Playwright Interview Questions:

A) Transitioning from Selenium to Playwright

Playwright Locators

Q1. Why did you choose Playwright over Selenium for testing?

APPROACH:

Highlight the advantages of Playwright over Selenium, considering factors like browser support, API simplicity, execution speed, and modern web application testing capabilities.

ANSWER:

Playwright was chosen over Selenium for several reasons, including its native support for all major browsers (Chrome, Firefox, Safari, and Edge) without needing separate drivers, faster execution of tests due to a more modern and efficient API, and better support for modern web applications using JavaScript frameworks. Playwright's ability to handle scenarios like single-page applications (SPAs), headless browser testing, and mobile viewports natively also contributed to the decision.

Q2. How did you evaluate Playwright against other automation solutions?

APPROACH:

Discuss the criteria used for evaluation, such as ease of use, community support, documentation, browser support, and specific features needed for testing modern web applications.

ANSWER:

The evaluation of Playwright against other automation solutions involved comparing documentation quality, community and developer support, ease of setup and use, browser support, and the ability to test modern web features like SPAs, network interception, and screenshot testing. Performance metrics, such as the speed of test execution and resource usage, were also considered.

Q3. Challenges faced during migration from Selenium to Playwright?

APPROACH:

Identify common challenges such as learning new syntax, adapting existing tests, and integrating with CI/CD pipelines.

ANSWER:

During the migration from Selenium to Playwright, challenges included adapting the team to the new syntax and APIs, rewriting existing tests to leverage Playwright's capabilities, ensuring compatibility with the existing CI/CD pipeline, and reconfiguring test environments to support Playwright's requirements for browser binaries.

Q4. Why implement Playwright in TypeScript?

APPROACH:

Discuss the benefits of using TypeScript, such as type safety, better development experience, and easier maintenance.

ANSWER:

Implementing Playwright in TypeScript offers several advantages, including enhanced code quality through type safety, improved developer productivity with better tooling and autocomplete, easier maintenance and refactoring of tests, and the ability to leverage modern JavaScript features while ensuring backward compatibility.

Q5. Differences between Java/Python and TypeScript in test automation?

APPROACH:

Compare Java and Python with TypeScript specifically in the context of syntax, type safety, and handling asynchronous operations within test automation projects.

ANSWER:

Syntax: Java's syntax is static and verbose, requiring explicit declarations, making test scripts structured but lengthy. Python's dynamic typing offers concise syntax for rapid script development, enhancing readability. TypeScript combines JavaScript's syntax with static typing, providing flexibility and succinctness.

Type Safety: Java has strong static typing, reducing runtime errors but requiring more declarations. Python's dynamic typing adds flexibility but risks runtime errors. TypeScript introduces static typing to JavaScript, enhancing script predictability and safety.

Async Handling: Java uses complex utilities like Future and CompletableFuture for asynchronous operations, which can be verbose. Python and TypeScript both support async/await, simplifying asynchronous code writing and making it more readable, beneficial for handling web interactions in tests.

Q6. Unique features of Playwright not found or hard to implement in Selenium?

APPROACH:

Highlight Playwright-specific features that enhance testing capabilities.

ANSWER:

Unique features of Playwright that distinguish it from Selenium include built-in support for multi-page scenarios, network request interception, a rich set of browser contexts for emulation of different devices, and native support for web components. Additionally, Playwright's automatic waiting mechanism ("auto-waits") simplifies interaction with dynamic content by implicitly waiting for elements to be ready before performing actions.

This eliminates the need for manual waits and polling, a common source of flakiness in Selenium tests. These capabilities are either not available in Selenium or require additional tools and configurations, making Playwright a more comprehensive solution for modern web application testing.

Q7. Describe the process and timeline for migrating your team to Playwright.

APPROACH:

Outline the steps for migration, including training, pilot testing, full migration, and integration into CI/CD.

ANSWER:

The migration process involved initial team training sessions on Playwright, followed by a pilot phase where key test suites were rewritten and tested in Playwright. After evaluating the pilot, a full migration plan was executed over several months, with continuous integration into our CI/CD pipeline. The entire process was aimed at minimizing disruption to our testing cycle.

Q8. Can you explain the Playwright architecture?

APPROACH:

Outline the framework's structure, emphasizing its components, interaction with browsers, and the role of browser contexts.

ANSWER:

Playwright is structured around a Node.js library that offers a high-level API for controlling both headless and GUI browsers through the DevTools Protocol. It communicates with browser-specific binaries for Chrome, Firefox, and WebKit, which enables direct interaction with the browsers for executing tests. **A** key component of its architecture is the concept of browser contexts, which are isolated environments within the same browser instance. These contexts allow Playwright to manage multiple pages, run tests in parallel, and emulate different devices or network conditions within the same test suite efficiently.

Browser contexts are central to Playwright's ability to provide test isolation, making it possible to simulate multiple sessions or users in a single test run without the overhead of launching multiple browsers.

B) Playwright Terminology and Concepts

Q9. What is a browser context in Playwright?

APPROACH:

Elucidate on the browser context's definition and expand on its practical applications within Playwright's testing framework.

ANSWER:

In Playwright, a browser context represents an isolated environment for web pages, essentially simulating an independent browser session. This isolation enables running parallel tests in separate contexts without the overlap of cookies, local storage, or session storage, thus supporting a variety of test scenarios efficiently. Use cases for browser contexts include testing multi-user experiences by simulating multiple logged-in users concurrently, checking the behavior of web applications across different device profiles or network conditions, and ensuring clean test states by resetting the environment before each test. Browser contexts are instrumental in facilitating scenarios like A/B testing, privacy settings validation, and feature flag testing, offering a robust mechanism for comprehensive web application testing.

Q10. Is it better to create a new browser context for each test or at the suite level?

APPROACH:

Compare the benefits of isolation versus efficiency when deciding on the scope of browser contexts.

ANSWER:

Creating a new browser context for each test ensures isolation and clean test environments, reducing test flakiness by avoiding shared state and side effects. However, for closely related tests or when initialization time is a concern, using a single context for a suite of tests can be more efficient.

Q11. How do you manage authentication states in Playwright?

APPROACH:

Discuss the mechanism for persisting and reusing authentication states across tests.

ANSWER:

To manage authentication states in Playwright, you can log in once and capture the authentication state (cookies, local storage) from the browser context. This state can then be programmatically set in new browser contexts before running subsequent tests, bypassing the need for logging in before each test.

Q12. Explain the hierarchy of browser, context, and page in Playwright.

APPROACH:

Outline the relationship and hierarchy between these core concepts in Playwright.

ANSWER:

In Playwright, the hierarchy starts with the browser, representing an instance of a browser application. Within a browser, you can create multiple browser contexts, each an isolated environment for pages, allowing for session and state management. Inside a context, you can open one or more pages, which are individual tabs or windows where tests are executed.

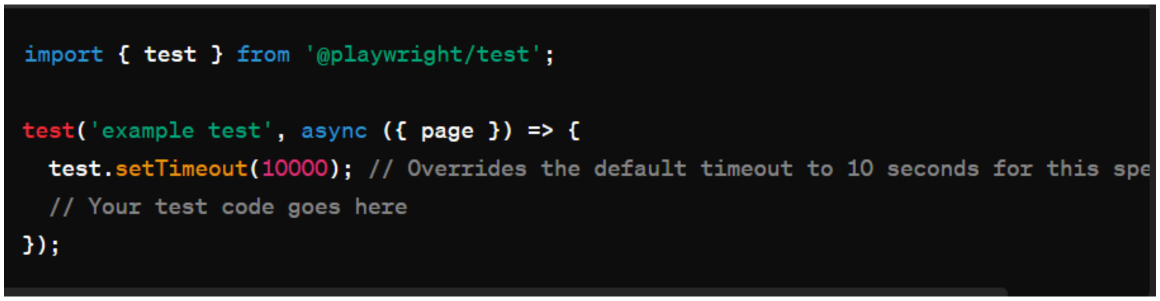
Q13. How to override test timeout for a specific test?

APPROACH:

Detail the method to adjust the timeout setting for a singular test within the Playwright Test framework.

ANSWER:

In the Playwright Test framework, overriding the test timeout for a specific test is straightforward. You can utilize the `test.setTimeout()` function within your test to set a custom timeout value. This allows you to specify how long Playwright should wait for the test to finish before considering it failed due to a timeout. Here's how you can apply it:



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

test('example test', async ({ page }) => {
  test.setTimeout(10000); // Overrides the default timeout to 10 seconds for this spe
  // Your test code goes here
});
```

This method provides granular control over the execution time allowed for individual tests, enabling adjustments for tests that might need more time to complete than the global default timeout setting.

Q14. When and how to use page.pause()?

APPROACH:

Explain the purpose and usage of `page.pause()` in Playwright.

ANSWER:

`page.pause()` is used to pause test execution and open up Playwright's Inspector, allowing developers to inspect the page, view DOM elements, and interact with the page manually.

This is particularly useful for debugging tests and understanding the page state at a specific point in the execution.

Q15. Difference between locator and selector in Playwright?

APPROACH:

Differentiate between locators and selectors within the context of Playwright tests.

ANSWER:

In Playwright, selectors are strings that describe a path to find an element on the page (e.g., CSS or XPath selectors). Locators, on the other hand, are objects that refer to an element or a set of elements, providing methods to interact with those elements. Locators are resolved lazily, meaning the element search is performed when an action is taken, not when the locator is created.

Q16. How do you handle multiple windows or tabs in Playwright?

APPROACH:

Outline strategies for effectively managing scenarios involving multiple browser windows or tabs during testing with Playwright, focusing on techniques for capturing and interacting with new windows or tabs as they are opened.

ANSWER

Event Listening: Utilize the `page.on('popup', async (popup) => {...})` listener for popups and new tabs triggered by user interactions. This allows you to capture and directly interact with newly opened windows or tabs in real-time.

waitForEvent Method: Implement `context.waitForEvent('page')` in conjunction with the action that triggers a new window or tab. This method is effective for synchronously waiting for and capturing new pages when they're initiated by specific actions, such as clicking a link.

Promise.all Pattern: Use the `Promise.all` method to synchronize the event waiting (`context.waitForEvent('page')`) and the action that triggers the new window or tab. This ensures that your test script waits for the new window to be fully ready before proceeding, facilitating immediate interaction with the newly opened window or tab.

Direct Page Creation: For scenarios that require opening a new window or tab without user interaction, `await context.newPage()` directly creates and opens a new page within the same browser context. This method provides a straightforward way to initiate new tabs or windows programmatically for testing purposes.

Managing Multiple Windows: Manage and switch between multiple `page` objects when your test involves interactions across various windows or tabs. This approach allows for isolated interactions within each window or tab, enabling complex test scenarios involving multiple steps or states across different pages.

Q17. Explain handling shadow DOMs in Playwright.

APPROACH:

Outline the methods Playwright employs to interact with elements within shadow DOMs, including the challenges encountered and the solutions for effective testing.

ANSWER:

Playwright addresses the challenge of interacting with shadow DOM elements by providing a powerful CSS selector engine capable of penetrating shadow roots. This allows testers to access elements within a shadow DOM using standard CSS selectors. The unique feature here is the use of the `shadow/` prefix, which explicitly indicates shadow DOM traversal, enabling direct and seamless interaction with shadow DOM elements.

Challenges:

1. Complex Selectors: Navigating through nested shadow DOMs can require complex selectors, making tests harder to read and maintain.

2. Dynamic Content: Shadow DOMs often encapsulate dynamic content, which can pose synchronization issues.

Solutions:

1. Simplified Selectors: Playwright's support for `>>` syntax simplifies the process of traversing through nested shadow DOMs, allowing for more readable selectors.

2. Auto-wait Features: Playwright's auto-wait capabilities ensure that actions only proceed once the elements are ready, mitigating issues with dynamic content within shadow DOMs.

This approach and these solutions make Playwright an effective tool for testing web components and applications that make extensive use of shadow DOMs, providing a straightforward way to interact with encapsulated elements while addressing common challenges.

Q18. Key flags in your Playwright config file and their impact?

APPROACH:

Highlight important configuration options in the Playwright configuration file and their effects on test behavior.

ANSWER:

Key flags in a Playwright config file might include `baseURL` for defining a base URL for all tests, `browsers` to specify which browsers to run tests in, `timeout` to set global timeout values for actions or navigations, and `headless` to toggle between headless and headed modes. These settings impact how tests are executed and interact with the web environment.

Q19. Managing sensitive data and preventing code repository exposure?

APPROACH:

Outline strategies for securely handling sensitive data like login credentials within Playwright tests.

ANSWER:

To manage sensitive data, use environment variables or secure vault services to inject credentials at runtime, avoiding hard-coding them in test scripts. Ensure that files containing sensitive information are excluded from source control through `.gitignore` or equivalent mechanisms in other version control systems.

Q20. Utilizing Playwright's network interception capabilities?

APPROACH:

Explain how to use Playwright's features for intercepting and modifying network requests.

ANSWER:

Playwright allows intercepting network requests using the `route` method on the browser context or page. You can match requests based on URL patterns and then stub responses, modify request parameters, or log request data. This is useful for testing how your application behaves under various network conditions or with different server responses.

Q21. Implementing and using event listeners in Playwright?

APPROACH:

Discuss the implementation and use cases of event listeners in a Playwright test suite.

ANSWER:

Event listeners in Playwright can be used to react to browser events, such as page navigation, popup opening, or network requests. You attach listeners to the page or browser context objects to handle specific events, enabling actions like logging, data collection, or conditional test flow based on event occurrence.

Q22. Handling asynchronous operations in Playwright?

APPROACH:

Describe strategies for managing asynchronous behavior in Playwright tests.

ANSWER:

Asynchronous operations in Playwright are handled using JavaScript's async/await syntax, allowing you to wait for actions, navigations, and other asynchronous events to complete before proceeding with the test. This ensures that your tests are stable and that interactions with the page occur only after the necessary elements and data are available.

Q23. Strategies for clearing input fields?

APPROACH:

Provide methods for efficiently clearing input fields during testing.

ANSWER:

To clear input fields in Playwright, you can use the `fill` method with an empty string, effectively clearing the field. Alternatively, you can select the input text using `click` with `{clickCount: 3}` to select all text and then press the Backspace or Delete key to clear the selected content.

Q24. What are actionability checks in Playwright?

APPROACH:

Explain the concept of actionability checks in the context of Playwright tests.

ANSWER:

Actionability checks in Playwright are preconditions verified before performing actions on elements, ensuring that elements are visible, enabled, and stable. This includes checks for visibility, reachability by clicks, and absence of movement, ensuring reliable interactions in tests.

Q25. Managing different timeouts in your framework?

APPROACH:

Discuss how to configure and manage various timeout settings in Playwright tests.

ANSWER:

Playwright allows configuring different timeout settings for actions, navigations, and the overall test timeout. These can be set globally in the Playwright config file or on a per-test or per-action basis, allowing flexibility in handling operations that might require more or less time to complete.

Q26. Difference between serial and parallel mode in testing?

APPROACH:

Compare serial and parallel execution modes in the context of automated testing.

ANSWER:

In serial execution, tests run one after another, ensuring that tests do not affect each other's state but taking more time to complete all tests. Parallel execution runs multiple tests at the same time, significantly reducing the total test time but requiring careful management of shared resources and state to avoid test interference.

Q27. Headed vs headless mode in CI environments?

APPROACH:

Contrast the use of headed and headless modes when running tests in continuous integration (CI) environments.

ANSWER:

In CI environments, tests are typically run in headless mode to reduce resource usage and speed up execution, as no graphical interface is needed. Headed mode might be used for debugging or specific tests requiring visual validation, but generally, headless mode is preferred for automated test runs in CI pipelines.

C) Framework Level Questions

Q28. Explain your project framework with a focus on integration and handling test case challenges.

APPROACH:

Describe the structure of the test automation framework, focusing on its integration with other tools and solutions for addressing common test case challenges.

ANSWER:

The framework is built on Playwright and integrates with CI/CD pipelines (like Jenkins or GitHub Actions) for automated test execution on code commits or scheduled runs. It leverages the Page Object Model (POM) for maintainability and reusability of code. Challenges like dynamic content handling and flakiness are addressed through Playwright's auto-wait features and custom retry mechanisms. Integration with bug tracking and reporting tools automates the defect lifecycle.

Q29. Why choose the Page Object Model, and its benefits?

APPROACH:

Discuss the reasons for selecting the Page Object Model and its advantages in test automation projects.

ANSWER:

The Page Object Model was chosen for its ability to abstract page details from tests, enhancing test maintainability and readability. Benefits include reduced code duplication, simplified maintenance through centralized page element management, easier updates to tests when UI changes, and the facilitation of reusability across different tests.

Q30. Can Playwright.config.ts have multiple configurations?

APPROACH:

Explore the possibility and implications of having multiple configurations within the Playwright.config.ts file.

ANSWER:

Yes, Playwright.config.ts can define multiple configurations using the `projects` feature. This allows for different test environments, browser setups, or custom settings for subsets of tests, improving flexibility and enabling more granular control over test execution conditions.

Q31. Managing environment-level data in your framework?

APPROACH:

Outline strategies for handling environment-specific data within the testing framework.

ANSWER:

Environment-level data is managed using a combination of configuration files and environment variables. Configuration files define default settings, which can be overridden by environment variables for flexibility across different testing stages (development, staging, production). This approach secures sensitive information and allows for easy changes without code modifications.

Q32. Utilizing projects in Playwright config for property overrides?

APPROACH:

Discuss how to use the `projects` feature in Playwright config for overriding properties.

ANSWER:

The `projects` feature in Playwright config is used to define specific settings for different test suites, such as browser types, viewport sizes, or custom test flags. This enables tailored test executions for various scenarios, like mobile testing or cross-browser checks, by overriding global configurations on a per-project basis.

Q33. Implementing global setup and teardown in Playwright?

APPROACH:

Describe the mechanism for setting up global preconditions and cleanup actions in Playwright tests.

ANSWER:

Global setup and teardown are implemented using Playwright's lifecycle hooks, allowing for execution of scripts before and after the entire test suite runs. This is useful for tasks like starting up test servers, initializing databases, and cleaning up resources after tests complete, ensuring a consistent environment for test execution.

Q34. Tagging test cases and different tags used?

APPROACH:

Explain the purpose of tagging test cases and the types of tags applied.

ANSWER:

Tagging test cases facilitates categorizing and selectively running subsets of tests based on criteria like functionality, priority, or execution speed. Common tags include `smoke`, `regression`, `sanity`, and `performance`, allowing for targeted test runs and efficient test management.

Q35. Customizing test reporting in Playwright?

APPROACH:

Outline methods for customizing test reports in Playwright.

ANSWER:

Custom test reporting in Playwright can be achieved by integrating with third-party reporting tools (like Allure or ReportPortal) or utilizing Playwright's built-in reporters. Custom reporters can be developed to format test results, capture additional metadata, and integrate with external systems for enhanced reporting capabilities.

Q36. Video recording in CI: when to enable or disable?

APPROACH:

Discuss considerations for enabling or disabling video recording of tests in continuous integration environments.

ANSWER:

Video recording in CI should be enabled for critical or flaky tests to aid in debugging, especially when issues are hard to reproduce locally. However, it should be disabled for routine test executions to save resources and time, as video capture can significantly increase test run durations.

Q37. Managing test case data in your project?

APPROACH:

Describe strategies for organizing and accessing test case data.

ANSWER:

Test case data is managed through external data files (JSON or CSV) or fixtures, allowing for easy maintenance and reusability across tests. Data-driven testing patterns are implemented to parameterize tests with different inputs, enhancing test coverage and reducing redundancy.

Q38. Use of fixtures in Playwright and their application?

APPROACH:

Explain the role of fixtures in Playwright tests and how they are applied.

ANSWER:

Fixtures in Playwright provide reusable setup and teardown logic for test environments, such as creating a new browser context or page, and logging in before tests. They ensure a consistent test environment, improve code reusability, and simplify test writing by abstracting common preconditions and cleanup actions.

D) Node & NPM Related Questions

Q39. What is Node.js and its importance in your project?

APPROACH:

Provide a definition of Node.js and explain its role within the context of the project.

ANSWER:

Node.js is an open-source, cross-platform, JavaScript runtime environment that executes JavaScript code outside a web browser. In our project, Node.js is crucial for backend services, serving APIs, handling server-side logic, and interacting with databases. It enables us to use JavaScript across the full stack, improving development efficiency and reducing context switching between the frontend and backend.

Q40. What is npm and how do you utilize its capabilities in your project?

APPROACH:

Describe npm and its application within the project environment.

ANSWER:

npm (Node Package Manager) is the default package manager for Node.js, facilitating the management of project dependencies. In our project, we use npm to install, update, and manage library dependencies efficiently. It also supports scripting, allowing us to automate repetitive tasks like builds, tests, and deployments, enhancing productivity and consistency across development, testing, and production environments.

Q41. How do you install and update packages using npm?

APPROACH:

Explain the process of installing and updating npm packages.

ANSWER:

To install a package using npm, we use the command `npm install <package-name>`, which adds the package to our project dependencies. To install a specific package version, we append `@<version>` to the package name. For updating packages, `npm update <package-name>` updates the specified package to the latest version based on the version rules in `package.json`. Running `npm install` without arguments installs all project dependencies based on `package.json` and `package-lock.json`.

Q42. Difference between package.json and package-lock.json?

APPROACH:

Highlight the distinctions between `package.json` and `package-lock.json` files.

ANSWER:

`package.json` is a manifest file that records important metadata about a project and lists its dependencies. It allows specifying versions using semantic versioning rules. `package-lock.json` is automatically generated and specifies the exact versions of all packages and their dependencies installed in `node_modules`. It ensures consistency of installed packages across different environments by locking down versions.

Q43. Using scripts in your Playwright project?

APPROACH:

Discuss how npm scripts are used within a Playwright project.

ANSWER:

In our Playwright project, npm scripts are defined in `package.json` to automate common tasks like starting the test runner, setting up environments, and running linting tools. For example, we might have a `"test"` script to execute Playwright tests (`playwright test`), a `"lint"` script for ESLint checks, and a `"start"` script for running a development server, streamlining development workflows and CI/CD pipelines.

Q44. Difference between npm and npx execution?

APPROACH:

Compare the use cases and execution methods of npm and npx.

ANSWER:

npm is used for managing dependencies and running scripts defined in `package.json`. npx, which comes with npm, is used to execute Node packages without installing them globally. It's particularly useful for running packages with binaries or when testing different package versions without affecting the global or project-specific installations. For instance, `npx playwright test` runs Playwright tests without needing Playwright installed globally.

Q45. Awareness of ESLint and its importance at the project level?

APPROACH:

Explain what ESLint is and its significance in maintaining project quality.

ANSWER:

ESLint is a static code analysis tool for identifying problematic patterns in JavaScript code. In our project, it's crucial for ensuring code quality, consistency, and adherence to coding standards. It helps prevent bugs and improves maintainability by enforcing rules on code style, syntax errors, and best practices. We integrate ESLint in our development process and CI/CD pipelines to automatically check code and ensure team members follow defined coding standards, contributing to the overall health and readability of the codebase.

E) Code Snippets and Practical Application

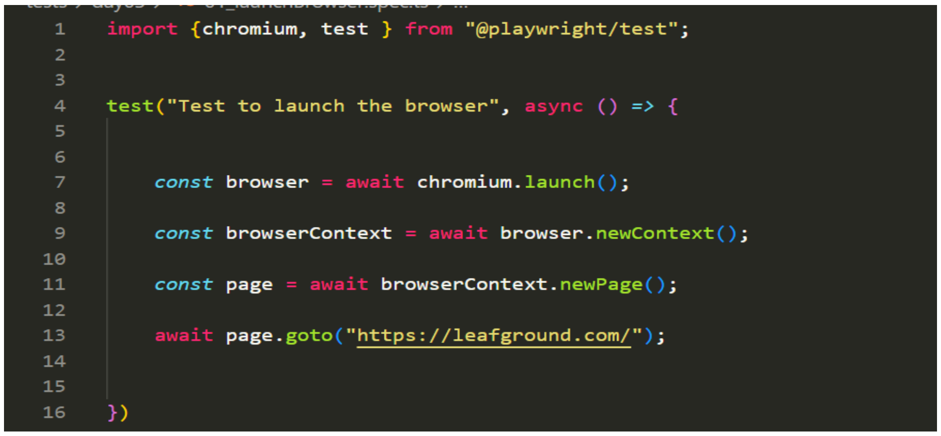
Q46. How do you invoke a new page in Playwright?

APPROACH:

Outline the steps required to open a new page within the context of Playwright's testing framework.

ANSWER:

Invoking a new page in Playwright involves creating a browser instance, initializing a new context, and then opening a page within that context. This process isolates the page from others, allowing for independent test scenarios.



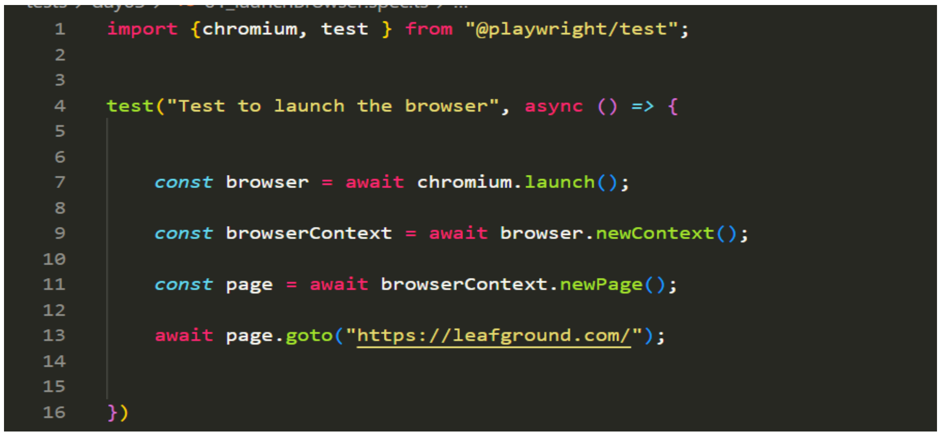
```
import {chromium, test } from "@playwright/test";


test("Test to launch the browser", async () => {


    const browser = await chromium.launch();

    const browserContext = await browser.newContext();

    const page = await browserContext.newPage();

    await page.goto("https://leafground.com/");


})
```

Q47. Code to create a new browser, context, and page in Playwright.

APPROACH:

Provide a code snippet that demonstrates the creation of a new browser instance, a browser context, and then opening a new page.

ANSWER



```
import {chromium, test } from "@playwright/test";


test("Test to launch the browser", async () => {


    const browser = await chromium.launch();

    const browserContext = await browser.newContext();

    const page = await browserContext.newPage();

    await page.goto("https://leafground.com/");


})
```

Q48. Handling dropdown interactions and assertions in Playwright.

APPROACH:

Explain how to interact with dropdown menus in Playwright and perform assertions on selected values.

ANSWER:

To handle dropdown interactions, you typically use the `selectOption` method on the element handle of the dropdown. Assertions can then be made on the selected value using the `expect` API.

```
// Use page.locator to target the dropdown. Replace '#dropdown' with the
// selector for your dropdown.
const dropdown = page.locator('#dropdown');

// Select an option from the dropdown. Replace 'optionValue' with the value
// attribute of the option you want to select.
await dropdown.selectOption({ value: 'optionValue' });

// Assert the selected option's value. Replace 'optionValue' with the expected
// value.
await expect(dropdown).toHaveValue('optionValue');

// Alternatively, assert based on the visible text of the selected option.
// Replace 'Visible Option Text' with the expected visible text.
await expect(dropdown).toHaveText('Visible Option Text');
```

Q49. Code for handling multiple windows in Playwright.

APPROACH:

Provide a code snippet for managing scenarios where interactions open new browser windows or tabs.

ANSWER:

```
// Clicking a link to open a new tab, while waiting for the new page event
const [newPage] = await Promise.all([
  context.waitForEvent('page'), // Wait for the new page (tab/window)
  page.click('#linkThatOpensNewTab') // The action to open new tab
]);

// With the new page object, you can interact with the new tab/window
await newPage.waitForLoadState();
// Example: Print the title of the new page
console.log(await newPage.title());
```

Q50. Function to check visibility of an element.

APPROACH:

Describe how to check if an element is visible on the page.

ANSWER:

```
const isVisible = await page.isVisible('selector');
expect(isVisible).toBe(true); // Assertion to check if element is visible
```

Q51. Making an API call, saving state, and reusing it in a new context.

APPROACH:

Outline the steps to make an API call, save the browser context state, and reuse this state in a new context.

ANSWER:

```
// Saves the state to a file
await context.storageState({ path: 'state.json' });

// Later, to reuse the state
const browser = await chromium.launch();
// Reuses the saved state
const context = await browser.newContext({ storageState: 'state.json' });
const page = await context.newPage();
```