✅ Playwright JavaScript – Top 50 FAQ

---

1. What is Playwright?
Playwright is an open-source automation framework by **Microsoft** used for end-to-end testing of web applications. It supports multiple browsers and languages including JavaScript.

---

2. Why Playwright over Selenium?
Playwright offers faster execution, auto-waiting, built-in parallel execution, network interception, and native support for modern browsers.

---

3. Which browsers does Playwright support?
Chromium, Firefox, and WebKit.

---

4. How do you install Playwright in JavaScript?

```
npm init playwright@latest
```

or

```
npm install -D @playwright/test

npx playwright install
```

---

5. What is auto-waiting in Playwright?
Playwright automatically waits for elements to be ready before performing actions.

---

6. How do you launch a browser in Playwright JS?

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


(async () => {

  const browser = await chromium.launch();
})();
```

---

7. What are sync and async APIs?
Playwright JavaScript uses asynchronous APIs with async/await for non-blocking execution.

---

8. What is a Browser Context?
An isolated environment inside a browser for independent sessions.

---

9. What is a Page object?
A Page represents a single browser tab.

---

10. How do you locate elements in Playwright?
Using CSS selectors, XPath, text selectors, and Playwright built-in locators.

---

11. What is Locator in Playwright?
A Locator is the recommended way to find and interact with elements reliably.

---

12. How do you handle waits?

```
await page.waitForSelector('#element');
```

Playwright also supports auto-waiting.

---

13. How to take screenshots?

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

---

14. How do you handle frames?

```
const frame = page.frameLocator('#frameId');
```

---

15. How do you handle alerts?

```
page.on('dialog', dialog => dialog.accept());
```

---

16. What is headless mode?
Browser runs without UI for faster execution.

---

17. How to run tests in headful mode?

```
await chromium.launch({ headless: false });
```

---

18. How do you perform assertions?
Using **Playwright Test assertions:

```
await expect(page).toHaveTitle(/Home/);
```

---

19. How do you integrate Playwright with a test runner?
Using the built-in Playwright Test framework.

---

20. What are fixtures in Playwright?
Reusable setup and teardown logic shared across tests.

---

21. How to handle dropdowns?

```
await page.selectOption('#dropdown', 'value');
```

---

22. How to upload files?

```
await page.setInputFiles('#upload', 'file.txt');
```

---

23. How to download files?

```
const download = await page.waitForEvent('download');
```

---

24. How to handle multiple tabs?

Using browser context and:

```
context.waitForEvent('page');
```

---

25. How to perform mouse actions?

```
await page.mouse.move(100, 100);
```

---

26. How to execute JavaScript?

```
await page.evaluate(() => document.title);
```

---

27. What is Page Object Model (POM)?

A design pattern that separates UI elements and test logic.

---

28. How do you structure a Playwright framework?

Folders for tests, pages, utilities, fixtures, and config files.

---

29. How to run tests in parallel?

Playwright Test runs tests in parallel by default.

---

30. How do you generate reports?

Using Playwright's built-in HTML reporter.

---

31. How do you handle authentication?

```
await context.storageState({ path: 'auth.json' });
```

---

32. What is tracing in Playwright?

Tracing records test execution for debugging.

---

33. How to enable tracing?

```
await context.tracing.start();
```

---

34. How to handle dynamic elements?
Using stable locators and auto-waits.

---

35. How do you scroll a page?

```
await page.mouse.wheel(0, 500);
```

---

36. How to handle cookies?

```
await context.cookies();
```

---

37. How to intercept network requests?

```
await page.route('**/api', route => route.continue());
```

---

38. How do you mock API responses?
Using page.route() to intercept and fulfill requests.

---

39. How to handle timeouts?

```
page.setDefaultTimeout(30000);
```

---

40. How to debug tests?
Using Playwright Inspector or debug mode.

---

41. What is Playwright Inspector?
A GUI debugging tool for Playwright scripts.

---

42. How do you record tests?

```
npx playwright codegen
```

---

43. How to handle drag and drop?

```
await source.dragTo(target);
```

---

44. How do you handle keyboard actions?

```
await page.keyboard.press('Enter');
```

---

45. How do you validate text?

```
await expect(locator).toHaveText('Success');
```

---

46. How do you retry failed tests?

```
retries: 2
```

(Config setting in Playwright)

---

47. How do you manage test data?
Using fixtures or external JSON/CSV files.

---

48. How do you integrate CI/CD?
Using Jenkins or GitHub Actions pipelines.

---

49. How do you improve test stability?
Using stable locators, retries, and proper waits.

---

50. What are best practices in Playwright automation?
Use POM, reusable fixtures, meaningful assertions, and clean test design.

✅ Locators in Playwright (JavaScript) — Complete Guide

In Playwright JS, locators are used to find and interact with elements on a webpage.

There are **3 main locator strategies**:

👉 XPath
👉 CSS Selectors
👉 Playwright Built-in Locators ⭐ (Best Practice)

✅ 1. XPath in Playwright (JavaScript)

👉 What is XPath?

XPath is a query language used to locate elements based on **HTML structure and attributes**.

In Playwright JS, XPath is used like this:

```
await page.locator("xpath=//your_xpath_here");
```

Or simply:

```
await page.locator("//your_xpath_here");
```

✅ Example HTML

```
<input type="text" id="username" name="user">

<button class="login-btn">Login</button>
```

✅ XPath Examples (with explanation)

1. Using ID

```
await page.locator("//input[@id='username']").fill("admin");
```

👉 Finds <input> element where id = username

---

2. Using Text

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

👉 Clicks button whose visible text is **Login**

---

3. Using Contains

```
await page.locator("//button[contains(@class,'login')]").click();
```

👉 Matches partial class name containing **login**

---

4. Relative XPath

```
await page.locator("//div[@class='form']//input").fill("test");
```

👉 Finds input inside a parent div with class **form**

✅ When to use XPath?

✔ Complex DOM structure
✔ Parent-child relationships
✔ No unique CSS selector available

❌ Avoid long XPath (hard to maintain)

✅ 2. CSS Selectors in Playwright (JavaScript)

👉 What is CSS Selector?

CSS selectors locate elements using:

- id
- class
- attributes
- element hierarchy

Playwright uses CSS by default.

✅ CSS Examples (with explanation)

1. By ID

await page.locator("#username").fill("admin");

Same as:

await page.locator("input#username");

👉 Selects element with id **username**

---

2. By Class

await page.locator(".login-btn").click();

👉 Selects element with class **login-btn**

---

3. By Attribute

await page.locator("input[name='user']").fill("test");

👉 Selects input with attribute name=user

---

4. Parent → Child

await page.locator("div.form input").fill("hello");

👉 Selects input inside div with class **form**

✅ **Advantages of CSS**

✔ Faster than XPath
✔ Cleaner syntax
✔ Easy to maintain
✔ Preferred in most cases

✅ **3. Playwright Built-in Locators ⭐ (Best Practice)**

Playwright provides smart, user-focused locators based on:

👉 Text
👉 Role
👉 Label

👉 Placeholder
👉 Test ID

These are **recommended in real projects and interviews**.

✅ 1. getByText()

```
await page.getByText("Login").click();
```

👉 Clicks element with visible text **Login**

✅ 2. getByRole() ⭐ Most Stable

```
await page.getByRole("button", { name: "Login" }).click();
```

👉 Finds button using accessibility role

Best for long-term stability.

✅ 3. getByLabel()

```
await page.getByLabel("Username").fill("admin");
```

👉 Uses associated label text

✅ 4. getByPlaceholder()

```
await page.getByPlaceholder("Enter username").fill("admin");
```

✅ 5. getByTestId() ⭐ Best for frameworks

HTML:

```
<button data-testid="login-btn">Login</button>
```

Code:

```
await page.getByTestId("login-btn").click();
```

👉 Best for automation frameworks

✅ Real Interview Question

👉 Which locator is best?

Priority order:

1. Built-in locators ☆
2. CSS selectors
3. XPath (only if necessary)

✅ Full JavaScript Example (All Locator Types)

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

(async () => {
  const browser = await chromium.launch({ headless: false });
  const page = await browser.newPage();

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

  // XPath
  await page.locator("//input[@id='username']").fill("admin");

  // CSS
  await page.locator("#password").fill("1234");

  // Built-in locator
  await page.getByRole("button", { name: "Login" }).click();

  await browser.close();
})();
```

✅ Handling Multiple Tabs in Playwright (JavaScript)

🗆 Create New Tab Manually

```
const context = await browser.newContext();


const page1 = await context.newPage();
await page1.goto("https://google.com");


const page2 = await context.newPage();
await page2.goto("https://github.com");
```

👉 context.newPage() opens a new tab

🗆 Open New Tab After Click

```
const [page2] = await Promise.all([
  context.waitForEvent("page"),
  page1.click("text=Open New Tab"),
]);


await page2.waitForLoadState();
console.log(await page2.title());
```

🗆 Switch Between Tabs

```
await page1.bringToFront();
await page2.bringToFront();
```

🗆 Get All Tabs

```
context.pages().forEach(p => console.log(p.url()));
```

▯ Close Specific Tab

```
await page2.close();
```

✅ Interview Answer (Multiple Tabs)

Tabs are handled using browser context. I use context.waitForEvent('page') to capture new tabs. Each tab is a Page object, and I switch using bringToFront().

✅ Advanced JavaScript Playwright Scenarios

---

1. Handle Popup Window

```
const [popup] = await Promise.all([
  context.waitForEvent("page"),
  page.click("#openPopup"),
]);


await popup.waitForLoadState();
console.log(await popup.title());
```

---

2. Multiple Browser Contexts

```
const context1 = await browser.newContext();
const context2 = await browser.newContext();


const page1 = await context1.newPage();
const page2 = await context2.newPage();
```

---

3. Refresh Page

```
await page.reload();
```

---

4. Browser Navigation

```
await page.goBack();

await page.goForward();
```

---

5. Wait for Page Load

```
await page.waitForLoadState("load");
```

Options: "domcontentloaded", "networkidle"

---

6. Check Element Visibility

```
const visible = await page.locator("#login").isVisible();

console.log(visible);
```

---

7. Hover Element

```
await page.locator("#menu").hover();
```

---

8. Double Click

```
await page.locator("#btn").dblclick();
```

---

9. Right Click

```
await page.locator("#btn").click({ button: "right" });
```

---

10. Scroll to Element

```
await page.locator("#footer").scrollIntoViewIfNeeded();
```

---

11. Get Text

```
const text = await page.locator("#title").textContent();

console.log(text);
```

---

12. File Download

```
const [download] = await Promise.all([
  page.waitForEvent("download"),
  page.click("#downloadBtn"),
]);


await download.saveAs("file.pdf");
```

---

13. File Upload

```
await page.setInputFiles("#upload", "testfile.txt");
```

---

14. Full Page Screenshot

```
await page.screenshot({ path: "full.png", fullPage: true });
```

---

15. Capture Console Logs

```
page.on("console", msg => console.log(msg.text()));
```

✅ Bonus Interview Question

👉 How do you handle synchronization issues?

Playwright provides built-in auto-waiting. I rely on locators and waitForLoadState() instead of hard-coded delays to keep tests stable.

Here’s a **clean Playwright JS folder structure** with **simple one-liner examples** for each important file — perfect for interview + practical understanding.

✅ Playwright JS Recommended Folder Structure

```
playwright-project/
│
├── tests/
│   └── login.spec.js
│
├── pages/
│   └── LoginPage.js
│
├── fixtures/
│   └── testData.js
│
├── utils/
│   └── helpers.js
│
├── playwright.config.js
│
├── package.json
│
└── README.md
```

📂 tests/ → Test Files

👉 Contains actual test cases.

login.spec.js (one-liner example)

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

🗁 pages/ → Page Object Model (POM)

👉 Stores reusable page actions.

LoginPage.js (one-liner example)

```
login = async () => await this.page.click('#loginBtn');
```

🗁 fixtures/ → Test Data

👉 Stores test data/constants.

testData.js (one-liner example)

```
export const user = { username: 'admin', password: '1234' };
```

🗁 utils/ → Helper Functions

👉 Common reusable utilities.

helpers.js (one-liner example)

```
export const wait = ms => new Promise(r => setTimeout(r, ms));
```

⚙ playwright.config.js → Configuration

👉 Controls browser setup, base URL, reporters.

One-liner example

```
export default { use: { browserName: 'chromium' } };
```

📦 package.json → Dependencies

👉 Manages Playwright installation.

One-liner example

```
"scripts": { "test": "playwright test" }
```

▯ How to Run Tests (One-liner)

```
npx playwright test
```

✅ Simple Real-World Structure (Interview Friendly)

tests → test cases

pages → page objects

fixtures → test data

utils → helpers

config → browser settings

👉 Short explanation for interview:

"In Playwright JS, we follow a modular folder structure where tests contain test cases, pages hold reusable page objects, fixtures manage test data, utils store helper functions, and config controls execution settings."
