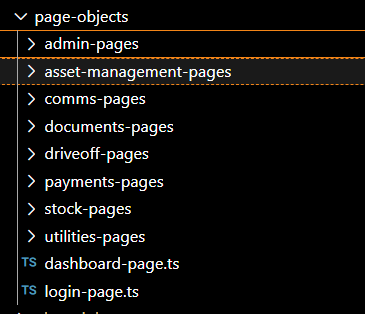
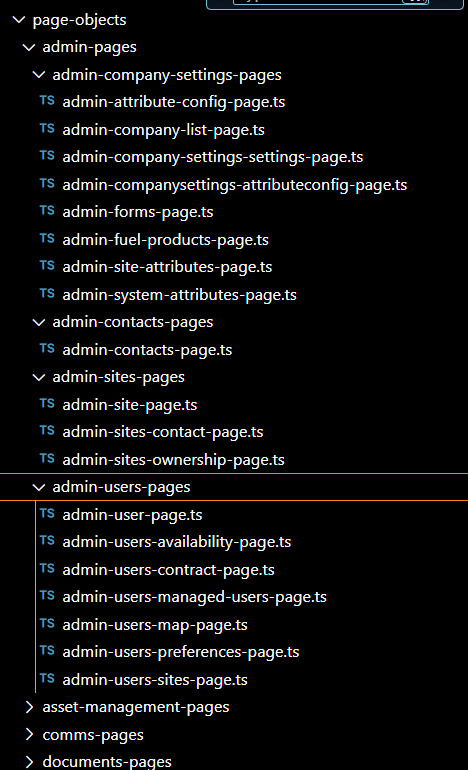
Page Objects represent web pages, Element Locators define how to locate elements on those pages, and Test Scripts interact with these Page Objects using Element Locators to perform tests.

**Advantages of POM:**

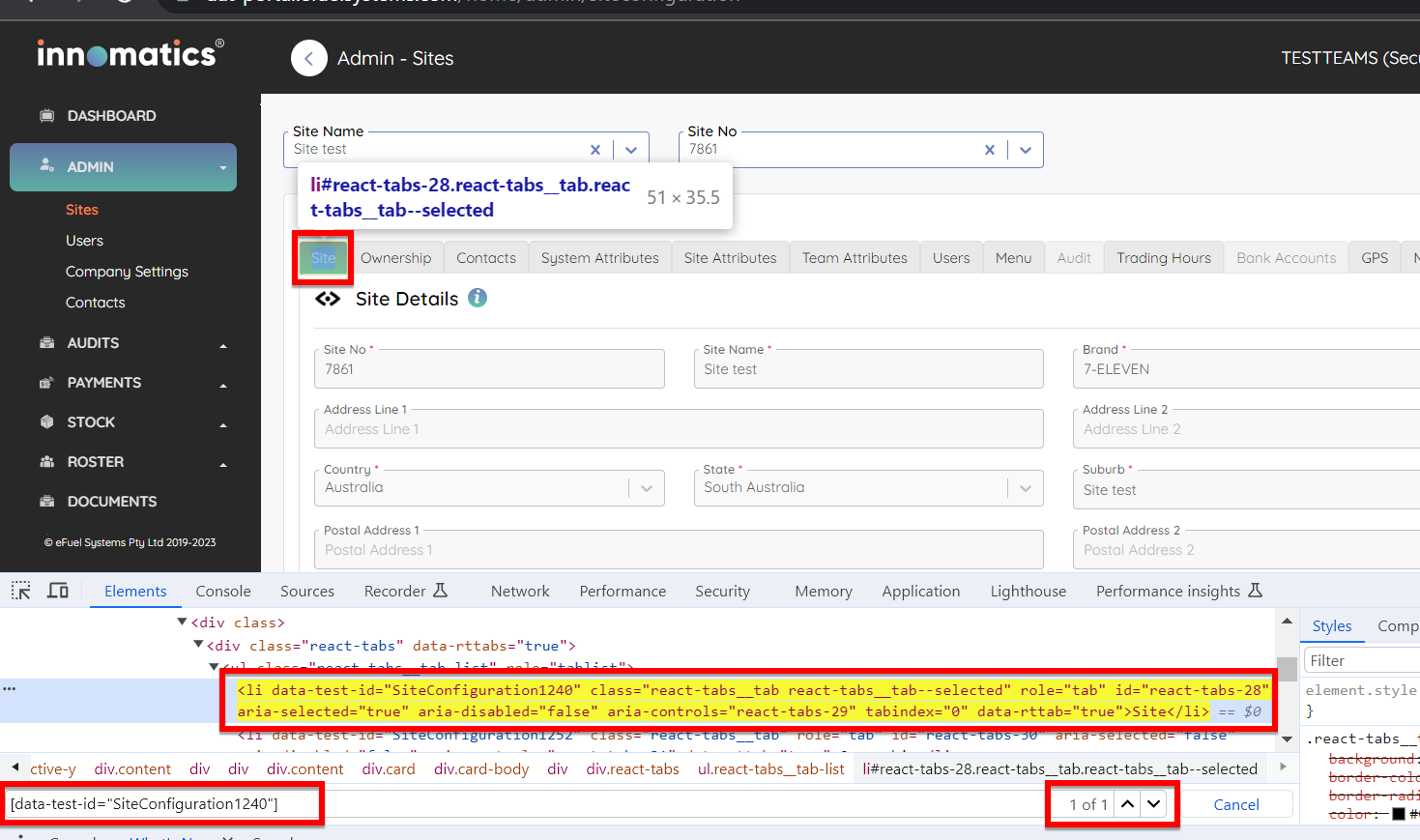
Implementing the POM in our framework offers several advantages. It simplifies code maintenance by centralizing element locators and page interactions. It enhances reusability, reducing duplication of code. Additionally, POM promotes collaboration between testers and developers.

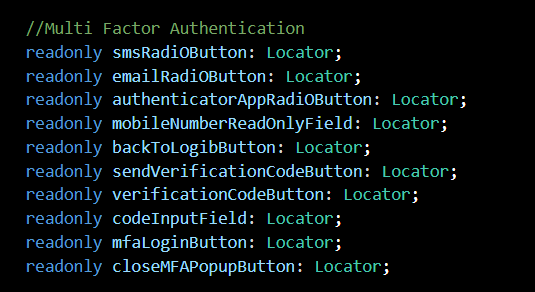
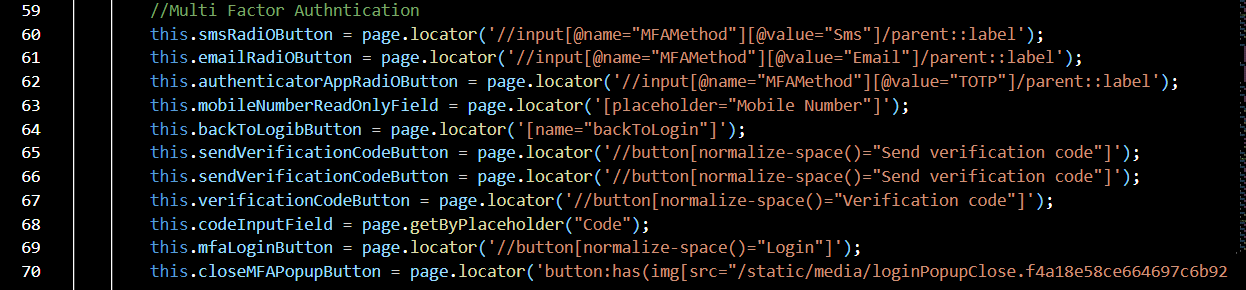
**Creating Page Objects:**

To create Page Objects in our framework, you should create a dedicated page object class for each web page in your application. These classes encapsulate the page's elements and interactions. In our framework, we have followed a specific naming convention, such as 'login-page.ts' or 'dashboard-page.ts' to maintain consistency.

**Defining Element Locators:**

Element Locators within Page Objects are responsible for identifying and interacting with web elements. You can define locators using Playwright's selectors, CSS selectors, or XPath, depending on your preference and the structure of the web page.

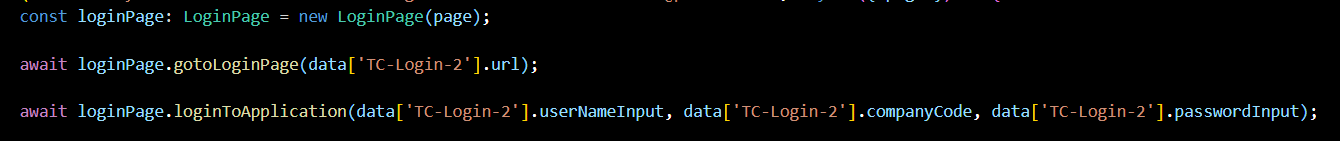
**1. Declaring the locator variable** **2. Initialising the locator variable with**

In our framework for most of the part we have used CSS selectors to locate elements but for few elements for example to locate dynamic elements we have also used XPath as a locator strategy.

We have used unique ‘data-test-id’ attribute to locate the elements on the web page. Few elements for which the data-test-id attribute is not present, we used class, id or the text () to locate those elements.

**Using Page Objects in Test Scripts:**

In our test scripts, you'll need to instantiate Page Objects and utilize Element Locators to interact with the web elements on the page. This separation of concerns allows for clean and concise test scripts that are easy to read and maintain. For example, to log in to the application in each test case we need locators and locator actions from ‘login-page.ts’ so we have to create an object or an instance of LoginPage class to get access to those elements and element specific actions as follow,



**Reusable Elements and Actions:**

By designing Page Objects and Element Locators for maximum reusability, we can save time and effort. Common elements and actions, such as navigation menus, form submissions or click actions, have been encapsulated in reusable components for efficient test script development.

1. **Test Scripts**

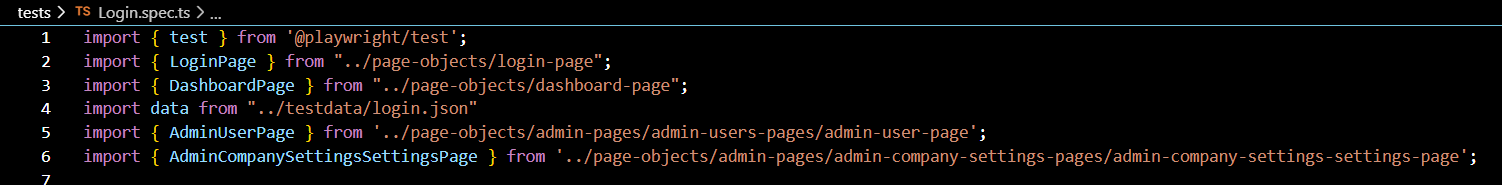
Creating test scripts using Page Objects in a Playwright with TypeScript Page Object Model (POM) framework is a structured approach that separates the test logic from the representation of web pages. Here's a step-by-step guide on how to create test scripts using Page Objects:

**Step 1: Set Up Your Test Project:**

Before you start creating test scripts, ensure that your Playwright with TypeScript POM framework is set up and configured properly. This includes having your Page Objects and Element Locators defined.

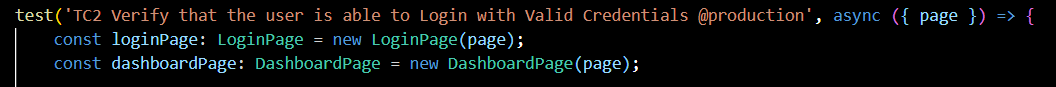
**Step 2: Import Necessary Dependencies:**

In your test script file, import the necessary dependencies. This typically includes importing Playwright, your custom Page Objects, and any other utility functions or libraries you may need.



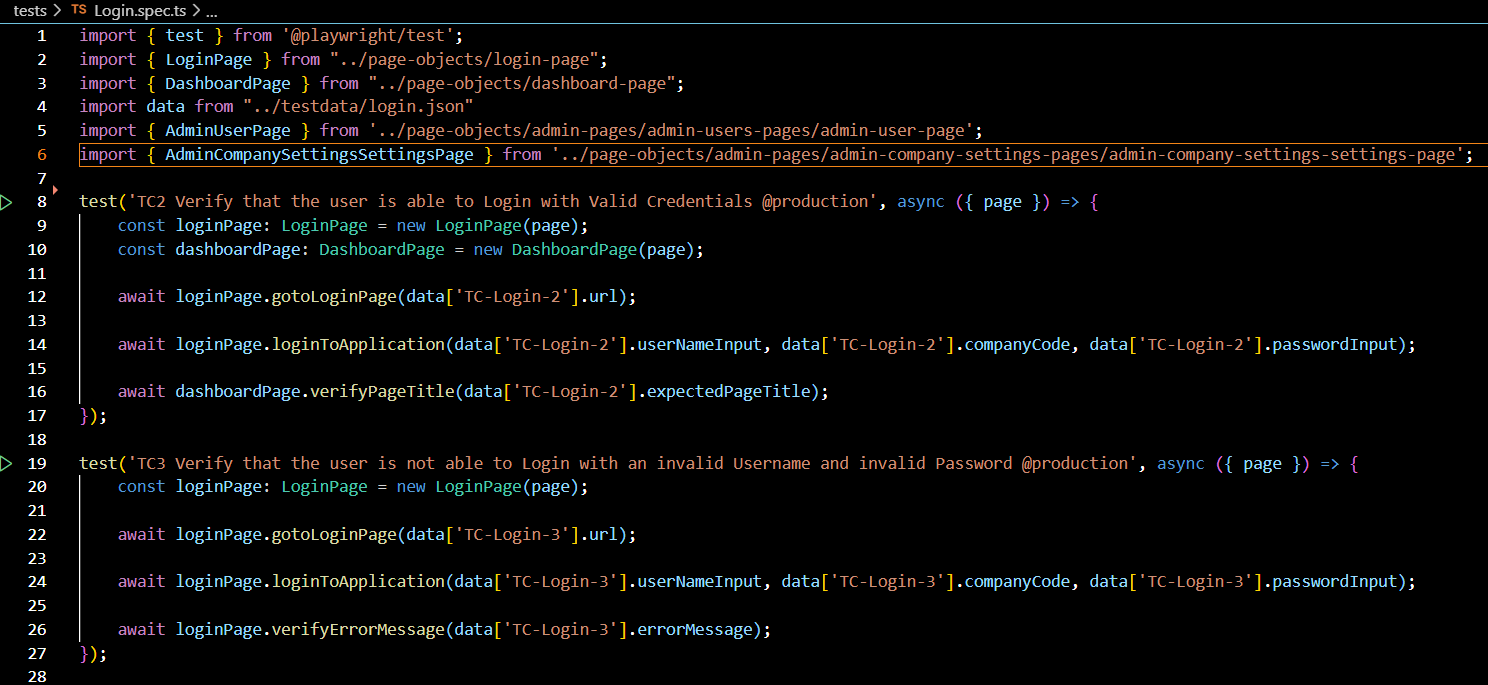
**Step 3: Initialize the Playwright Browser and Page Objects:**

Create instances of the Playwright browser and pages, and initialize your Page Objects.



**Step 4: Write Test Cases:**

With the browser, page, and Page Objects initialized, you can start writing your test cases. Interact with the web elements using the Page Objects and Element Locators defined in your Page Objects.



1. **Assertions and Verifications**

Assertions and verifications are essential elements in test scripts that help ensure the expected behaviour of the application. In a Playwright with TypeScript Page Object Model (POM) framework, we have performed assertions and verifications using the expect function as follow.

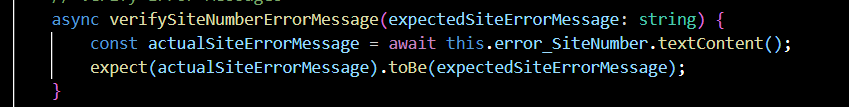
1. **Import the expect function:**

**import {expect} from 'playwright/test';**

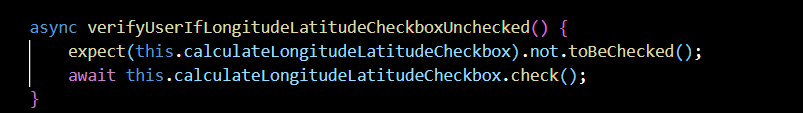
1. **Perform Assertions:**

We have Used the ‘expect function’ to make assertions by comparing actual and expected values. Common assertions include checking equality, containment, visibility, attributes, and more. Following is the way how we can use the assertions.

i) In below assertion we are checking if the actual error message that is coming on not filling the site number input field is same as expected or not. If actual error message and the expected error message are exact match then our test case will get passed otherwise the test will fail.



ii) In below assertion we are checking if the latitude checkbox on our page is by default unchecked or not. If it is unchecked then our test case will get passed otherwise the test will fail.



Following is the list of few assertions that we can use according to our test cases from playwright.

1. **expect(page).toHaveURL(url):** Asserts that the current page URL matches the expected URL.
2. **expect(page).toContainText(text):** Asserts that the page contains the specified text.
3. **expect(element).toExist():** Asserts that the element exists on the page.
4. **expect(element).toNotExist():** Asserts that the element does not exist on the page.
5. **expect(element).toHaveAttribute(name, [value]):** Asserts that the element has the specified attribute with an optional value.
6. **expect(element).toHaveText(text):** Asserts that the element's text content matches the expected text.
7. **expect(element).toHaveValue(value):** Asserts that the element's value matches the expected value.
8. **expect(element).toHaveCSS(property, value):** Asserts that the element has the specified CSS property with the expected value.
9. **expect(element).toHaveFocus():** Asserts that the element has focus.
10. **expect(element).not.toHaveFocus():** Asserts that the element does not have focus.
11. **expect(element).toBeChecked():** Asserts that a checkbox or radio button element is checked.
12. **expect(element).not.toBeChecked():** Asserts that a checkbox or radio button element is not checked.
13. **expect(element).toBeDisabled():** Asserts that the element is disabled.
14. **expect(element).not.toBeDisabled():** Asserts that the element is not disabled.
15. **expect(element).toHaveCount(number):** Asserts that there are the specified number of matching elements.
16. **expect(element).toContainText(text):** Asserts that the element contains the specified text.
17. **expect(element).toHaveClass(className):** Asserts that the element has the specified class.
18. **expect(element).not.toHaveClass(className):** Asserts that the element does not have the specified class.

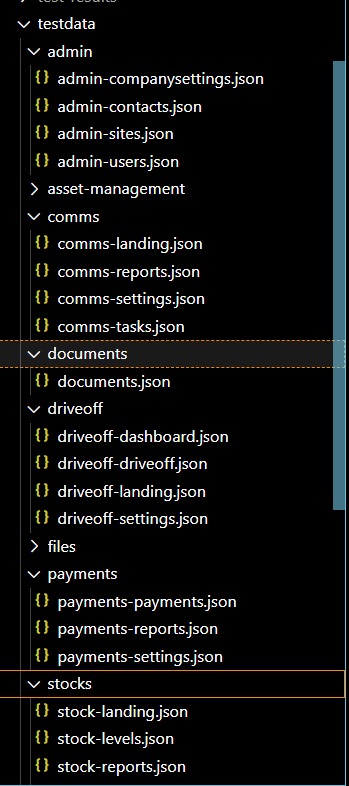
These assertion functions make it easy to write clear and expressive test cases in Playwright/test by allowing you to check the state and properties of elements and pages. You can use them to validate the behaviour of our web application during testing.

1. **Test Data Management**

Managing test data is an important aspect of test automation. Here's a guide on how to manage test data in our Playwright framework using JSON files:

**1. Organizing Test Data:**

We have created a dedicated folder ‘testdata’ in our project for test data at root folder level. Inside ‘testdata’ folder we have other folders named as the modules of the application. For example: admin, comms, documents, driveoff, payments and stocks. Inside each folder we have JSON files named according to the submodules under each module.



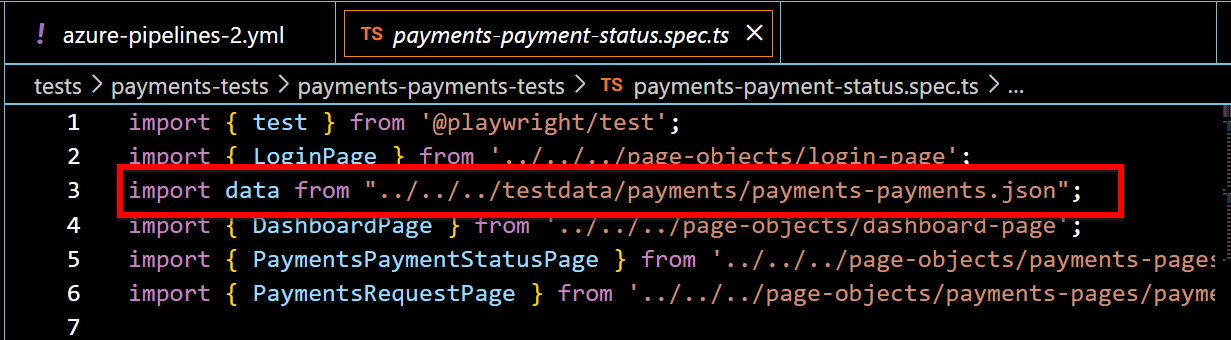
**2. Using JSON Files:**

JSON files are a structured and common way to store test data Each JSON file will contain the separate test data for the test cases for that particular module > sub module. Following is the example of JSON file which contains test data for ‘Payments’ submodule from the Payments module. Each JSON object is named according to the ‘test id’ of a particular test case. For example, **‘TC\_Pay\_001’** represents the test data for the test case **‘TC\_Pay\_001’** and **‘TC\_Pay\_002’** represents the test data for the test case **‘TC\_Pay\_002’** and so on.



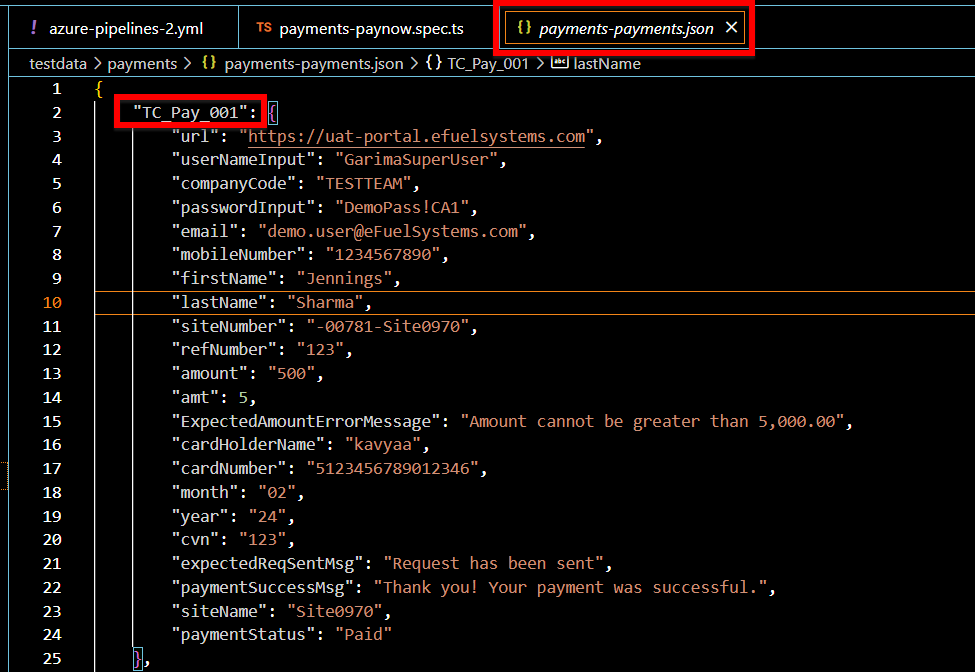
1. **Reading JSON Data:**

In our Playwright test scripts, we have TypeScript to read data from the JSON files. To use data from JSON we have to import a data from the particular JSON file. Following is the way how we import it.



Once imported the data from the JSON file is used the following way:





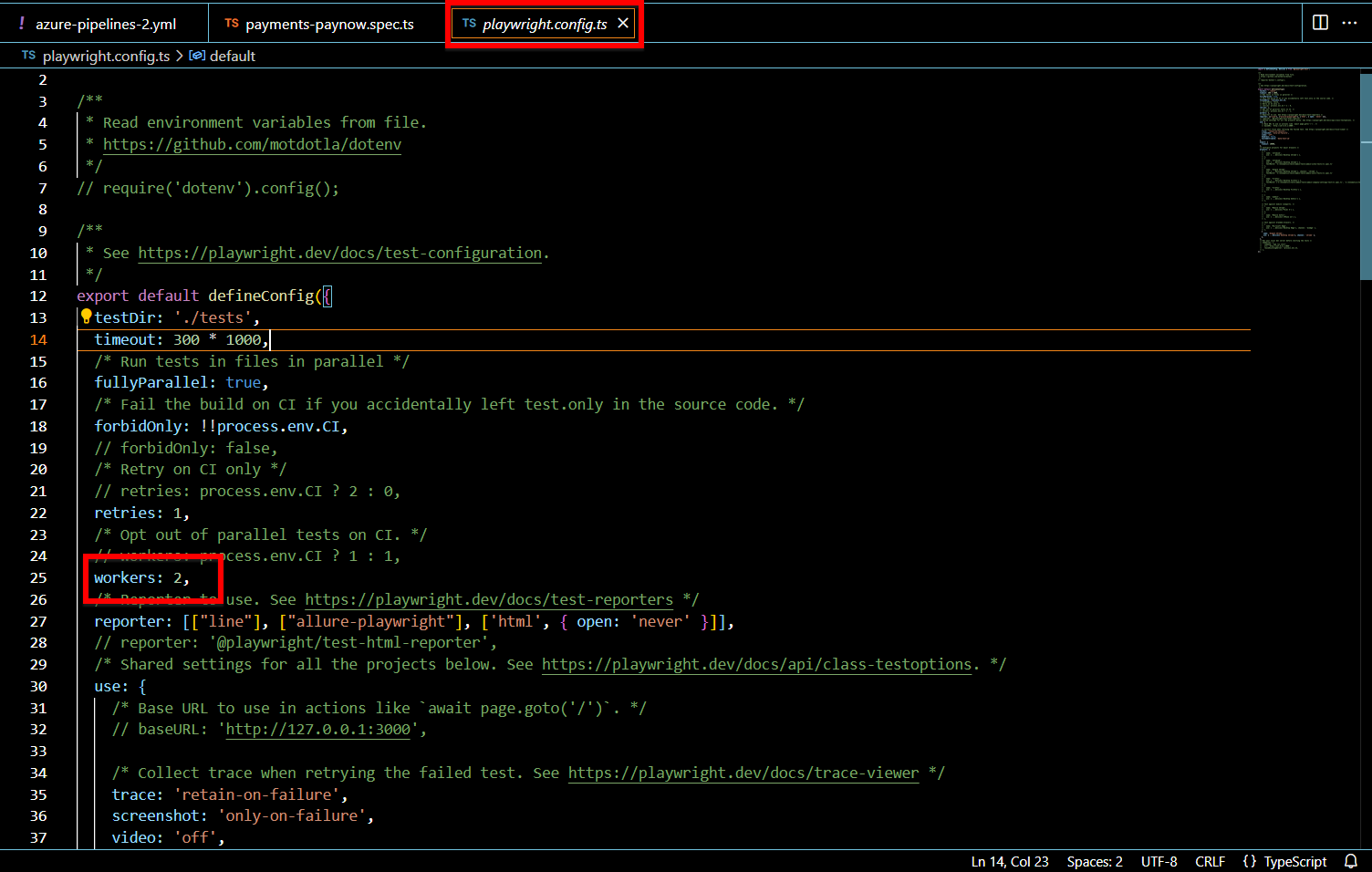
In above example we have imported the data **from ‘../../../testdata/payments/payments-payments. Json’** and now let’s say if we want to use a value of ‘url’ from **payments-payments.json** then we use it a following way **‘data['TC\_Pay\_001'].url’** and so on for other key-values from JSON file.

1. **Parallel Execution and Cross Browser Execution**

**A. Parallel Execution:**

Running tests in parallel and performing cross-browser testing in Playwright with TypeScript can significantly improve the speed and coverage of your test execution. Here's how we can achieve this:

Modify the **‘playwright.config.ts’** file to specify the number of parallel workers you want to use. The workers option controls parallel execution. For example, if we want to run our test cases in 2 parallel browsers, then we should use **“workers: 2,”** in the **‘playwright.config.ts’** file.



In the above example, two parallel workers are configured. Adjust the number to match your requirements and available resources. Now once the configuration is set run the test cases using the following command:

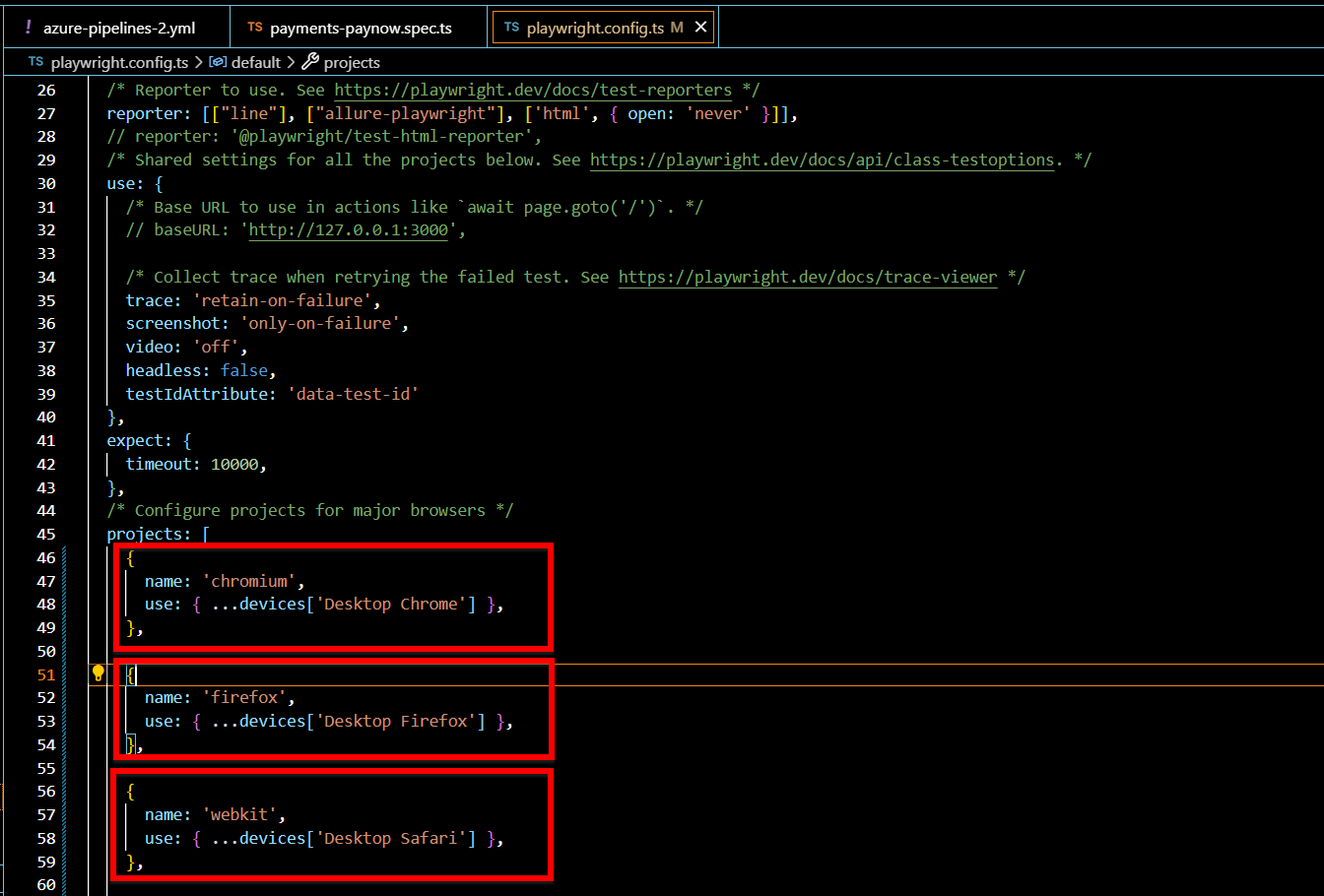
**npx playwright test**

**B. Cross Browser Execution:**

Cross-browser testing in Playwright allows you to test your web application on different browsers like Chromium, Firefox, and WebKit. Here's how to perform cross-browser testing:

In your playwright.config.ts file, define separate **projects** for each browser you want to test. Each project should have its configuration settings, such as browser, devices, and context options.

Here's an example of a playwright.config.ts file with separate projects for **Chromium, Firefox, and WebKit**:



Now as the projects are set for each browser and if you run the command to run the playwright test cases, All the test cases will get executed in each of the browser one by one using the number of workers you set.

**npx playwright test**

1. **Reporting**

In Playwright/test with TypeScript, test results are reported in a clear and structured manner, making it easy to understand the outcome of test runs. Playwright/test provides built-in reporting capabilities, and it also integrates with popular test reporting frameworks like allure reports. Here's how test results are reported in our framework using the playwright **‘index.html’** report.

**Step 1: Run your test cases using following command**

**npx playwright test**

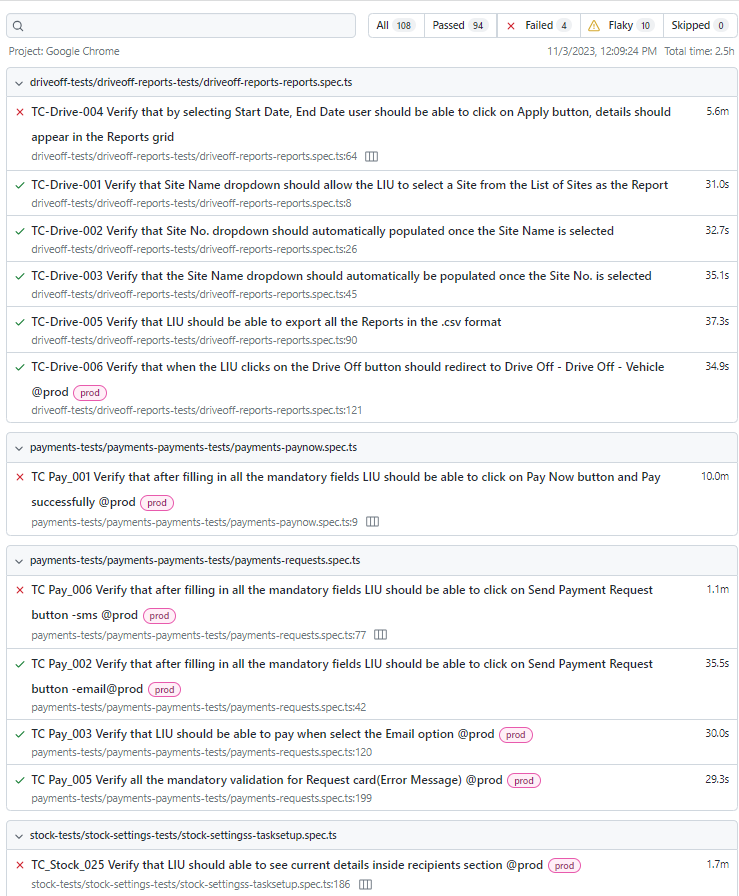
**Step 2: To view the html report created in ‘playwright-report’ folder**

There are two ways to view the html report –

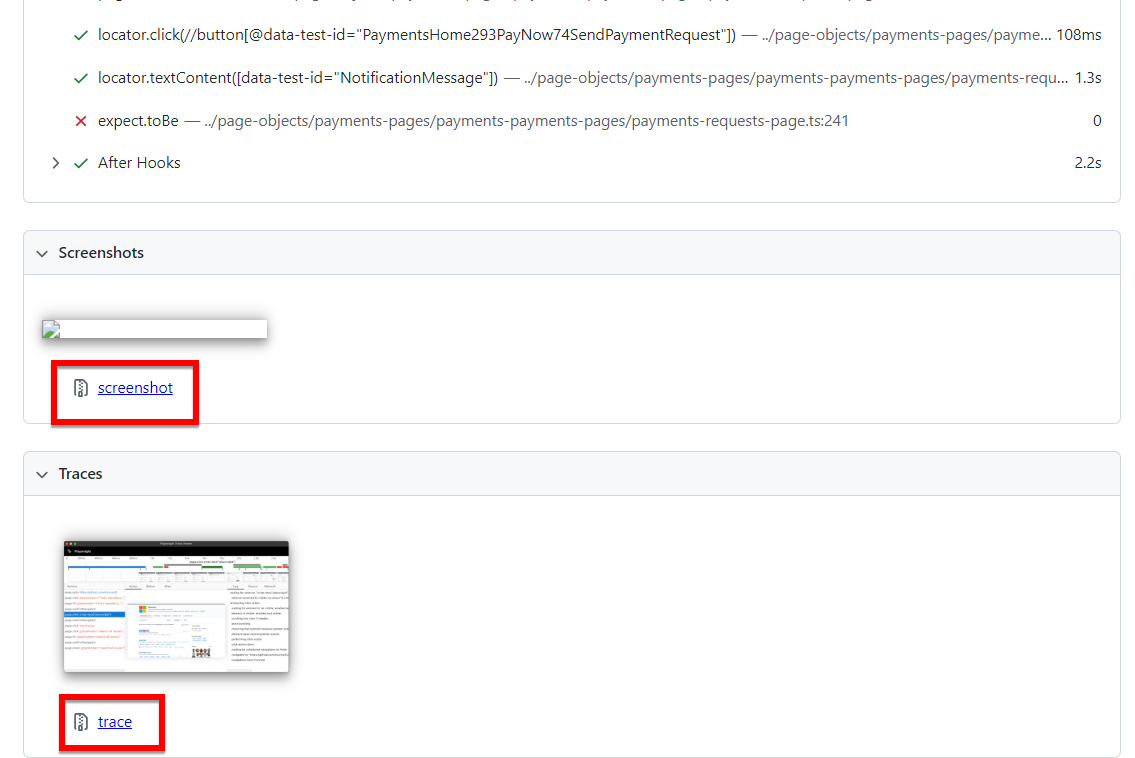
1. Go to playwright-report folder and open the **‘index.html’** file in any of the browser of your choice
2. Run following command to see the html report

**npx playwright show-report**

Once entered the above command an html report will get open in the browser with all the details like, The steps and Locator actions. All the executed test cases will be categorised as Passed, Failed, Flaky, Skipped test cases according to the status of their execution.



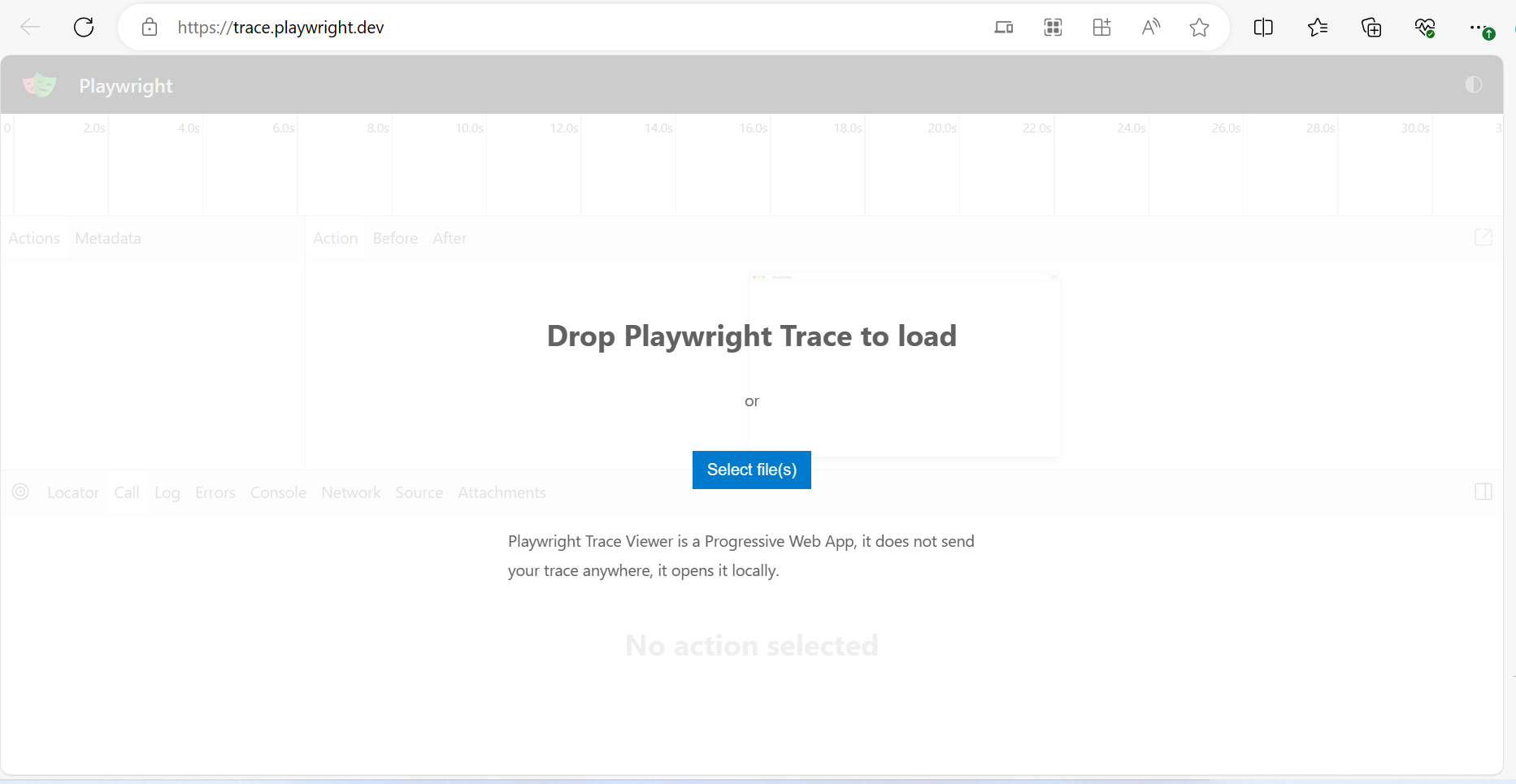
This is how the html report will look like. You can click on each test cases and see more detailed report of a particular test case with step-by-step execution, The time taken by each test to execute and at which step a particular test got failed and the reason for its failure. With the screenshot of the web page when the test got failed.

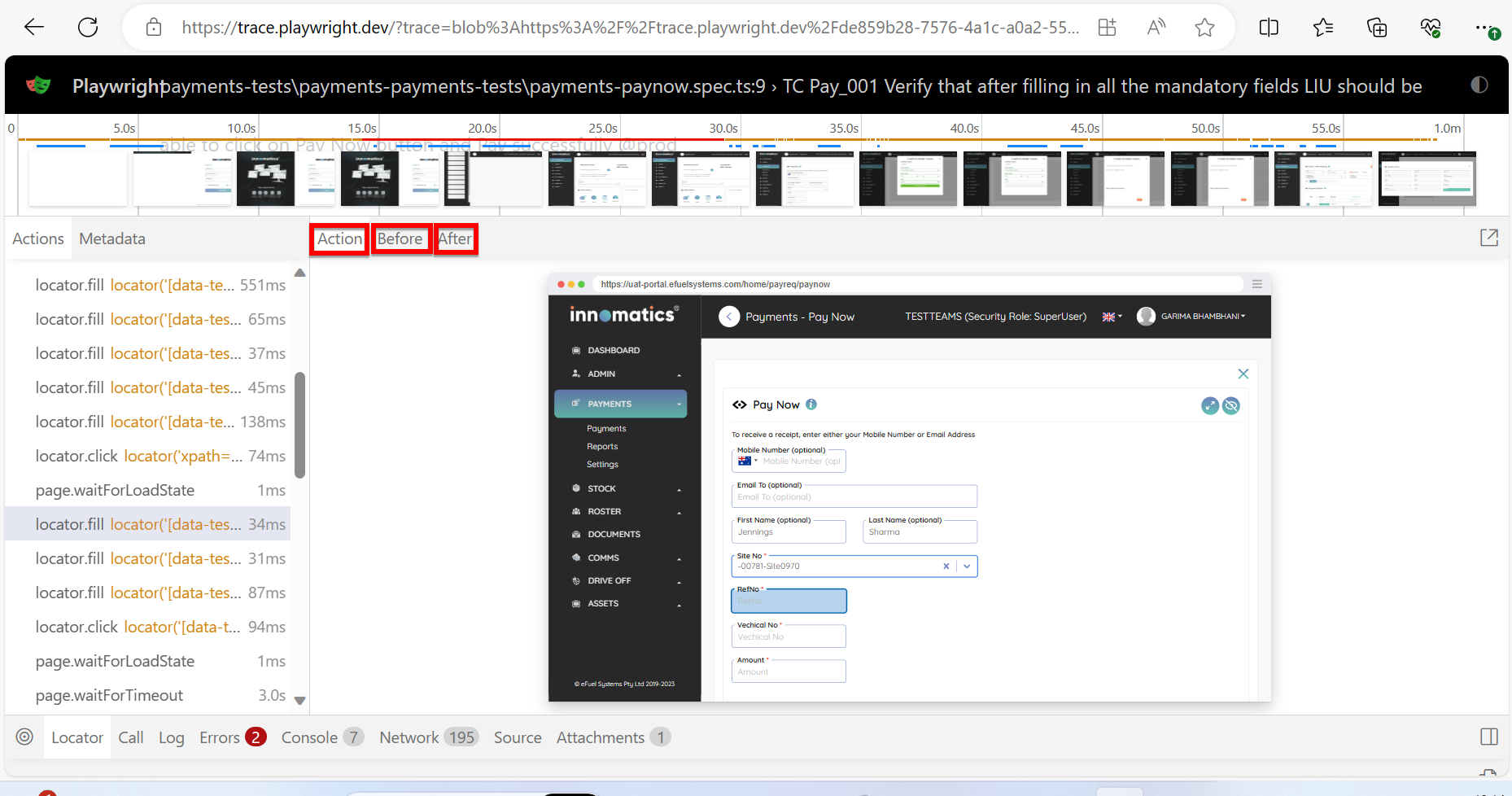
**Playwright trace viewer:**

Playwright comes with an interesting feature called **‘trace’.** Playwright trace allows you to capture detailed logs and diagnostic information during the execution of browser automation scripts. These logs are helpful for debugging, analysing performance, and understanding what is happening within the browser during script execution. When we click on a particular test case in the html report, we can see the trace. Click on the trace file and the ‘**trace.zip’** file will get downloaded.

Now go to: [**https://trace.playwright.dev**](https://trace.playwright.dev)



This is how the UI of [**https://trace.playwright.dev**](https://trace.playwright.dev)will look like, you can drag and drop your playwright trace file here or click on **‘Select file(s)’** button to select one or multiple Trace files to see the traces and debug the test case. Once you drag the trace file here, the traces will look as follows.

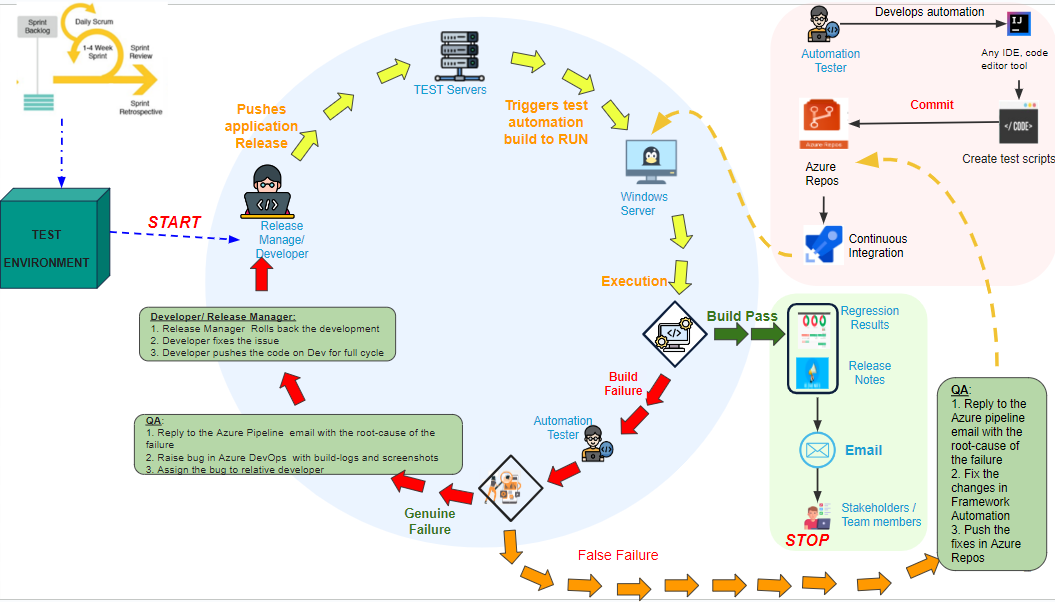


Playwright trace captures various types of information, including network activity, browser events, console messages, and more. It records all the events and actions that occur within the browser during the script's execution. On each step you can see what happens on the web page Before and After the particular step is executed. All the screenshots are on each step are taken which helps in debugging.

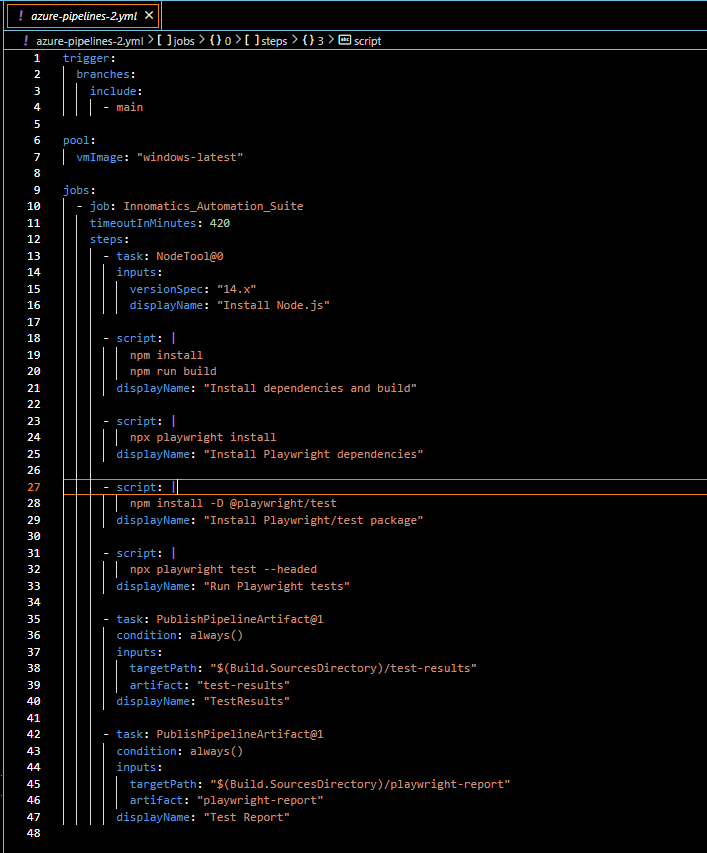
1. **Continuous Integration (CI)**

Continuous Integration (CI) and Continuous Delivery/Continuous Deployment (CD) are essential practices in modern software development. Integrating your Playwright automation framework with CI/CD pipelines ensures that your tests are automatically executed and validated in various environments, allowing for faster feedback and improved software quality.

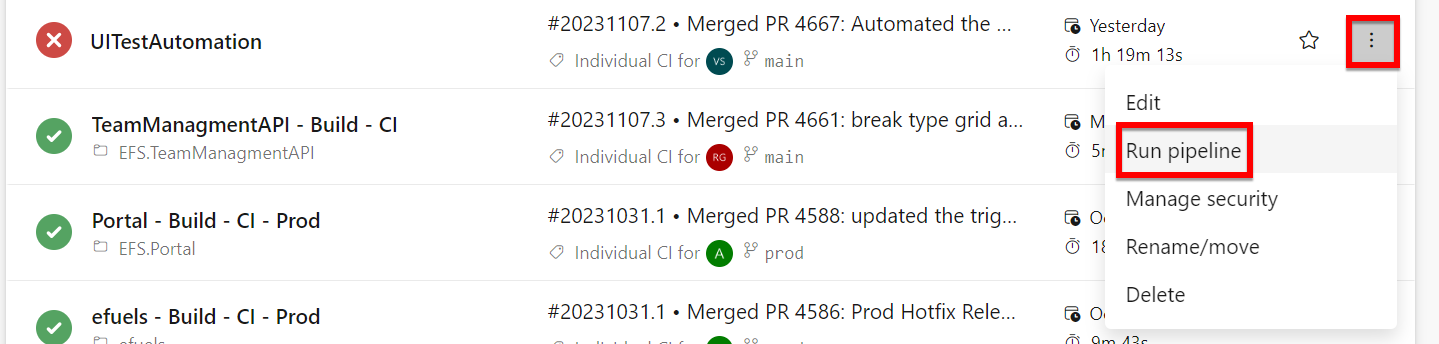
Following is the architecture that we followed for continuous integration and continuous deployment.



We trigger this pipeline with the help of YAML file – **‘azure-pipelines-2.yml’ –**

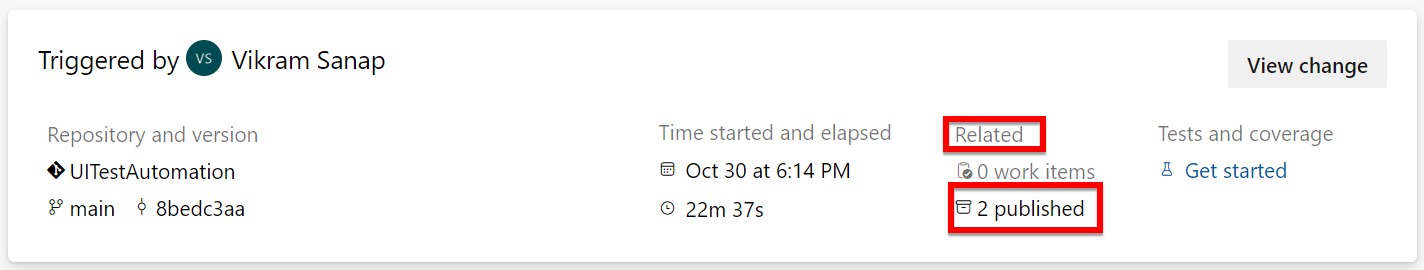


By default, we have triggered **‘main’** branch for the execution in above YAML file. In case if we want some other branch to trigger, we just need to replace main branch with that particular branch in the trigger section of the **‘azure-pipelines-2.yml’** file. Whenever we create a pull request and complete the merge request and code in the main branch gets updated, it automatically triggers the pipeline. We can also manually trigger the pipeline whenever required –



**Reporting in CI/CD:**

Once the pipeline ran successfully, the important part is to see the reports and test results, The report and test results have been published in the related section of a particular build –

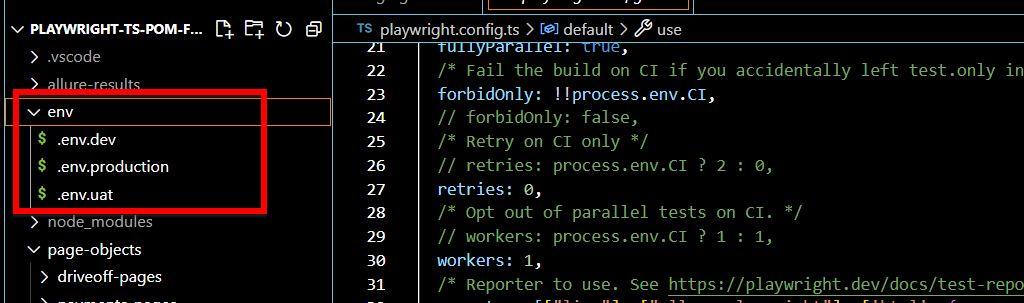


On clicking ‘2 published’ we can see two folders – 1) playwright-report and 2) test-results. Playwright-report folder contains **‘index.html’** report which can be downloaded and opened in any of the browser to analyse the report.

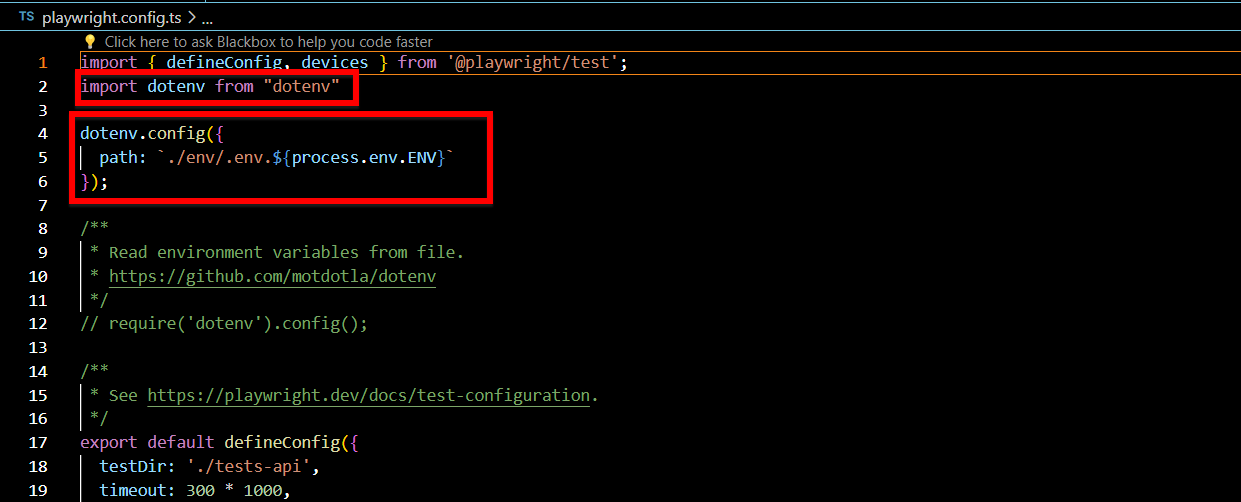
1. **Execution in different Environments**
2. To run our test cases in different environments we have used ‘dotenv’ npm package.

npm install dotenv

1. We have ‘env’ folder at root level which contains separate files for each environment: .env.uat, .env.production, .env.dev.



1. Load variables in playwright.config.ts (using import and config()):



1. Before executing the test cases, set the environment variable inn command prompt using following command and then run your test cases. If no environment is selected, uat environment is default.

set ENV=uat (or production or dev)

1. Run your test cases:

npx playwright test

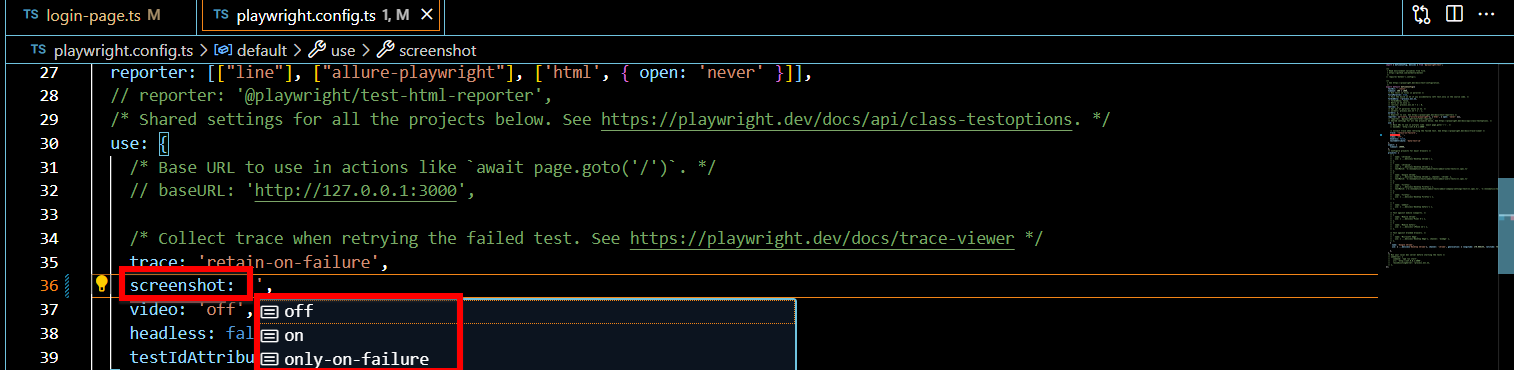
**IMPORTANT NOTE: Always make sure that both above commands should be in the same command prompt window.**

1. **Debugging and Troubleshooting**

Debugging test failures is an essential skill in automated testing with Playwright. Playwright provides various ways through which you can debug your failed test cases. Following are some of the techniques to debug your failed test cases.

**Debugging Techniques for Test Failures:**

1. **Failure Screenshots:**

Capturing screenshots when a test fails, is a powerful debugging technique. Screenshots provide visual evidence of test failures, making it easier to understand what went wrong. Our framework is configured to automatically capture screenshots when a test fails. We configure the screenshot taking mechanism from **‘playwright.config.ts’**

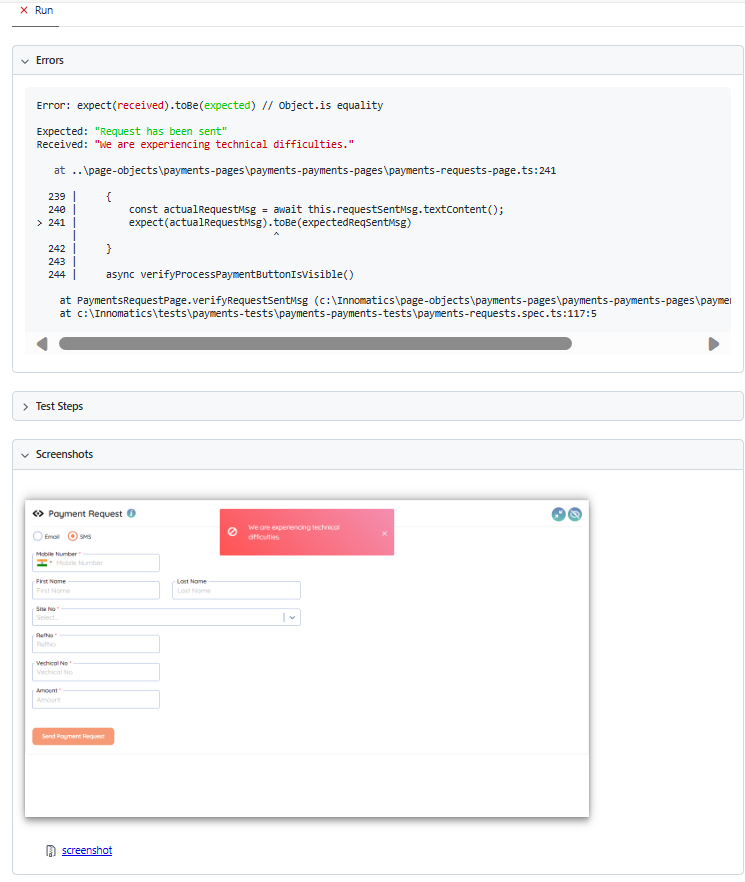
There are three options to configure the screenshots- off, on and only-on-failure.

**screenshot: ‘off’** – Will never take a screenshot

**screenshot: ‘on’** – Will always take a screenshot

**screenshot: ‘only-on-failure’** – Will take screenshot only when test fails.

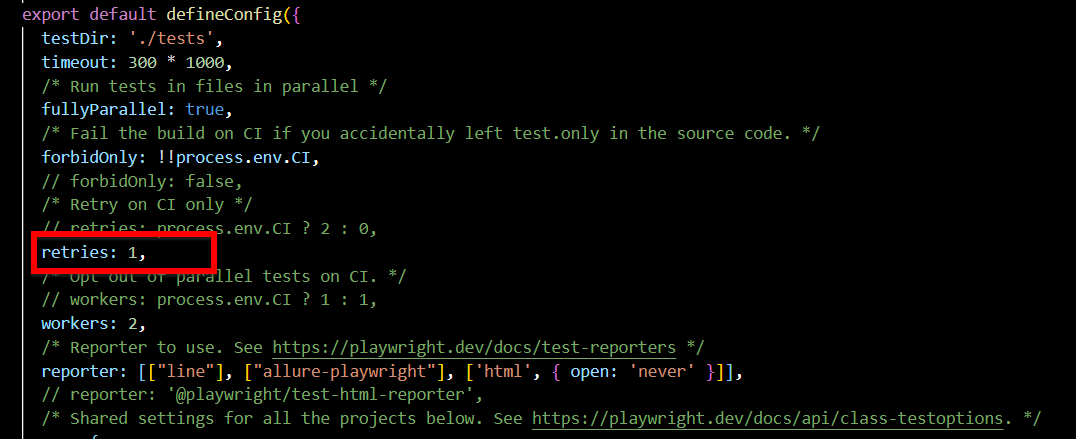
The screenshot will be stored in the test-results folder as well as in the ‘playwright-report/data’ folder and will be logged in the html report as well.



1. **Retries and Waits:**

Handling intermittent issues with retries and waits is an important debugging technique. Elaborate on how users can strategically use these features to address flaky tests:

**Retries:** Playwright/test allows you to specify retry logic for tests before marking it as fail. If an action fails, it can be retried a certain number of times before marking the test as failed. This can be achieved from the **‘playwright.config.ts’** file – you can retry your test any number of times if it fails before marking it as ‘Failed’. If your test gets passed in second or third attempt, this test case will be marked as **‘Flaky’** in the html report.



1. **Playwright Inspector:**

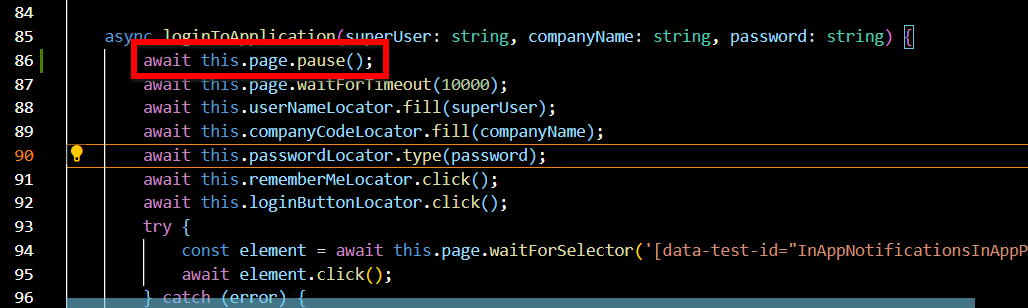
The Playwright Inspector is a valuable tool for interactive debugging. In the Inspector, you can:

**Inspect Elements**: Hover over elements on the page to highlight them and see their properties.

**Console Logs**: Examine console logs and network requests.

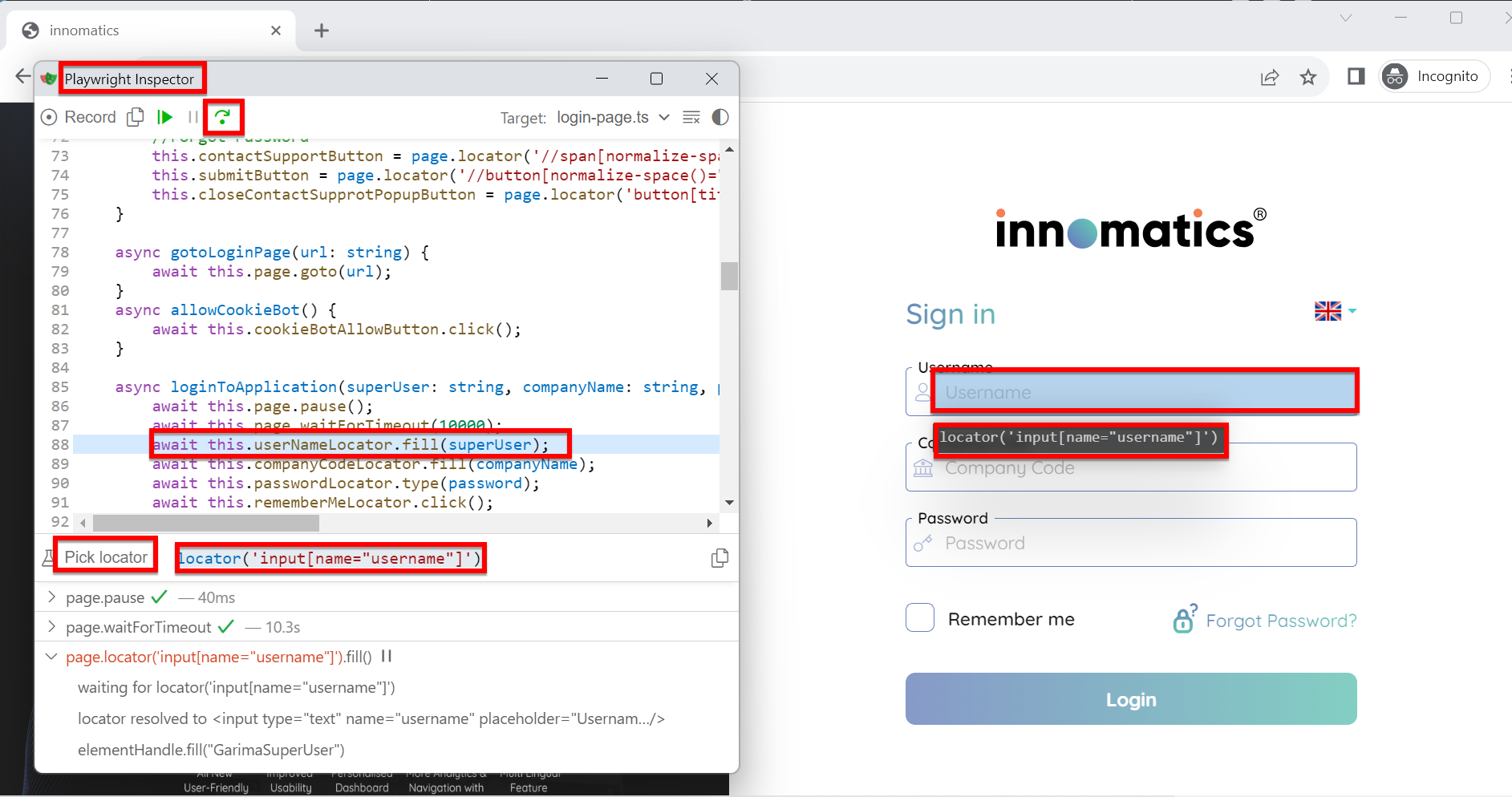
**How to open Playwright Inspector:**

Go to your code where you are facing a failure or you want to debug. Just before the line of code where we are facing a failure or exception, write – ‘await this.page.pause();’.



Now when you run your script, during the execution the playwright inspector will get open and the execution will get paused for you to debug the code. You can now inspect elements, see actions highlighted before they are executed.

Click on the ‘next’ button on top left in the playwright inspector to go to the next step in your code. It will highlight the element on web page on which the action is going to happen along with showing the element locator.



Playwright Inspector will also show the part of your code in execution highlighting the current line of code being executed. Following is the screenshot of the playwright inspector in action on the login page of our framework.

1. **Conclusion**

**Summary:**

Congratulations! You've reached the end of our comprehensive documentation for the Playwright with TypeScript Page Object Model (POM) framework. We've covered everything you need to know to harness the power of automated testing and achieve better test coverage, reliability, and efficiency.

You've learned how to create page objects, write clean and maintainable tests, handle data-driven testing, and integrate your tests into CI/CD pipelines. We've also discussed debugging techniques, reporting, and parallel execution. Armed with this knowledge, you're ready to take your testing efforts to the next level.

**Encouragement to Get Started:**

Now is the perfect time to dive in and start using our framework. With the support of our community, a wealth of resources, and the benefits of automation on your side, there's no limit to what you can achieve.

**Call to Action:**

1. Visit our [**Azure repository**](https://efuelsystems@dev.azure.com/efuelsystems/efuels/_git/UITestAutomation) to access the latest code and documentation.
2. Install the necessary dependencies using the following command:

**npm install.**

1. Write your first test case and experience the magic of automated testing with Playwright.

We're excited to see what you can achieve with our framework. Happy testing!