**Playwright**

* Playwright is a powerful end-to-end testing framework that enables reliable automation for modern web applications.
* **Why Playwright?**

1. **Reliable End-to-end Testing:** 
   * Playwrigth’s Auto-wait capability ensures reliable and stable end-to-end testing for modern web applications, even in the face of dynamic and complex user interactions.
   * [**Auto-waiting:**](https://playwright.dev/docs/actionability#:~:text=On%20this%20page-,Auto%2Dwaiting,-Introduction%E2%80%8B)

The playwright performs a range of actionability checks on the elements before making actions to ensure these actions behave as expected. It auto-waits for all the relevant checks to pass and only then performs the requested actions. If required checks do not pass within the given **timeout**, action fails with the **TimeoutError.**

* + Before execution for that element playwright check some conditions are eligible on that element or not like it visible, stable or enable or not

**A screenshot of a computer screen

Description automatically generated**

1. **Cross-Browser Compatibility:**
   * Playwright supports all major browsers, including Chrome, Edge, Firefox, Safari and Opera, allowing you to test your web applications across a wide range of browsers and platforms
2. **Multiplatform Support:**
   * Playwright works seamlessly on Windows, macOS, and Linux, and also supports native mobile emulation for Google chrome on Android and Safari on iOS, enabling comprehensive testing across different devices and operating systems.
3. **Multilingual Flexibility:**
   * Playwright provides language binding for JavaScript, TypeScript, Java, Python and C# allowing you to choose the programming language that best fits your team’s preferences and expertise.

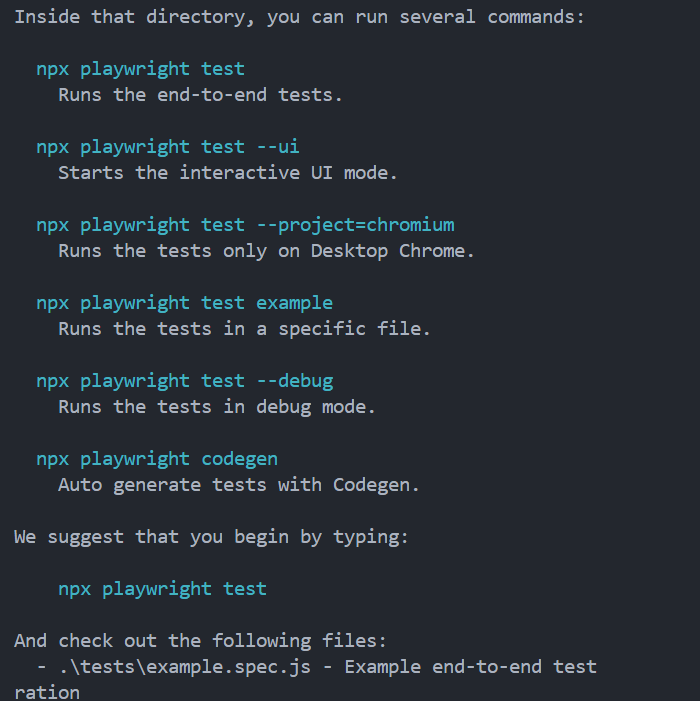
* **Playwright’s Advanced Features:**

1. **Tracing and Debugging:**
   * Playwright has built-in tracing and debugging capabilities, including automatic screenshots, test video recording and comprehensive logging, to simplify the process of identifying and resolving issues in your test suite.
2. **Network Interception:**
   * Utilize playwright’s API testing libraries to intercept and validate network calls within your web application, enabling you to test edge case scenarios and ensure the resilience of your application’s network interaction.
3. **Browser Context Management:**
   * Explore playwright’s browser context feature, which allows you to save and transfer browser state across your test suite, improving test efficiency and reducing the overhead of setting up the same browser state for each test case.
4. **Codegen Tool:**
   * It can generate code by recording your actions, saving your time and effort in creating initial test case and providing a starting point for further customization.

* **To create playwright project, we use following command**

**npm init playwright**

* + This command is creating playwright structure project with all playwright dependencies
  + **Playwright.config.json** file is important file which is also called as playwright test runner file



* **Sample test cases:**

A computer screen shot of a program code

Description automatically generated

We can manually create **page** using **browser** fixture or we can directly use **page** fixture to goto URL.

* To set which browser should execute we should mention in Playwright.config.json file.

A black background with red and green text

Description automatically generated

* To execute playwright project we use following command

**npx playwright test**

* Playwright by default run in headless mode means we won’t see anything of what is happening in the browser when our script runs. If we want to see browser execution then we should run in **headed** mode using following command

**npx playwright test –headed**

**or**

add in configuration file

A computer screen shot of a computer code

Description automatically generated

* To generate test report:

**npx playwright show-report**

* To execute in debug mode:

**npx playwright test --debug**

* To verify title of webpage we have assertion

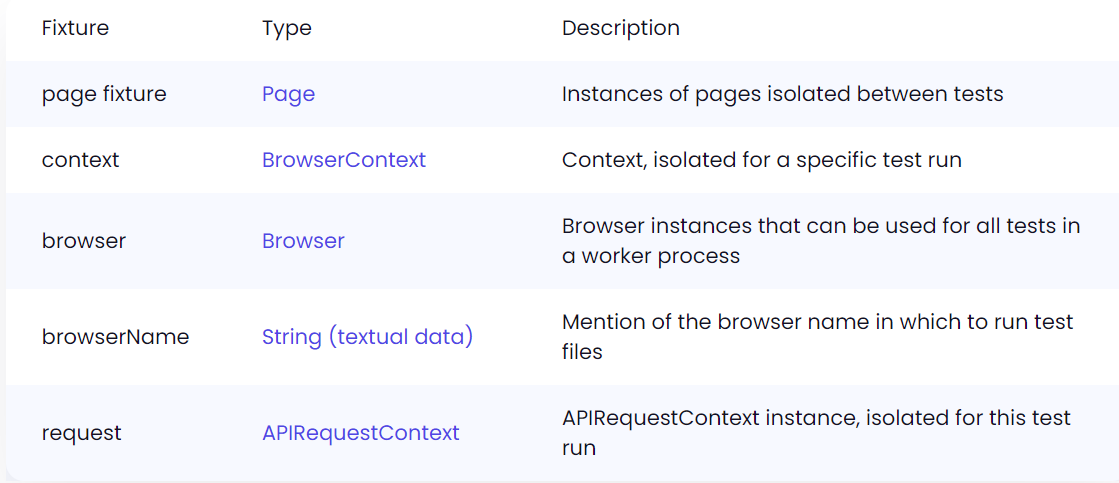


It will check and verify title of webpage if match then **passes** otherwise **failed**

* **Note: webkit** is a default playwright specific browser engine.
* **Fixtures:** [**🔗**](https://testomat.io/blog/what-is-the-use-of-fixtures-in-playwright/)
  + Playwright commonly uses with the **Page Object Model,** which is one of the approaches to structuring tests to optimize their speed and efficiency. In the framework under consideration, fixtures simplify grouping tests and are considered a way manage them as the page object method.
  + Test fixtures are preliminary conditions or steps that are executed before running a test. The testing concept of fixtures is based on their use – existing fixtures create a precise environment for each test, avoiding anything unnecessary.
  + In playwright, test fixtures allow you to reuse code for different test cases. In essence, a fixture is a function that wraps the inheritance of classes. It’s a convenient way to encapsulate your testing functionality and its data in separate blocks and call it when needed.

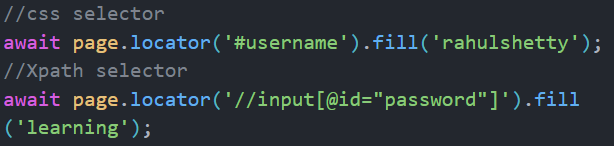
****

* + Below, we present a list of fixtures used when writing a base test in playwright:

****

* **Different types of methods in Playwright:**

1. **.fil():** 
   * **.fill()** method is use to fill/pass data in input filled

****

1. **.pressSequencially():**
   * Type into the field character by character, as if it was a user with a real keyboard with **locator.pressSequentially().**

**A black background with white text

Description automatically generated**

* + This method will emit all the necessary keyboard events, with all the keydown, keyup, keypress event in place. You can even specify the optional delay between the key presses to simulate real user behavior.

1. **.click():**
   * **.click()** method is used to perform click event on elements like button.

****

1. **.textContent():**
   * **.textContent()** method is used to extract text from element

****

1. **. toContainsText():**
   * **.toContainsText()** check given text available in there or not



It will check given text available in that located element or not that’s basis it will passes or failed the case

This method is assertion method.

1. **.waitFor():**
   * **.waitFor()** method is primarily used with locators to paused the execution until a specific condition is met on a web page element.
   * It helps in synchronizing your test scripts with dynamic content changes.

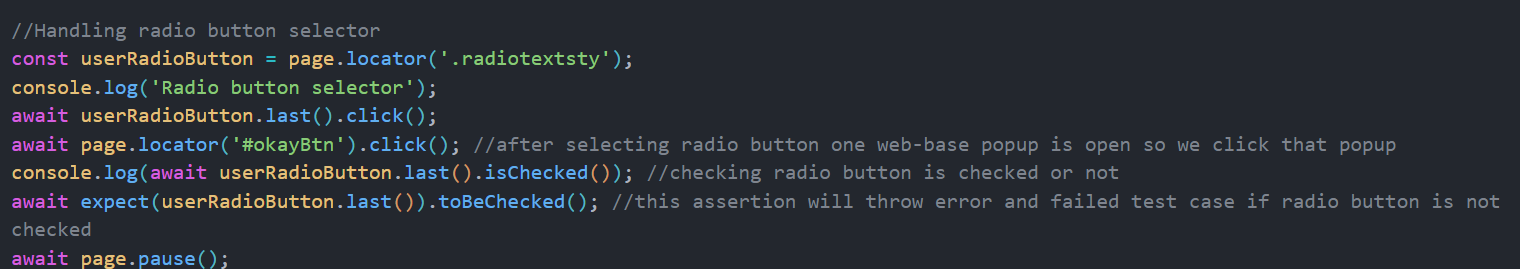
****

* + This method works on single element if multiple element is detect it won’t work. If we want to work we should select only one element by using **.first() / .last()** function.

1. **.selectOption():**
   * **.selectOption()** method is used to select option in dropdown menu

****

1. **Handling Radio button and popup:**

****

* + **.toBeChecked()** assertion is passed case if radio button is checked otherwise it failed this test case.
  + **.isChecked()** method is check button is checked or not checked.

1. **Handling checkbox:**

**A screen shot of a computer code

Description automatically generated**

* + **.toBeFalsy()** check given condition return falsy value or not. It failed test case when condition return **true**
  + **.toBeTruthy()** method failed test case if condition return **false.**
  + **.uncheck()** method is used to uncheck a button/checkbox

1. **Handle child window:**

**A screen shot of a computer code

Description automatically generated**

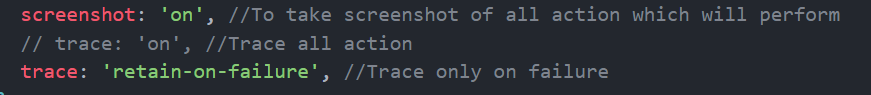
* + **Promise.all([])** create array of new pages by executing conditions written inside it
  + **.waitForEvent()** this is browser context method which can be used to wait for a specific event to occur on a page. You can use this method to wait for an event, such as a DOM element being clicked, a form being submitted, a page is loaded, or a network request being made.

1. **Generate code use Codegen tool:**

**npx playwright codegen url**

* + This command is used to open link in recording mode.

1. **To generate screenshot and trace report:**

****

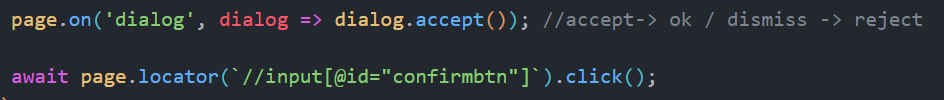
* + Add this two properties in **playwright.config.json** file in use object

1. **To go back to previous page or go to next web page:**

**A screenshot of a computer program

Description automatically generated**

1. **To handle popup/alert/dialog in playwright”**

****

* + There is **page.on()** method which activated when anywhere popup/dialog box is generated in web page.
  + It get two parameters which **event** and **callback function** which has **accept and dismiss** method

1. **To handle hover:**

****

* + There is separate **hover()** method to handle hover in playwright
* **Playwright Inbuilt Locators:**

1. **.getByLabel():**
   * It is used to locate the control by its associated label using **page.getByLabel()**

**A screen shot of a computer program

Description automatically generated**

* + Use this locators when locating form field.

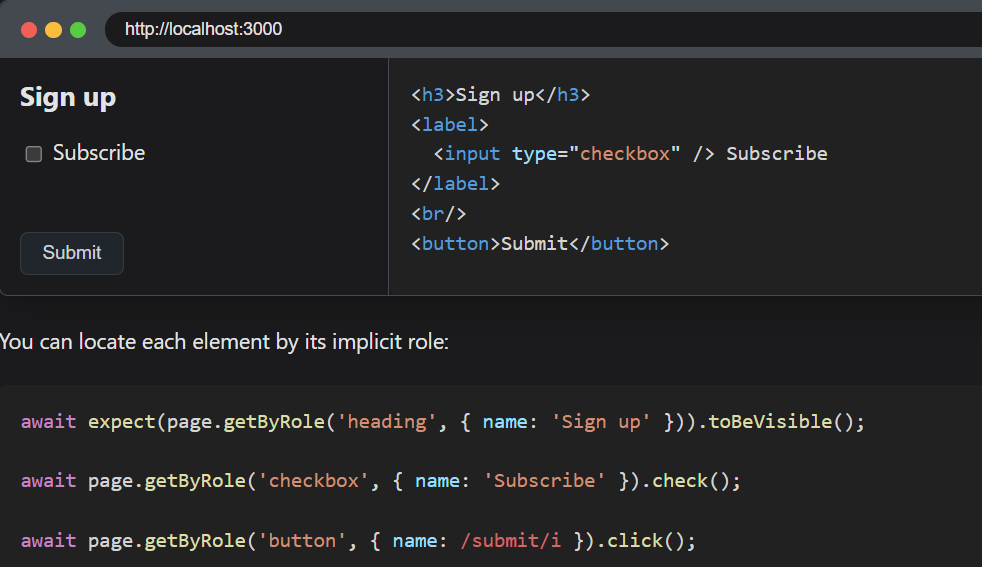
1. **.getByPlaceholder():**
   * Input may have a placeholder attribute to hint to the user what value should be entered. You can locate such an input using **page.getByPlaceholder().**

**A screenshot of a computer program

Description automatically generated**

* + Use this locator when locating form elements that do not have labels but have placeholder texts.

1. **.getByRole():**
   * The **page.getByRole()** locator reflects how users and assistive technology perceive the page, for example whether some element is a button or a checkbox. When locating by role, you should usually pass the accessible name as well, so that the locator pinpoints the exact element.

****

* + Role locators include buttons, checkboxes, headings, links, lists, tables and may more elements.
  + Note that many html elements have an implicitly defined role that is recognized by the role locator.

1. **.getByText():**
   * Find an element by the text it contains. You can match by a substring, exact string, or a regular expression when using **page.getByText().**

****

* + Matching by text always normalizes whitespace

1. **.getByTitle():**
   * Locate an element with a matching title attribute using **page.getByTitle();**

**A screenshot of a computer program

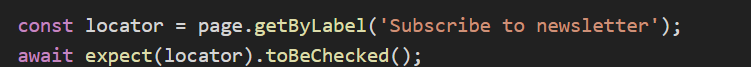
Description automatically generated**

* **Assertions:** [**🔗**](https://playwright.dev/docs/test-assertions#:~:text=On%20this%20page-,Assertions,-Introduction%E2%80%8B)
  + Playwright includes test assertions in the form of **expect()** function. To make an assertion, call **expect(value)** and choose a matcher that reflects the expectation. There are many generic matchers like **toEqual, toContain, toBeTrutht** that can be used to assert any condition

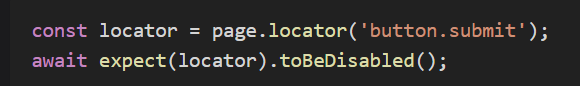
1. **.toBeAttached():**
   * Ensure that locator points to an element that is connected to a Document or a shadowRoot

Eg: **await expect(page.getByText('Hidden text')).toBeAttached();**

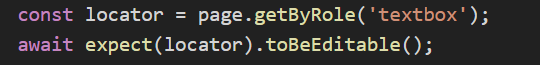
1. **.toBeChecked():**
   * Ensure the locator points to checked input

****

1. **.toBeDisable():**
   * Ensure the locator point to a disable element. Element is disabled if it has “**disabled**” attribute or is disabled via **‘aria-disabled’.**
   * Note that only native control element such as HTML button, input, select, textarea, option, optgroup can be disabled by setting **‘disabled’** attribute.
   * **Disabled** attribute on other elements is ignored by the browser.

****

1. **.toBeEditable():**
   * Ensure the Locator points to an editable element.

****

1. **.toBeVisible():**
   * Ensure that locator points to an **attached** and **visible** Dom node.
   * To check that at least one element from the list is visible, use **locator.first()**

**A computer screen shot of text

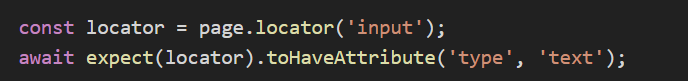
Description automatically generated**

1. **.toContainText():** [**🔗**](https://playwright.dev/docs/api/class-locatorassertions#locator-assertions-to-contain-text:~:text=Added%20in%3A%20v1.20-,Ensures,-the%20Locator%20points%20to%20an%20element%20that)
   * Ensure the Locators points to an element that contains the given text. All nested elements will be considered when computing the text content of the element.
   * You can use regular expression for the value as well.

**A screen shot of a computer

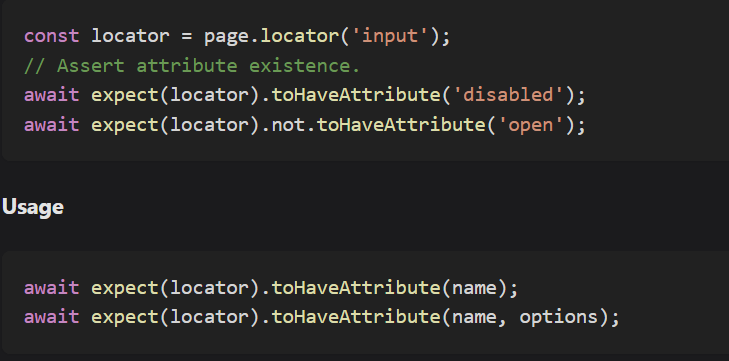
Description automatically generated**

1. **.toHaveAttribute(name,value):**
   * Ensure the locator points to an element with given attributes.

****

* + **name:** attribute name
  + **value:** Expected attribute value

1. **.toHaveAttribute(name):**
   * Ensure the locator points to an element with given attribute.
   * The method will assert attribute presence.

****

1. **.toBe():**
   * Compares value with expected by calling **object.is.**
   * This method compares objects by reference instead of their content, similarly to the strict equality operator **===.**

**A screen shot of a computer code

Description automatically generated**

1. **.toBeFalsy():**
   * Ensure that value is false in Boolean context, one of **false, 0, ‘’, null, undefined and NaN.**
   * Use this method when you don’t care about the specific value

**A black background with white text

Description automatically generated**

1. **.toBeTruthy():**
   * Ensure that value is true in a Boolean context, **anything but false, 0, ‘’, null, undefined or NaN.**
   * Use this method when you don’t care about the specific value

**A black background with white text

Description automatically generated**

1. **.toEqual():**
   * Compares content of the value with contents of **expected,** performing **‘deep equality’** check.
   * For objects, this method recursively checks quality of all fields, rather than comparing objects by reference as performed by **expect(value).toBe().**
   * For primitive values, this method is equivalent to **expect(value).toBe().**

**A black background with white text

Description automatically generated**

* [**Playwright Test:**](https://playwright.dev/docs/api/class-test#test-call:~:text=Playwright%20Test-,Playwright%20Test,-Playwright%20Test%20provides)
  + Playwright Test provides a **test** function to declare tests and **expect** function to write assertions

****

* + There are different hooks on test function.

1. **test.afterAll():**
   * Declare an **afterAll** hook that is executed once per worker after all tests.
   * When called in the scope of a test file, runs after all tests in the file,
   * When called inside a **test.describe()** group, runs after all tests in the group.

**A screenshot of a computer program

Description automatically generated**

1. **test.afterEach():**
   * Declares an **afterEach** hook that is executed after each test.
   * When called in the scope of a test file, runs after each test in the file.
   * When called inside a **test.describe()** group, runs after each test in the group.

**A computer screen shot of a program code

Description automatically generated**

1. **test.beforeAll():**
   * Declares a **beforeAll** hook that is executed once per worker process before all tests.
   * When called in scope of a test file, runs before all tests in the file
   * When called inside a **test.describe()** group, runs before all tests in the group.

**A computer screen shot of text

Description automatically generated**

* + We can use this for login, session creation

1. **test.beforeEach():**
   * Declares an **beforeEach** hook that is executed after each test.
   * When called in the scope of a test file, runs before each test in the file.
   * When called inside a **test.describe()** group, runs before each test in the group.
   * You can access al the same **fixtures** as the test body itself, and also the **TestInfo** object that gives a lot of useful information.
   * Eg. You can navigate the page before starting the test.

**A computer screen shot of text

Description automatically generated**

1. **test.describe():**
   * Declare a group of tests.
   * Syntax:

**A screenshot of a computer screen

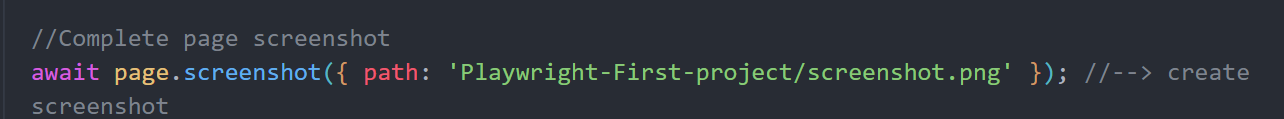
Description automatically generated**

* + You can declare a group of tests with a title. The title will be visible in the test report as a part of each test’s title.

**A computer code with text

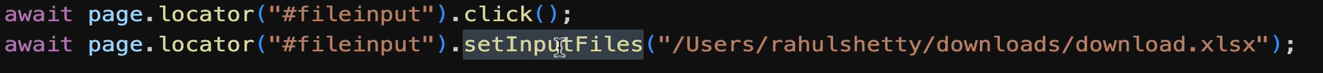
Description automatically generated with medium confidence**

* **Page.addInitScript():**
  + Add a Script which would be evaluated in one of the following scenarios:
    - Whenever the page is navigated.
    - Whenever the child frame is attached or navigated. In this case the script is evaluated in the context of the newly attached frame.
  + The script is evaluated after the document was created but before any of its scripts were run. This is useful to amend the JavaScript environment, Eg. To seed Math.random
* **To take screenshot:**
  + **Locator level screenshot:  
    A screen shot of a computer

    Description automatically generated**
  + **Complete page screenshot:  
    **
* **Perform Visual test by matching/comparing screenshots of ui:**

**A computer screen shot of a program code

Description automatically generated**

* + **.toMatchSnapshot()** assertion is first create a snapshot if not present
  + If snapshot already present it match that snapshot and compare each element are present in same position with previous snapshot.
  + If any changes occurs then it will failed the test case and throws a error.
* **To upload file using playwright:  
  **
  + **.setInputFiles()** method is use to upload files using playwright.
  + But there is a condition, for applying this method our html element should have **input type=”file”.**
* **Playwright Framework Features:**
  + Implement Page Object Pattern for web automation tests
  + Create API utilities for Data setup and clean up.
  + How to drive test data from external files for playwright tests
  + How to parameterize the test with different data sets in playwright.
  + Create project configuration with playwright config file
  + Tests retry feature in playwright and how it works
  + How to record the video of test execution within playwright framework
  + Understand how tests run in parallel mode in playwright framework
  + Test automation on run, skip, serial run and parallel run
  + Playwright command line arguments options to run tests
  + Detailed view of HTML reporting with trace viewer logs
  + Generating all your reporting for playwright execution results
  + How to create custom scripts in package.json to trigger tests
  + Integrate the playwright framework with Jenkins CI/CD jobs and schedule runs.
* **Page Object Model Pattern:**
  + Page Object Model is a design pattern that creates a repository for storing all web elements.
  + It is useful in reducing code duplication and improve test script maintenance.
  + In POM, 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.
  + Large test suites can be structured to optimized ease of authoring and maintenance. POM is one such approach to structure your test suite
  + Page objects simplify authoring by creating a higher-level API which suits your application and simplify maintenance by capturing element selectors in one place and create reusable code to avoid repetition.
  + **Advantages of Page Object Model:**

1. **Easy Maintenance:**

* POM makes maintenance easier even if there is a change in the DOM tree and selectors we don’t have to modify everywhere.

1. **Increased Reusability:**

* Using POM, we can reuse the code which can written for another test.
* Also, we can create helper method to achieve this.
* Code reusability reduce the code, thus saving time and effort

1. **Readability:**

* As the tests are independent, it increase the readability.
  + **Disadvantages of Page Object Model:**
    - Initial design and building framework take some time.
    - Good coding skills are required to set the POM framework
    - Elements are stored in a shared file, so even a tiny mistake in the page object file can lead to breaking the whole test suite.