1. What is Playwright?

Playwright is an open-source automation library developed by Microsoft for testing web applications across different browsers. It allows developers to write scripts in various programming languages to automate browser actions, perform end-to-end testing, and interact with web pages.

2. What is the difference between Selenium and Playwright?

Browser Support: Playwright supports multiple browsers (Chromium, Firefox, WebKit) with a single API, while Selenium requires different drivers for different browsers.

Speed: Playwright is generally faster due to its ability to run tests in parallel and its efficient handling of network conditions.

Auto-Waiting: Playwright automatically waits for elements to be ready before performing actions, reducing the need for manual waits.

Headless Mode: Playwright has built-in support for headless mode, while Selenium requires additional configuration.

3. What are the advantages of Playwright?

Cross-Browser Testing: Supports multiple browsers with a single API.

Auto-Waiting: Automatically waits for elements to be ready.

Network Interception: Allows interception and modification of network requests.

Multiple Contexts: Supports multiple browser contexts for parallel testing.

Mobile Emulation: Supports mobile device emulation.

4. Name some disadvantages of Playwright.

Newer Tool: Being relatively new, it may have fewer community resources compared to Selenium.

Limited Documentation: While improving, some areas of documentation may be less comprehensive.

Learning Curve: Developers familiar with Selenium may need time to adapt to Playwright's API.

5. What are the different testing types that Playwright supports?

End-to-End Testing: Testing the entire application flow.

API Testing: Testing backend services.

Visual Testing: Comparing visual output against expected results.

Performance Testing: Measuring application performance under load.

6. What are the programming languages that Playwright supports?

JavaScript

TypeScript

Python

Java

C#

7. Briefly describe the commands that are used for Playwright installation and execution of tests.

To install Playwright**, use the following command:**

bash

npm install playwright

To execute tests using the Playwright Test runner:

bash

npx playwright test

8. What is a Configuration File in Playwright? Explain.

A configuration file in Playwright (usually playwright.config.ts or playwright.config.js) is used to define global settings for your tests, such as timeouts, reporter settings, and test directory paths.

typescript

```
// playwright.config.ts
import { defineConfig } from '@playwright/test';

export default defineConfig({
  timeout: 30000,
  reporter: 'html',
  use: {
    headless: true,
```

```
  },
});
```

9. What is the @playwright/test package in Playwright?

The @playwright/test package is the official test runner for Playwright. It provides a framework for writing and running tests, including features like fixtures, parallel execution, and built-in assertions.

10. What is the Page class in Playwright?

The Page class represents a single tab or page in the browser. It provides methods to interact with the page, such as navigating, clicking elements, and evaluating JavaScript.

11. How to navigate to specific URLs in Playwright? Explain with sample tests.

You can navigate to a specific URL using the goto method of the Page class.

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

test('navigate to example.com', async ({ page }) => {
  await page.goto('https://example.com');
  const title = await page.title();
  expect(title).toBe('Example Domain');
});
```

12. What are the different types of reporters that Playwright supports?

Playwright supports several reporters, including:

Dot Reporter: Outputs a dot for each test.

List Reporter: Outputs a list of tests.

HTML Reporter: Generates an HTML report.

JSON Reporter: Outputs results in JSON format.

JUnit Reporter: Outputs results in JUnit format.

13. What are the locators in Playwright? List any five.

Locators in Playwright are used to find elements on the page. Here are five types of locators:

CSS Selector: page.locator('div.classname')

Text Selector: page.locator('text=Submit')

XPath Selector: page.locator('//button[text()="Submit"]')

Role Selector: page.locator('role=button[name="Submit"]')

ID Selector: page.locator('#elementId')

14. What are the different types of text selectors available in Playwright?

Exact Match: page.locator('text=Submit')

Partial Match: page.locator('text=Submit') (matches any text containing "Submit")

Regular Expression: page.locator(/Submit/) (matches text that matches the regex)

Case-Insensitive Match: page.locator('text=submit', { exact: false })

15. How to use assertions in Playwright? List any 5.

Assertions in Playwright are used to verify conditions in tests. Here are five examples:

toBe(): expect(value).toBe(expectedValue);

toEqual(): expect(value).toEqual(expectedValue);

toContain(): expect(array).toContain(item);

toHaveText(): expect(locator).toHaveText('expected text');

toBeVisible(): expect(locator).toBeVisible();

16. What are soft assertions in Playwright?

Soft assertions allow tests to continue running even if an assertion fails. Playwright does not have built-in soft assertions, but you can implement them using try-catch blocks to log failures without stopping the test.

17. How to negate the Matchers/Assertions in Playwright? Or how can I verify not conditions in Playwright?

You can negate assertions using the not keyword.

typescript

```
expect(locator).not.toBeVisible();
```

18. Does Playwright support XPath? If so, how can I use them in the test?

Yes, Playwright supports XPath. You can use the locator method with an XPath expression.

typescript

```
const element = await page.locator('//button[text()="Submit"]');
await element.click();
```

19. What are command line options in Playwright? Explain 5 helpful options.

Some helpful command line options in Playwright include:

--headed: Run tests in headed mode (with a visible browser).

--headless: Run tests in headless mode (without a visible browser).

--trace: Enable tracing for debugging.

--reporter: Specify the reporter to use (e.g., --reporter=html).

--config: Specify a custom configuration file (e.g., --config=custom-config.ts).

20. Some of the Important command line options

--list: List all tests.

--grep: Run tests that match a specific pattern.

--timeout: Set a global timeout for tests.

--workers: Set the number of parallel workers.

--project: Specify a project to run tests for.

21. What is headed and headless mode in Playwright?

Headed Mode: The browser runs with a visible UI, allowing you to see the actions being performed.

Headless Mode: The browser runs without a UI, which is faster and often used for automated testing in CI/CD pipelines.

22. Does Playwright support HTML reporters? How to Generate HTML reports in Playwright?

Yes, Playwright supports HTML reporters. You can generate HTML reports by specifying the reporter in the configuration file or using the command line.

typescript

```
// playwright.config.ts
export default defineConfig({
  reporter: 'html',
});
```

23. What are timeouts in Playwright? What are the different types of Timeouts available in Playwright?

Timeouts in Playwright specify how long to wait for certain actions to complete. Types of timeouts include:

Global Timeout: Set in the configuration file for all tests.

Test Timeout: Set for individual tests using the test() function.

Action Timeout: Set for specific actions like goto(), click(), etc.

24. How to navigate forward and backward in Playwright?

You can navigate forward and backward using the goBack() and goForward() methods.

typescript

```
await page.goBack(); // Navigate back
await page.goForward(); // Navigate forward
```

25. How to perform actions using Playwright?

You can perform actions like clicking, typing, and hovering using the Page class methods.

typescript

```
await page.click('button#submit');

await page.type('input[name="username"]', 'myUsername');

await page.hover('div.menu');
```

26. Does Playwright support the Safari browser? If so, can we execute the test on Safari?

Yes, Playwright supports Safari through WebKit. You can execute tests on Safari by using the WebKit browser.

typescript

```
const { webkit } = require('playwright');

const browser = await webkit.launch();
```

27. How to wait for a specific element in Playwright?

You can wait for an element using the waitForSelector() method.

typescript

```
await page.waitForSelector('div#result');
```

28. What is browser context?

A browser context is an isolated environment within a browser instance. Each context can have its own cookies, local storage, and session storage, allowing for parallel testing without interference.

29. How to open multiple windows in Playwright?

You can open multiple windows by creating new pages in the same context.

typescript

```
const page1 = await context.newPage();

const page2 = await context.newPage();
```

30. How to handle iFrame in Playwright?

You can handle iFrames by first locating the frame and then interacting with it.

typescript

```
const frame = await page.frame({ name: 'frameName' });
await frame.click('button#submit');
```

31. Explain some of the click and double-click actions with its options.

Click: Clicks on an element.

typescript

```
await page.click('button#submit', { delay: 100 }); // Adds a delay
```

Double Click: Double-clicks on an element.

typescript

```
await page.dblclick('button#edit');
```

32. How to perform a right-click on Playwright?

You can perform a right-click using the click() method with the button option set to right.

typescript

```
await page.click('button#context-menu', { button: 'right' });
```

33. How to evaluate JavaScript in Playwright?

You can evaluate JavaScript using the evaluate() method.

typescript

```
const result = await page.evaluate(() => {
  return document.title;
});
```

console.log(result);

34. What are Playwright fixtures?

Fixtures in Playwright are reusable components that set up the environment for tests. They can be used to create browser contexts, pages, or any other setup needed for tests.

typescript

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

test.beforeEach(async ({ page }) => {
  await page.goto('https://example.com');
});
```

35. What is CodeGen in Playwright?

CodeGen is a tool that generates Playwright scripts based on user interactions with the browser. It can be used to quickly create test scripts by recording actions.

bash

```
npx playwright codegen https://example.com
```

36. How to parameterize tests in Playwright?

You can parameterize tests using the test.each method.

typescript

```
test.each([
  ['input1', 'expected1'],
  ['input2', 'expected2'],
])('test with %s', async (input, expected) => {
  await page.fill('input', input);
```

```
  const result = await page.textContent('result');
  expect(result).toBe(expected);
});
```

37. Write a code to upload a file.

```
typescript
await page.setInputFiles('input[type="file"]', 'path/to/file.txt');
```

38. Write a code to download a file.

```
typescript
const [download] = await Promise.all([
  page.click('a#download'),
  page.waitForEvent('download'),
]);
const path = await download.path();
console.log(`Downloaded file path: ${path}`);
```

39. How to perform drag and drop in Playwright?

You can perform drag and drop using the dragAndDrop() method.

```
typescript
await page.dragAndDrop('div#source', 'div#target');
```

40. How to handle browser popups or dialogs?

You can handle dialogs using the on('dialog') event.

```
typescript
page.on('dialog', async dialog => {
  console.log(dialog.message());
```

```
  await dialog.accept(); // or dialog.dismiss();
});
```

41. What is the testInfo Object?

The testInfo object provides information about the currently running test, including its title, status, and duration. It can be used for logging and reporting.

typescript

```
test('example test', async ({ page }, testInfo) => {
  console.log(testInfo.title);
});
```

42. What is the testError Object?

The testError object contains information about any errors that occur during the test execution. It can be used for debugging and logging.

typescript

```
test('example test', async ({ page }, testInfo) => {
  try {
    await page.click('button#nonexistent');
  } catch (error) {
    console.error(testInfo.error); // Log the error
  }
});
```

43. What is global setup and teardown? Explain.

Global setup and teardown are functions that run once before and after all tests in a suite. They are useful for initializing resources or cleaning up after tests.

typescript

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

test.beforeAll(async () => {
  // Global setup code
});

test.afterAll(async () => {
  // Global teardown code
});
```

44. How to capture Network logs in Playwright?

You can capture network logs by listening to the request and response events.

```
typescript
page.on('request', request => {
  console.log('Request:', request.url());
});

page.on('response', response => {
  console.log('Response:', response.url(), response.status());
});
```

45. How to capture screenshots in Playwright?

You can capture screenshots using the screenshot() method.

```
typescript
await page.screenshot({ path: 'screenshot.png' });
```

46. Does Playwright support API testing? If so, how can we perform API testing?

Yes, Playwright supports API testing. You can use the request object to make API calls.

typescript

```
const response = await page.request.get('https://api.example.com/data');

const data = await response.json();

console.log(data);
```

47. What is Visual Testing? Why do we need it?

Visual testing involves comparing the visual appearance of a web application against a baseline image to detect visual changes. It is important for ensuring UI consistency and catching unintended changes.

48. Write a simple code to Test Visually.

typescript

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


test('visual test', async ({ page }) => {
  await page.goto('https://example.com');
  expect(await page.screenshot()).toMatchSnapshot('example-screenshot.png');
});
```

49. How to configure multiple reporters in Playwright?

You can configure multiple reporters in the configuration file.

typescript

```
export default defineConfig({
  reporter: [['html'], ['list']],
});
```

50. What is the serial mode in Playwright?

Serial mode runs tests one after another, rather than in parallel. This can be useful for tests that depend on shared state.

typescript

```
// playwright.config.ts
export default defineConfig({
  workers: 1, // Run tests in serial mode
});
```

51. How to perform parallel execution in Playwright?

You can perform parallel execution by default, as Playwright runs tests in parallel across multiple workers.

typescript

```
// playwright.config.ts
export default defineConfig({
  workers: 4, // Run tests in parallel with 4 workers
});
```

52. How to perform mobile device emulation in Playwright?

You can emulate mobile devices using the devices module.

typescript

```
import { devices, chromium } from 'playwright';

const iPhone = devices['iPhone 12'];
const browser = await chromium.launch();
const context = await browser.newContext({ ...iPhone });
```

```
const page = await context.newPage();

await page.goto('https://example.com');
```

53. Mention some of the helpful ways to debug Playwright tests.

Use debug(): Insert debugger; in your code to pause execution.

Run in headed mode: Use headless: false to see the browser actions.

Use trace: Enable tracing to capture detailed execution logs.

Screenshots: Capture screenshots on failure to see the state of the application.

Console logs: Use console.log() to output variable values and states.

54. What is actionability in Playwright? Explain in detail.

Actionability refers to the state of an element being ready for interaction. Playwright automatically waits for elements to be actionable (visible, enabled, etc.) before performing actions like clicks or typing. This reduces the need for manual waits and improves test reliability.

55. Mention some of the advantages of Playwright compared to Cypress.

Cross-Browser Support: Playwright supports multiple browsers (Chromium, Firefox, WebKit) with a single API, while Cypress primarily supports Chromium-based browsers.

Auto-Waiting: Playwright automatically waits for elements to be actionable, reducing flakiness.

Multiple Contexts: Playwright allows for multiple browser contexts, enabling parallel testing without interference.

Mobile Emulation: Playwright has built-in support for mobile device emulation.

Network Interception: Playwright allows for easy interception and modification of network requests.

This comprehensive guide covers a wide range of topics related to Playwright, providing a solid foundation for understanding and using the framework effectively. If you have any further questions or need more details on specific topics, feel free to ask!