Playwright Interview Questions:

Playwright Locators

Q: What are locators in Playwright and how do they differ from traditional Selenium selectors?

APPROACH:

Explain the concept of locators in Playwright, emphasizing their role in identifying elements on a webpage. Compare them to Selenium selectors to highlight differences in functionality and efficiency.

ANSWER:

Locators in Playwright are abstractions used to interact with page elements. Unlike Selenium, where selectors directly interact with elements, Playwright's locators are resilient to changes in the DOM, allowing for more robust tests. They enable actions to wait for elements to be actionable before executing, reducing flakiness in automated tests.

Q: How do you handle dynamic content or elements that load asynchronously with Playwright locators?

APPROACH:

Describe methods to deal with elements that aren't immediately available on page load, such as those loaded through JavaScript actions or dynamic content.

ANSWER:

Playwright automatically waits for elements to be present and stable before interacting with them. For dynamic content, using locators with waitForSelector or combining actions with waitFor functions ensures interactions occur only after elements are fully loaded and stable.

Assertions

Q: What is the difference between auto-retrying and non-retrying assertions in Playwright tests?

APPROACH:

Define both types of assertions and explain how they operate differently within the context of a Playwright test.

ANSWER:

Auto-retrying assertions automatically wait for a condition to be true before proceeding, making them ideal for dealing with asynchronous operations. Non-retrying assertions, on the other hand, check the condition immediately at the time of execution, which can lead to failures if the page has not reached the expected state.

Q: How can you implement a non-retrying assertion to check if an element is visible on the page?

APPROACH:

Describe the steps to perform a non-retrying assertion in Playwright, using visibility checks as an example.

ANSWER:

To implement a non-retrying assertion, you can directly use the `expect` statement with `.toBeVisible()` matcher immediately after the action that should make the element visible, without waiting for the element to appear. For example, `expect(page.locator('button.submit')).toBeVisible();` ensures the button is visible right after a form submission trigger.

Q: Provide an example of a complex assertion involving auto-retrying mechanism that waits for a specific element's text to change to 'Success' after an asynchronous operation.

APPROACH:

Explain the process of creating an auto-retrying assertion that involves waiting for text changes in an element, highlighting the use of asynchronous operations.

ANSWER:

For auto-retrying assertions that wait for text changes, use the `expect` function with the `.toHaveText()` matcher, which automatically retries until the condition is met or a timeout occurs. For instance, after initiating a background data processing task, use `await expect(page.locator('.status')).toHaveText('Success');` to wait for the `.status` element's text to change to "Success," indicating the operation's completion.

Playwright Config File

Q: What is the purpose of the Playwright config file?

APPROACH:

Outline the role and capabilities of the Playwright configuration file in a testing environment.

ANSWER:

The Playwright config file centralizes the setup and configuration for Playwright tests, including default browser settings, test directories, timeouts, and global hooks. It streamlines test execution and environment setup, making tests more maintainable and configurable.

Q: How do you specify a custom test timeout and test retries in the Playwright config file?

APPROACH:

Describe the process of customizing test timeouts and setting the number of retries for failed tests within the config file.

ANSWER:

In the Playwright config file, you can set a custom test timeout using the `timeout` property and configure retries for failed tests with the `retries` property. This is particularly useful for adapting to varying network conditions or debugging. For example, to set a global timeout of 30 seconds and allow 2 retries for failed tests, your configuration would include:

```
// playwright.config.js
export default defineConfig({
 timeout: 30000, // 30 seconds
 retries: 2,
});
```

Q: How do you configure Playwright to use different browsers for tests and integrate with a CI/CD pipeline using the config file?

APPROACH:

Discuss the steps to set up Playwright for cross-browser testing and its integration with continuous integration/continuous deployment (CI/CD) systems through modifications in the config file.

ANSWER:

To configure Playwright for cross-browser testing, specify the browsers in the `projects` section of the Playwright config file. For CI/CD integration, ensure the config file includes settings for headless execution and possibly customizes the output for test results. An example configuration might look like:

```
// playwright.config.js
export default defineConfig({
 projects: [
  {
   name: 'Chromium',
   use: { browserName: 'chromium', headless: true },
  },
  {
   name: 'Firefox',
```

```
    use: { browserName: 'firefox', headless: true },
  },
  {
    name: 'WebKit',
    use: { browserName: 'webkit', headless: true },
  },
 ],
 // Additional CI/CD specific configurations
 use: {
  // Enable video only on CI
  video: process.env.CI ? 'on' : 'off',
 },
});
```

This setup allows for testing across different browsers and provides configurations that are optimal for both local development and CI/CD environments.