Introduction to Playwright

What is Playwright?

Playwright is a modern open-source automation testing framework developed by Microsoft for automating web/API applications.

It helps testers and developers automate browser actions just like a real user:

- Clicking buttons
- Entering text
- Selecting dropdown values
- Uploading files
- Navigating between pages
- Validating application behavior

Playwright is mainly used for:

- End-to-End (E2E) Testing
- UI Testing
- Regression Testing
- Cross-Browser Testing
- API Testing

Why Do We Need Playwright?

In today's world, applications are becoming:

- More dynamic
- More interactive
- Faster changing

Examples:

- React Applications
- Angular Applications
- Vue Applications

Traditional automation tools often face:

- Synchronization issues
- Flaky tests
- Slow execution

Playwright was designed to solve these challenges.

Key Features of Playwright

1. Cross-Browser Testing

One script can run on:

- Chromium (Chrome, Edge)
- Firefox
- WebKit (Safari)

No code changes required.

2. Cross-Platform Support

Playwright works on:

- Windows
- Linux
- macOS

3. Auto Waiting

Playwright automatically waits before performing actions.
Example:
When clicking a button, Playwright checks:

- Is the element present?
- Is it visible?
- Is it enabled?
- Is it stable?

This reduces test failures.

4. Fast Execution

Playwright directly communicates with browser engines, making execution faster than many traditional tools.

5. Parallel Execution

Multiple test cases can run simultaneously.
Benefits:

- Faster test execution
- Reduced CI/CD build time

6. Headless and Headed Execution

Headed Mode

- Browser is visible
- Useful for learning and debugging

Headless Mode

- Browser runs in background
- Faster execution
- Ideal for CI/CD pipelines

7. Mobile Emulation
Playwright can simulate mobile devices like:

- iPhone
- Samsung Galaxy
- iPad

without requiring real devices.

8. Multiple Language Support
Playwright supports:

- JavaScript
- TypeScript
- Java
- Python
- C#

Playwright Architecture

Test Script
↓
Playwright API
↓
Browser
↓
Web Application

Playwright acts as a bridge between your test script and the browser.

Why Is Playwright Becoming Popular?

Because it provides:
1. Auto Waiting
2. Faster Execution
3. Less Flaky Tests
4. Easy Parallel Execution
5. Modern Browser Support
6. API Testing Support
7. Excellent Reporting and Debugging

Q: What is Playwright?
Answer:
Playwright is an open-source automation testing framework developed by Microsoft that enables end-to-end testing of web applications across Chromium, Firefox, and WebKit browsers. It supports multiple programming languages, automatic waiting, parallel execution, mobile emulation, and API testing.

Feature	Selenium	Playwright
Auto Wait	No	Yes
Speed	Moderate	Fast
Parallel Execution	Complex	Easy
API Testing	No	Yes
Trace Viewer	No	Yes
Network Mocking	Limited	Excellent

Browser Context	No	Yes
Mobile Emulation	Limited	Built-in
Modern Web Apps	Good	Excellent

1. Installation of Playwright

Prerequisites
Before installing Playwright, ensure the following are installed:
Node.js
Verify installation:
node -v
npm -v

Example Output:
v22.0.0
10.9.0

Create a New Playwright Project
npm init playwright@latest

When prompted:
✔ Do you want to use TypeScript or JavaScript?
→ JavaScript

✔ Name of your Tests Folder?
→ tests

✔ Add GitHub Actions workflow?
→ Yes/No

✔ Install Playwright browsers?
→ Yes

Install Browsers Separately(Optional)

npx playwright install

Install specific browser:

npx playwright install chromium

Verify Installation
npx playwright test

If installation is successful, sample tests will execute.

2. Playwright Project Structure
After installation, you'll see something like:

playwright-project

```
│
├── node_modules
│
├── tests
│   └── example.spec.js
│
├── playwright.config.js
│
├── package.json
│
├── package-lock.json
│
└── test-results
```

Important Files and Folders
tests/
Contains all test scripts.
Example:

tests

```
│
├── login.spec.js
├── search.spec.js
└── checkout.spec.js
```

playwright.config.js
Main configuration file.
Used for:

- Browser configuration
- Timeouts
- Base URL
- Screenshots
- Video recording
- Parallel execution

Example:

```
module.exports = {
  use: {
    browserName: 'chromium',
    headless: false
  }
};
```

package.json
Contains:

- Project information
- Dependencies
- Scripts

Example:

```
{
  "name": "playwright-project",
  "dependencies": {
    "@playwright/test": "^1.50.0"
  }
}
```

test-results/
Stores:

- Screenshots
- Videos
- Traces
- Execution results

Generated automatically after test execution.

Playwright Test File Naming Convention
Playwright recognizes:
*.spec.js
*.test.js

Examples:
login.spec.js
homepage.test.js

3. First Playwright Test Script
Create file:
tests/google.spec.js

Basic Test Structure

```
const { test, expect } = require('@playwright/test');

test('Open Google', async ({ page }) => {

    await page.goto('https://www.google.com');

});
```

Understanding the Code
Import Playwright Methods

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

- test() → Creates a test case
- expect() → Performs validations

Create Test

```
test('Open Google', async ({ page }) => {
```

- Test name = Open Google
- page represents browser tab

Open URL

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

Launches browser and navigates to URL.

First Validation Example

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

test('Verify Google Title', async ({ page }) => {

   await page.goto('https://www.google.com');

   await expect(page).toHaveTitle(/Google/);

});
```

What is await?
Playwright operations are asynchronous.
Example:

```
await page.goto();
await page.click();
await page.fill();
```

await tells JavaScript:
"Wait until this action is completed before moving to the next step."

Running the Test
Run all tests:

```
npx playwright test
```

Run specific file:
npx playwright test tests/google.spec.js

Run in Headed Mode

npx playwright test --headed

Browser becomes visible.

View Report
After execution:

npx playwright show-report

Demo Script

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


test('Launch OrangeHRM Application', async ({ page }) => {

    await page.goto('https://opensource-demo.orangehrmlive.com/');

    await expect(page).toHaveTitle(/OrangeHRM/);

});
```

Program 1

```
// Import required modules from Playwright
import { test, expect } from "@playwright/test";

// ================= BASIC UNDERSTANDING =================

// test --> Used to define a single test case
// expect --> Assertion library used to validate results

// page --> Represents a browser tab (like WebDriver in Selenium)
// Playwright automatically provides "page" using fixtures

// async --> Marks function as asynchronous (returns a Promise)
// Promise --> Represents future result (Pending / Fulfilled / Rejected)

// await --> Waits for async operation to complete before moving ahead
//My first test - Test Name
// ================= TEST CASE =================
```

```
test("My first test", async ({ page }) => {

  // Step 1: Navigate to a URL
  await page.goto("https://www.facebook.com/");

  // Step 2: Get page title
  let title = await page.title();

  // Step 3: Print title in console
  console.log("Page title :", title);

});



// 🔷 1. Run all tests
// npx playwright test

// 🔷 2. Run a specific test file
// npx playwright test tests/example.spec.js

// 🔷 3. Run tests in headed mode (browser visible)
// npx playwright test --headed


// 🔷 3. Install Playwright
// npm init playwright@latest

// 👉 This will:

// Install Playwright
// Install browsers (Chromium, Firefox, WebKit)
// Create sample tests
// Create playwright.config.js

//🔷 4. Install Browsers Manually(if needed)
//npx playwright install

/*
_____________________________________________________

JavaScript Async Concepts (Important Notes)
_____________________________________________________

1️⃣ async
  - The 'async' keyword is used before a function to make it asynchronous.
  - An async function always returns a Promise.
```

- It allows asynchronous operations like network calls or page loads
  without blocking the rest of the code.

2️⃣ Promise
- A Promise is a built-in JavaScript object representing a future value.
- It can be in one of three states:
  Fulfilled → Operation completed successfully
  Rejected → Operation failed
  Pending → Operation still in progress

3️⃣ await
- The 'await' keyword is used only inside async functions.
- It pauses the function execution until the Promise is resolved or rejected.
- Example:
  await page.goto("https://www.facebook.com/");
  This line waits until the page has fully loaded before continuing.

---

4. In Playwright, page represents a single browser tab or window and is used to
perform all user interactions on a web application.
*/

Program 2

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

// page.goto()
// Used to navigate to the specified URL.

// page.waitForTimeout()
// Static wait.
// Pauses execution for specified milliseconds.
// Not recommended in real projects.
// Prefer Playwright auto-waiting and assertions.

// page.title()
// Returns the title of the current web page.

test("Test Demo", async ({ page }) => {

   // Navigate to OrangeHRM Login Page
   await page.goto(
      'https://opensource-demo.orangehrmlive.com/web/index.php/auth/login'
   );

   // Wait for 2 seconds
```

```
    await page.waitForTimeout(2000);

    // Capture page title
    let pageTitle = await page.title();

    // Print title in console
    console.log("Page Title :", pageTitle);

    // Validate page title
    await expect(page).toHaveTitle(pageTitle);

});

// Notes:

// Default Assertion Timeout = 5 Seconds

// Example:
// await expect(page).toHaveTitle("OrangeHRM");

// Default Test Timeout = 30 Seconds

// page.title()
// Returns current page title as String.

// Output Example:
// Page Title : OrangeHRM
```

Program 3

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

// Verify Page Title

test("Home Page Test", async ({ page }) => {

    // Navigate to OrangeHRM Login Page
    await page.goto(
        "https://opensource-demo.orangehrmlive.com/web/index.php/auth/login"
    );

    // Get page title
    let title = await page.title();

    console.log("Page Title :", title);
```

```
    // Validate page title
    await expect(page).toHaveTitle(title);
});


// Verify Page URL

test("Page URL", async ({ page }) => {

    // Navigate to OrangeHRM Login Page
    await page.goto(
        "https://opensource-demo.orangehrmlive.com/web/index.php/auth/login"
    );

    // Validate current URL
    await expect(page).toHaveURL(
        "https://opensource-demo.orangehrmlive.com/web/index.php/auth/login"
    );
});


// test.only()
// Executes only this test case.
// All other tests will be ignored.

test.only("Page URL 1", async ({ page }) => {

    // Navigate to OrangeHRM Login Page
    await page.goto(
        "https://opensource-demo.orangehrmlive.com/web/index.php/auth/login"
    );

    // Validate current URL
    await expect(page).toHaveURL(
        "https://opensource-demo.orangehrmlive.com/web/index.php/auth/login"
    );
});


// Notes:

// test.only()
// Runs only selected test case.

// test.skip()
// Skips a test case.

// page.goto()
```

```
// Opens application URL.

// page.title()
// Returns page title.

// expect()
// Used for validation/assertion.

// toHaveTitle()
// Validates page title.

// toHaveURL()
// Validates page URL.

// Default Test Timeout = 30 Seconds

// Default Assertion Timeout = 5 Seconds

// Playwright provides Auto-Waiting
// so explicit waits are usually not required.
```

Program 4

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

// test   -> Used to define a test case.
// expect -> Used to perform assertions (verification/validation).

test("TC001", async ({ page }) => {

    // Wait for 5 seconds (Not recommended in real projects)
    await page.waitForTimeout(5000);

    // Navigate to the Registration page
    await page.goto("https://demo.automationtesting.in/Register.html");

    // Wait for 5 seconds to observe the page
    await page.waitForTimeout(5000);

});

test.only("Complete navigation flow on Facebook", async ({ page }) => {

    // ============================
    // Step 1: Navigate to Facebook
    // ============================
    await page.goto("https://www.facebook.com/");
```

```
console.log("Step 1 - Open Facebook page: " + page.url());

// Capture the page title for later verification
const facebookTitle1 = await page.title();

console.log("Facebook Page Title: " + facebookTitle1);

// Verify page title and URL
// (Replace "Test" with the actual Facebook title)
await expect(page).toHaveTitle("Test");
await expect(page).toHaveURL("https://www.facebook.com/");

console.log("Facebook page opened successfully. Title and URL verified.");

await page.waitForTimeout(3000);

// ============================
// Step 2: Navigate to SauceDemo
// ============================
await page.goto("https://www.saucedemo.com/");

console.log("Step 2 - Open SauceDemo page: " + page.url());

await page.waitForTimeout(3000);

// Capture the SauceDemo page title
const sauceDemoTitle1 = await page.title();

console.log("SauceDemo Page Title: " + sauceDemoTitle1);

// Verify title and URL
await expect(page).toHaveTitle(sauceDemoTitle1);
await expect(page).toHaveURL("https://www.saucedemo.com/");

console.log("SauceDemo page opened successfully. Title and URL verified.");

await page.waitForTimeout(3000);

// ============================
// Step 3: Navigate Back
// ============================
await page.goBack();

console.log("Navigated back to Facebook page.");

await page.waitForTimeout(3000);
```

```
    // Verify Facebook page after navigating back
    await expect(page).toHaveTitle(facebookTitle1);

    await page.waitForTimeout(3000);

    // ============================
    // Step 4: Navigate Forward
    // ============================
    await page.goForward();

    console.log("Navigated forward to SauceDemo page.");

    await page.waitForTimeout(3000);

    // Verify SauceDemo page after navigating forward
    await expect(page).toHaveTitle(sauceDemoTitle1);

    // ============================
    // Step 5: Reload the Current Page
    // ============================
    await page.reload();

    console.log("Reloaded the current page.");

    await page.waitForTimeout(3000);

    console.log("Navigation flow completed successfully.");

});


//Test timeout - 30 Sec
//Assertion timeout - 5 Sec
//Navigation timeout - 30 Sec
//Action timeout - 0 (Auto waiting)
```

Q: Can we minimize the browser using Playwright?
Answer:
No, Playwright does not provide a built-in API to minimize the browser window. It supports browser launch options, maximizing, and viewport resizing, but browser minimization is not directly supported.
This is a common difference between **Selenium** and **Playwright**.

What is a Locator?

A Locator is a mechanism used to identify and interact with web elements on a webpage.

In automation testing, before performing any action such as click, type, select, hover, or validate, we must first locate the element.

Examples:

- Username Textbox
- Password Textbox
- Login Button
- Search Box
- Product Name
- Add To Cart Button

Without locators, Playwright cannot identify elements on the webpage.

Why are Locators Important?

Locators help automation scripts:

1. Find web elements
2. Perform actions on elements
3. Validate UI elements
4. Handle dynamic web pages
5. Build reliable automation scripts

Real-Time Example:

Manual Testing:

Open Application
↓
Find Username Field
↓
Enter Username

Automation Testing:

Locate Username Field
↓
Enter Username

Locator Example:

```
await page.locator("#username").fill("Admin");
```

How Playwright Finds Elements?

Flow:

Web Page
↓
HTML DOM
↓
Locator
↓
Playwright Action

Playwright Recommended Locators

1. CSS Selector 2. Xpath 3. Bulit in locators

1. getByRole()
2. getByLabel()
3. getByPlaceholder()
4. getByText()
5. getByAltText()
6. getByTitle()
7. getByTestId()
8. XPath
9. CSS

1. What is XPath?

XPath (XML Path Language) is a locator strategy used to find and identify web elements in the HTML DOM structure.

XPath helps automation tools like Playwright and Selenium locate elements when ID, Name, or other attributes are not available.

Why XPath is Used?

- Locate web elements
- Handle dynamic elements
- Navigate through HTML structure
- Find parent, child, sibling elements
- Perform complex element identification

Example HTML
<input id="username" type="text" placeholder="Username">

//*[@id='username']

await page.locator("//*[@id='username']").fill("Admin");

Types of XPath

There are two types of XPath:

1. Absolute XPath
2. Relative XPath

1. Absolute XPath

Definition

Absolute XPath starts from the root node (html) and follows the complete path to reach the target element.

/html/body/div/form/input

Disadvantages

- Very lengthy
- Not stable
- Breaks if page structure changes
- Not recommended in automation projects

2. Relative XPath

Definition

Relative XPath starts from anywhere in the DOM and uses attributes or relationships to locate elements.

//tagname[@attribute='value']

Advantages

- Short and readable
- Stable
- Easy to maintain
- Most commonly used in automation

Absolute XPath vs Relative XPath

Absolute XPath	Relative XPath
Starts from root node	Starts from any node

Long path	Short path
Fragile	Stable
Difficult to maintain	Easy to maintain
Not recommended	Recommended
Breaks when DOM changes	More reliable

Which XPath is preferred in Automation Testing?

Answer:

Relative XPath is preferred because:

- More stable
- Easy to maintain
- Less dependent on page structure
- Better suited for dynamic web applications

Absolute XPath should generally be avoided in real-time automation frameworks.

Relative XPath SubTypes

1. XPath by Attribute

```
//input[@id='username']
//input[@name='username']
//button[@type='submit']
```

2. XPath by Text

```
//button[text()='Login']
//a[text()='WebTable']
```

3. XPath by Contains Text

```
//button[contains(text(),'Login')]
//span[contains(text(),'Create')]
```

4. XPath by Contains Attribute

```
//input[contains(@id,'user')]
//button[contains(@class,'btn')]
```

5. XPath by Index
(AnyXapth)[1]

```
(Anyxapth)[2]
```

CSS Selectors

What is a CSS Selector?

A **CSS Selector** is a pattern used to identify and locate HTML elements on a web page.

Originally, CSS selectors were designed to apply styles to HTML elements. However, automation tools like Playwright and Selenium also use CSS selectors to locate web elements for interaction.

Examples of interactions:

- Click a button
- Enter text into a textbox
- Select a checkbox
- Choose a value from a dropdown
- Read text from a webpage

Why Do We Use CSS Selectors in Automation?

In automation, before performing any action, the tool must first identify the web element.

For example:

Login Button
↓
Locate the Button
↓
Perform Click Action

CSS selectors provide a fast and reliable way to locate these elements.

Advantages of CSS Selectors

- Simple and easy to learn.
- Faster than XPath in most cases.
- Shorter syntax.
- Easy to maintain.
- Supported by all modern browsers.
- Excellent for locating elements using IDs, classes, and attributes.

CSS Selector Syntax

A CSS selector consists of one or more parts that identify an HTML element.

Example: #username
means
Select the element whose ID is **username**.

Types of CSS Selectors

1. ID Selector

Syntax

#idValue

HTML

<input id="username">

Playwright

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

Explanation

- # represents an ID.
- IDs should be unique within a webpage.
- Fastest and most commonly used selector when available.

2. Tag Name with ID

Syntax - tagName#idValue

HTML

<input id="username">

Playwright

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

Explanation

This selector specifies both the tag name and the ID, making it more specific.

3. Class Selector

Syntax - .className

HTML

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

Playwright

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

Explanation

- . represents a class.
- Multiple elements can share the same class.

Example:

```
<input class="textbox">
<input class="textbox">
<input class="textbox">
```

The class selector may match multiple elements.

4. Tag Name with Class

Syntax - tagName.className

HTML

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

Playwright

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

Explanation

This selector combines the tag name and class name to make the locator more specific.

5. Attribute Selector

Syntax- [attributeName=“attributeValue"]

HTML

```
<input name="email">
```

Playwright

```
await page.locator('[name="email"]').fill("test@gmail.com");
```

Explanation

Selects an element based on the value of an attribute.

Common attributes:

- id
- name
- type
- placeholder
- value
- data-testid
- data-test

Examples: [type="password"]
[name="username"]
[data-test="login-button"]

6. Tag Name with Attribute Selector

Syntax - tagName[attributeName=“attributeValue"]

HTML

<input name="email">

Playwright

await page.locator('input[name="email"]').fill("Admin");

Explanation

This selector uses both the tag name and an attribute, making it more precise.

Playwright Built-in Locators

What are Built-in Locators?

Playwright provides several **built-in locators** that allow you to locate web elements based on how users interact with the application instead of relying only on CSS or XPath.

These locators are:

- More readable
- More reliable
- Less likely to break
- Recommended by Microsoft Playwright

1. getByRole() - (First Preference)

When to Use

Use getByRole() when interacting with elements that have an Accessible Rich Internet Applications (**ARIA) role**.

Examples:

- Button
- Link
- Heading
- Textbox
- Checkbox
- Radio Button

Example HTML

<button>Login</button>

Syntax

page.getByRole(role, { name: "Accessible Name" })

Playwright

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

Best For

- Buttons
- Links

- Headings
- Checkboxes
- Radio Buttons
- Dropdowns

Common Roles

1. Textbox

await page.getByRole("textbox", { name: "Username" }).fill("Admin");

await page.getByRole("textbox", { name: "Password" }).fill("admin123");

2. Button

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

3. Link

await page.getByRole("link", { name: "Forgot your password?" }).click();

4. Heading

await expect(page.getByRole("heading", { name: "Dashboard" })).toBeVisible();

5. Checkbox

await page.getByRole("checkbox", { name: "Remember Me" }).check();

6. Radio Button

await page.getByRole("radio", { name: "Male" }).check();

7. Image

await expect(page.getByRole("img", { name: "Company Logo" })).toBeVisible();

2. getByLabel()

Definition

Locates form controls using their associated label.

Example

<label>Username</label>

<input>

Playwright

await page.getByLabel("Username").fill("Admin");

When should you use?

Whenever working with forms.

Examples

- Username
- Password
- Email
- Phone Number

3. getByPlaceholder()

Definition

Locates input elements using placeholder text.

Example

<input placeholder="Search">

Playwright

await page.getByPlaceholder("Search").fill("Laptop");

4. getByText()

Definition

Locates element using visible text.

Example

<button>Login</button>

Playwright

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

5. getByAltText()

Definition

Locates images using alt attribute.

Example

<img alt="Company Logo">

Playwright

await page.getByAltText("Company Logo");

6. getByTitle()

Definition

Locates element using title attribute.

Example

<button title="Delete">

Playwright

await page.getByTitle("Delete").click();

7. getByTestId()

Definition

Locates elements using custom test id.

Example

<button data-testid="login-btn">

Playwright

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

1. What is Codegen?

Codegen stands for Code Generator.

Playwright Codegen is a built-in Playwright tool that **records your interactions with a web application and automatically generates Playwright test code**.

Instead of manually writing:

```
await page.getByRole('textbox', { name: 'Username' }).fill('Admin');
await page.getByRole('textbox', { name: 'Password' }).fill('admin123');
await page.getByRole('button', { name: 'Login' }).click();
```

you can perform those actions in the browser, and Codegen generates the corresponding Playwright code.

2. Why do we use Codegen?

Codegen is mainly useful for:

- Learning Playwright syntax
- Quickly creating initial test scripts
- Finding locators
- Exploring a new application
- Understanding how Playwright identifies elements
- Creating a starting point for automation

Important

Codegen is a starting point, not a replacement for framework development or manual coding.

In a real project, we normally **review and improve the generated code**.

3. How Codegen Works

The basic flow is:

```
Run Codegen
    ↓
Browser Opens
    ↓
Open Application
    ↓
Perform Actions
    ↓
Playwright Records Actions
```

↓
Generates Code

For example:

Open OrangeHRM
↓
Enter Username
↓
Enter Password
↓
Click Login
↓
Open Dashboard

Codegen may generate something like:

```
await page.goto('https://opensource-demo.orangehrmlive.com/');
await page.getByPlaceholder('Username').fill('Admin');
await page.getByPlaceholder('Password').fill('admin123');
await page.getByRole('button', { name: 'Login' }).click();
```

4. How to Start Codegen

Open your terminal inside the Playwright project.

Run:

```
npx playwright codegen
```

A browser window will open along with the Playwright Inspector.

5. Codegen with a URL

You can directly open a website:

```
npx playwright codegen https://www.saucedemo.com
```

For example:

```
npx playwright codegen https://opensource-demo.orangehrmlive.com/
```

Now you can interact with the application.

📘 Playwright Dropdown Handling

🔷 What is a Dropdown?

A dropdown is a UI element that displays a list of options. The user selects one or more values from the available options.

Example:
Country
▼
India
USA
Canada
Australia

🔷 Why Do We Use Dropdowns?

✔ Reduce user typing
✔ Prevent invalid input
✔ Improve user experience
✔ Maintain consistent data

🔷 Types of Dropdowns

1. Static Dropdown (HTML Select)
2. Dynamic Dropdown (Auto Suggestion)
3. Multi-Select Dropdown
4. Custom Dropdown (Bootstrap/React/Angular)
5. Searchable Dropdown

1. Static Dropdown

A static dropdown is created using the HTML <select> tag.

Example:

```
<select id="country">
    <option>India</option>
    <option>USA</option>
    <option>Canada</option>
</select>
```

Characteristics:
✔ Uses <select> tag
✔ Contains <option> tags

✔ Easy to automate
✔ Supports Playwright selectOption()

2. Dynamic Dropdown

A dynamic dropdown loads options while the user types or after clicking the dropdown.

Example:
Google Search
Amazon Search
MakeMyTrip City Search

Characteristics:
✔ No <select> tag
✔ Options appear dynamically
✔ Need click + type + select

3. Multi-Select Dropdown

Allows selecting multiple values.

Example:

```
<select multiple>
   <option>Java</option>
   <option>Python</option>
   <option>JavaScript</option>
</select>
```

Used for:
Skills
Languages
Interests

4. Searchable Dropdown

User types value to filter options.

Examples:
Country Search
Employee Search
Product Search

Automation Steps:

1. Click dropdown
2. Enter text
3. Click matching option

How to Identify Dropdown Type?

Step 1:
Inspect Element (F12)

If you see:
<select>
👉 Static Dropdown

If you see:

👉 Custom/Dynamic Dropdown

Playwright Methods

Static Dropdown
selectOption()

Dynamic Dropdown
click()
fill()
click()

Custom Dropdown
click()
locator()
click()

Different Ways to Select Option

By Visible Text

```
await page.locator("#country")
.selectOption({ label: "India" });
```

By Value

```
await page.locator("#country")
.selectOption({ value: "IN" });
```

By Index

```
await page.locator("#country")
.selectOption({ index: 2 });
```

Common Methods

selectOption()
locator()
click()
fill()
press()
nth()

Interview Questions

Q1. What is a dropdown?

A dropdown is a UI element that allows users to select one or more values from a list.

Q2. How many types of dropdowns are there?

1. Static Dropdown
2. Dynamic Dropdown
3. Multi-Select Dropdown

Q3. How do you identify a static dropdown?

By inspecting the HTML.
If it contains:
<select>
It is a static dropdown.

Q4. Which Playwright method is used for static dropdown?
selectOption()

Q5. Can we use selectOption() for custom dropdown?
No.
Custom dropdowns do not use the HTML <select> element.
We use click() and locators.

Q6. What is the difference between Static and Dynamic Dropdown?

Static Dropdown
✔ Uses <select>
✔ Fixed options
✔ selectOption()

Dynamic Dropdown
✔ No <select>
✔ Options loaded dynamically
Program 1

Playwright Screenshots

What is a Screenshot?

A Screenshot is an image captured during the execution of a test. It records the current state of the web page or a specific web element. Screenshots help testers verify the application's UI, identify failures, and provide visual evidence of test execution.

Screenshots are one of the most commonly used debugging and reporting features in Playwright.

==========
Why Do We Need Screenshots?
==========

Screenshots are useful because they:

✔ Capture the exact state of the application.
✔ Help identify UI issues.
✔ Provide proof that a test executed successfully.
✔ Make debugging easier when tests fail.
✔ Help compare expected and actual UI.
✔ Can be attached to test reports such as Allure or HTML Reports.
✔ Help developers understand failures quickly.

==========
Types of Screenshots in Playwright
==========

1. Page Screenshot
2. Full Page Screenshot
3. Element Screenshot
4. Screenshot on Failure

==========

1. Page Screenshot

==========

A Page Screenshot captures only the visible portion of the webpage currently displayed in the browser window.

It does not capture the content that requires scrolling.

Use Cases:
• Capture the login page.
• Capture the dashboard.
• Capture confirmation pages.
• Store UI evidence.

Advantages:
✔ Fast
✔ Small image size
✔ Suitable for most UI validations

==========

2. Full Page Screenshot

==========

A Full Page Screenshot captures the entire webpage, including the content that is not visible on the screen and requires scrolling.

Playwright automatically scrolls through the page and combines everything into one image.

Use Cases:
• Long forms
• Product pages
• Reports
• Documentation pages
• Landing pages

Advantages:
✔ Captures complete webpage
✔ Useful for visual testing

✔ Better for UI comparison

==========

3. Element Screenshot

==========

An Element Screenshot captures only a specific web element instead of the entire webpage.

Examples:
• Logo
• Button
• Image
• Table
• Login Form
• Product Card

Use Cases:
• Validate logos
• Capture charts
• Verify images
• Compare UI components

Advantages:
✔ Small image size
✔ Focuses only on required element
✔ Useful for UI component validation

==========

4. Screenshot on Failure

==========

A Screenshot on Failure is captured automatically whenever a test fails.

Instead of manually checking what happened, the screenshot shows the exact browser state at the moment of failure.

Usually implemented using:

• try-catch block
• Playwright Test Hooks
• Playwright Configuration

Use Cases:
• Test debugging
• Defect reporting
• Failure analysis
• Automation reports

Advantages:
✔ Saves debugging time
✔ Provides visual proof
✔ Easy to share with developers
✔ Improves defect analysis

==========

Screenshot File Formats

==========

Playwright supports:

✔ PNG (Default)
✔ JPEG

==========

Where are Screenshots Stored?

==========

Screenshots are saved inside the project folder.

Example:

Project
│
├── tests
├── screenshots

```
│   ├── Login.png
│   ├── Dashboard.png
│   ├── Failure.png
│   └── Logo.png
├── playwright.config.js
└── package.json
```

==========

Benefits of Screenshots

==========

✔ Easy debugging
✔ Better reporting
✔ UI verification
✔ Failure investigation
✔ Evidence for automation execution
✔ Supports visual regression testing

==========

Screenshot Configuration in Playwright

==========

Playwright allows you to automatically capture screenshots during test execution by configuring the playwright.config.js file.

Instead of writing page.screenshot() in every test, Playwright can capture screenshots automatically.

This feature is useful for:

✔ Failed Test Cases
✔ Passed Test Cases
✔ Retry Attempts
✔ Debugging
✔ Automation Reports

==========
Screenshot Configuration Location
==========

File Name:

playwright.config.js

==========
Screenshot Options
==========

Playwright provides three screenshot options.

1. "off"
2. "on"
3. "only-on-failure"

==========
1. screenshot: "off"
==========

Definition:

No screenshots will be captured during test execution.

Use Case:

• Faster execution
• No image generation
• Saves disk space

Configuration:

```
use: {
   screenshot: "off"
}
```

==========
2. screenshot: "on"
==========

Definition:

Playwright captures a screenshot for every test,
whether it passes or fails.

Use Case:

• UI Verification
• Training
• Demo Projects
• Visual Evidence

Configuration:

```
use: {
   screenshot: "on"
}
```

==========
3. screenshot: "only-on-failure"
==========

Definition:

Playwright captures screenshots only when a test fails.

This is the most commonly used option in
real-time automation projects.

Advantages:

✔ Saves storage
✔ Faster execution

✔ Easy debugging

Configuration:

```
use: {
   screenshot: "only-on-failure"
}
```

==========
Full Example
==========

```
module.exports = defineConfig({

   use: {
      screenshot: "only-on-failure"
   }

});
```

==========
Difference Between Manual and Automatic Screenshot
==========

Manual Screenshot

✔ Written inside test scripts
✔ Captures whenever developer wants
✔ Uses page.screenshot()

Automatic Screenshot

✔ Configured in playwright.config.js
✔ No code required inside test
✔ Automatically captured by Playwright

Q1. What is a Screenshot in Playwright?

A Screenshot is an image captured during test execution that records the current state of a webpage or a specific web element.

Q2. What are the different types of screenshots in Playwright?

- Page Screenshot
- Full Page Screenshot
- Element Screenshot
- Screenshot on Failure

Q3. What is the difference between Page Screenshot and Full Page Screenshot?

Page Screenshot:
Captures only the visible portion of the webpage.

Full Page Screenshot:
Captures the complete webpage, including the scrollable area.

Q4. Why do we capture screenshots on test failure?

To identify the exact state of the application when the test failed, making debugging faster and easier.

Q5. Which screenshot format does Playwright use by default?

PNG

Q6. Can Playwright capture screenshots of a specific element?

Yes. Playwright can capture screenshots of individual web elements.

Q7. What are the advantages of screenshots in automation testing?

• Visual evidence
• Easy debugging
• Better reporting
• Faster defect analysis
• UI validation

Q8. Where is screenshot configuration done?

Answer:

In playwright.config.js or playwright.config.ts.

Q9. What are the screenshot options available in Playwright?

• off
• on
• only-on-failure

Q10. Which screenshot option is mostly used in real-time projects?

only-on-failure

Q11. What is the difference between manual and automatic screenshots?

Manual screenshots are captured using
page.screenshot() inside test scripts.

Automatic screenshots are captured using
the screenshot configuration in playwright.config.js.

Q12. Why is "only-on-failure" recommended?

- Saves storage
- Faster execution
- Easier debugging
- Generates screenshots only when required.

PLAYWRIGHT ALERTS (JAVASCRIPT DIALOGS)

What is an Alert?

Definition

An Alert (also called a **JavaScript Dialog**) is a browser-generated popup window used to display information, request confirmation, or collect user input.

Unlike HTML popups, alerts are created by the **browser**, not by HTML elements.

Important: Browser alerts are **not part of the HTML DOM**, therefore Playwright locators cannot interact with them.

Why Do We Need Alert Handling?

When an alert appears:

✅ Browser execution pauses.

✅ User cannot click anywhere on the page.

✅ Keyboard input is blocked.

✅ Test execution waits until the alert is handled.

If Playwright does not handle the alert, the automation test cannot continue.

JavaScript Functions Used to Create Alerts

Function	Purpose
`alert()`	Display information
`confirm()`	Ask for confirmation
`prompt()`	Accept user input

Types of Browser Alerts

1️⃣ Simple Alert

Purpose

Displays a message with only an **OK** button.

JavaScript

```
alert("Login Successful");
```

Buttons

✔ OK

Common Uses

- Login Success
- Payment Success
- Warning Messages
- Validation Messages

2️⃣ Confirmation Alert

Purpose

Asks the user to confirm an action.

JavaScript

```
confirm("Delete this record?");
```

Buttons

✔ OK

✔ Cancel

Common Uses

- Delete Record
- Logout
- Exit Application
- Remove Product

3️⃣ Prompt Alert

Purpose

Allows the user to enter text.

JavaScript

```
prompt("Enter your name");
```

Buttons

✔ OK

✔ Cancel

Input Box

✔ Available

Common Uses

- OTP
- Employee ID
- Username
- Coupon Code

Playwright Alert Handling

Playwright handles browser alerts using the **dialog** event.

Syntax

```
page.on("dialog", async (dialog) => {
});
```

Dialog Methods

Method	Purpose
`accept()`	Click OK
`dismiss()`	Click Cancel
`message()`	Read alert text
`type()`	Return dialog type
`defaultValue()`	Read prompt default value

Alert Handling Flow

Click Button
↓
JavaScript Creates Alert
↓
Browser Displays Alert
↓
Playwright Detects Dialog
↓
Read Message
↓
Accept / Dismiss
↓
Continue Execution

Browser Alert vs HTML Popup

Browser Alert	HTML Popup
Browser Generated	HTML Generated
Outside DOM	Inside DOM
Cannot Inspect	Can Inspect
Use dialog Event	Use Playwright Locators

Blocks Browser	Doesn't Block Browser

Best Practices

✅ Register page.on("dialog") before clicking.

✅ Validate the alert message.

✅ Handle only expected alerts.

✅ Use dismiss() only when required.

✅ Validate the application after closing the alert.

Common Mistakes

❌ Clicking before registering the listener.

❌ Using locator() on browser alerts.

❌ Using XPath to locate alerts.

❌ Forgetting to call accept() or dismiss().

❌ Using waitForSelector() for browser alerts.

Interview Questions

Q1. What is an Alert?

A browser-generated popup used to display information, request confirmation, or receive user input.

Q2. Which Playwright event handles alerts?

page.on("dialog")

Q3. Which method clicks OK?

dialog.accept()

Q4. Which method clicks Cancel?

dialog.dismiss()

Q5. How do you read the alert message?

dialog.message()

Q6. How do you identify the alert type?

dialog.type()

Q7. How do you enter text into a Prompt dialog?

dialog.accept("Playwright")

Q8. Why can't locators handle browser alerts?

Because browser alerts are **outside the HTML DOM**.

Question: *Why do we write page.on("dialog") before clicking the button that opens the alert?*

Answer:

"In Playwright, alerts are handled using the dialog event. Since a JavaScript alert appears immediately after the user action, Playwright must already be listening for that event before the alert is triggered.

The page.on('dialog') method registers an event listener. When we click a button that opens an alert, the browser immediately creates the dialog and Playwright fires the dialog event. If the listener is already registered, it catches the event and allows us to handle the alert using methods like dialog.accept() or dialog.dismiss().

If we register the listener after clicking the button, the alert may already have appeared and the event has already fired. In that case, the listener misses the event because events cannot be captured after they have occurred.

That's why the best practice is to always register the dialog listener before performing the action that triggers the alert."

Or

"We register page.on('dialog') before clicking because alerts appear immediately after the action. The dialog event is fired as soon as the browser creates the alert. If the listener is registered after the click, it misses the event since it has already occurred. Therefore, Playwright recommends registering the listener first so it can immediately handle the alert using accept() or dismiss()."

If the Interviewer Asks: "What Happens Internally?"

You can answer:

"Internally, Playwright communicates with the browser using the browser automation protocol. When a JavaScript dialog is created, the browser sends a dialog event to Playwright. Playwright checks whether a dialog event listener is registered. If it finds one, it executes the callback and exposes the dialog object. We can then read the message, check the dialog type, and call accept() or dismiss(). After the dialog is handled, the browser resumes normal execution."

1. Introduction to iFrames

1.1 What is an iFrame?

An iframe (short for *inline frame*) is an HTML element that lets you embed another HTML document inside the current HTML document. In simple language:

"An iframe is a page inside another page."

The outer page (the one the user opens) is called the parent document. The page loaded inside the <iframe> tag is called the child document. Each document has its own DOM, its own JavaScript context, and (usually) its own URL.

Definition

An iframe is an HTML tag <iframe> that renders an independent HTML document inside a rectangular region of the parent page. Each iframe has its own DOM tree, its own window object, and its own document object.

1.2 Purpose of an iFrame

Websites use iframes to:

1. Embed external content (YouTube videos, Google Maps, tweets).
2. Isolate untrusted content — an ad or third-party widget cannot easily read the parent page.
3. Split a large single-page app into smaller independent pieces.
4. Reuse a form or page (e.g., a payment page) across many sites.
5. Improve security using the same-origin policy — cross-origin scripts cannot read across boundaries.

1.4 Why Browsers Support iFrames

- To let one site safely include content from another site.
- To keep DOM, cookies, storage, and JavaScript separated for security.
- To let advertisement and analytics vendors ship code without affecting the host page.

1.5 Why Companies Use iFrames

- Payment providers (Stripe, Razorpay, PayPal) show card fields inside their own iframe so the merchant never touches the raw card data — this reduces PCI-DSS scope.
- Video / map providers give a small <iframe> snippet — one line and the video is embedded.
- Chat widgets (Intercom, Zendesk) run inside iframes so their CSS never conflicts with the host site.

1.6 Advantages

- Isolation — CSS/JS of the child does not affect the parent.
- Security — cross-origin iframes cannot read parent DOM.
- Easy embedding — one HTML tag ships the whole feature.

- Independent reloading — an iframe can refresh without reloading the parent page.

2. Real-world Examples

Website / Feature	Why an iframe is used
Stripe / Razorpay card field	PCI-DSS: merchant page never sees card number.
Google Maps embed	Google ships map JS inside its own document to protect their APIs.
YouTube video	Reuse player, controls, ads across sites.
Intercom / Zendesk chat	Isolate widget CSS from host page.
Google AdSense	Sandbox untrusted ad code away from host page.
Google / Facebook Login popup or embed	Keep authentication in provider's origin.
Typeform / Google Forms embed	Reuse form UI inside blogs, landing pages.

3. HTML Structure of an iFrame

3.1 Syntax

```
<iframe
   src="https://example.com"
   id="paymentFrame"
   name="pay"
   title="Payment form"
   width="600"
   height="400"
   loading="lazy"
   allow="camera; microphone"
   sandbox="allow-scripts allow-forms">
</iframe>
```

3.2 Attribute Explanation

Attribute	Meaning
`src`	URL of the document to load inside the frame.
`id`	Unique DOM id — most stable locator.
`name`	Frame name — accessible via `window.frames['name']`.
`title`	Accessibility label announced by screen readers.
`width` /	Size in pixels (or CSS via style).
`loading`	`eager` (default) or `lazy` — lazy delays load until scrolled near.
`allow`	Feature policy: `camera`, `microphone`, `fullscreen`, `payment`, etc.

`sandbox`	Extra restrictions: `allow-scripts`, `allow-forms`, `allow-same-origin`. Empty value = strictest.
`referrerpolicy`	Controls what referrer is sent when the iframe loads.
`srcdoc`	Inline HTML content — no external URL required.

Interview tip: If asked **"How do you locate an iframe?"**, answer in this priority order:

id → name → CSS/XPath → index (lass).

4. How Browsers Handle iFrames

4.1 DOM Hierarchy

Every iframe becomes a new browsing context. The parent page has:

- window (parent)
- document (parent DOM)

Each iframe adds:

- window.frames[i] — child window
- iframe.contentDocument — child DOM (only accessible when same origin)

4.3 Nested iFrames

```
Parent
 └── iframe A
     └── iframe B
         └── iframe C   <-- element you want to click
```

5. Why Playwright Needs Special Handling

Consider this failing code:

```
await page.locator('#username').fill('Admin');
```

If #username is inside an iframe, this will time out with:

locator.fill: Timeout ... waiting for locator '#username' to be visible

Reason: page.locator() searches only the main frame's DOM. It does **not** cross iframe boundaries automatically. Each iframe is a separate DOM tree.

We must first "enter" the iframe using either frameLocator(), page.frame(), page.frameLocator(), or elementHandle.contentFrame().

6. How Playwright Handles iFrames

Playwright gives four main APIs:

API	Return type	Best for
`page.frameLocator(selector)`	`FrameLocator`	Recommended: lazy, auto-waiting, chainable.
`page.frame({ name / `	`Frame` or	When you already know the frame's name or URL.
`page.frames()`	`Frame[]`	Iterating / debugging all frames.
`elementHandle.contentFrame()`	`Frame`	After you already have an `elementHandle` of the `<iframe>` element.

Rule of thumb: Use frameLocator() first. Only fall back to page.frame() / contentFrame() if you specifically need a Frame object.

7. frameLocator() in Depth

7.1 Syntax

```
const frame = page.frameLocator('iframeSelector');
await frame.locator('#username').fill('Admin');
```

7.2 How It Works

- Returns a FrameLocator — a **lazy** handle. Nothing is queried yet.
- When you chain .locator(...) and then an action (click, fill, press), Playwright:
    1. Waits for the outer iframe to be attached.
    2. Enters the iframe.
    3. Waits for the inner element to be actionable.
    4. Performs the action.

Because it is lazy + auto-waiting, it is far more stable than Selenium's switchTo().frame() pattern.

7.3 Advantages

- No explicit waits needed.
- Chainable — same style as page.locator.
- Works even across cross-origin iframes.
- Re-evaluates every call, so it survives iframe re-renders.

7.5 Examples

7.5.1 By ID

```
const paymentFrame = page.frameLocator('#paymentFrame');
await paymentFrame.locator('#cardNumber').fill('4111111111111111');
```

Line by line:

- page.frameLocator('#paymentFrame') → target the iframe whose id is paymentFrame.
- .locator('#cardNumber') → inside that iframe, target the element with id cardNumber.
- .fill('4111...') → auto-waits and types.

7.5.2 By Name

```
const loginFrame = page.frameLocator('iframe[name="login"]');
await loginFrame.locator('#username').fill('Admin');
```

7.5.3 By CSS

```
const chatFrame = page.frameLocator('iframe.intercom-frame');
await chatFrame.getByRole('button', { name: 'Send' }).click();
```

7.5.4 By XPath

```
const frame = page.frameLocator('xpath=//iframe[contains(@src, "checkout")]');
await frame.locator('#pay').click();
```

7.5.5 Nested iFrames

```
const outer = page.frameLocator('#outerFrame');
const inner = outer.frameLocator('#innerFrame');
await inner.locator('#finalButton').click();
```

Just keep chaining .frameLocator(...).

8. contentFrame() in Depth

8.1 Syntax

```
const handle = await page.$('iframe#paymentFrame'); // ElementHandle
```

Or the newer locator-friendly variant:

```
const frame = await page.locator('iframe#paymentFrame').contentFrame();
await frame.locator('#cardNumber').fill('4111...');
```

8.2 Why It Returns a Frame Object

A Frame gives you the full frame API: frame.title(), frame.url(), frame.evaluate(), frame.waitForLoadState(), etc. FrameLocator intentionally hides most of these to stay simple.

8.3 When to Use It

- You need to call frame.evaluate() to run JS inside the frame.
- You need to wait for the frame's own load state (domcontentloaded, load, networkidle).
- You already have an ElementHandle from a previous step.

Q1. What is an iframe?

An HTML <iframe> element that embeds another HTML document inside the current one. Each iframe is a separate browsing context with its own window, document, cookies (for cross-origin), and DOM.

Q2. Difference between <frame> and <iframe>?

<frame> (with <frameset>) was used to split the whole window into panes and is deprecated in HTML5. <iframe> is an inline frame you can place anywhere in the body. Only <iframe> is used today.

What is an iFrame?

"An iframe (Inline Frame) is an HTML element (<iframe>) that allows one HTML page to be embedded inside another HTML page.

Each iframe has its own DOM, its own window object, and its own document, which means it behaves like a completely separate web page inside the parent page.

Since the iframe has its own DOM, the parent page cannot directly interact with the elements inside it. Before accessing any element inside an iframe, we must first switch to that iframe.

One-Minute Interview Answer (Recommended)

"An iframe, or Inline Frame, is an HTML element that allows one web page to be embedded inside another web page. Each iframe has its own DOM, window, and document, making it an independent browsing context. Because of this separate DOM, automation tools cannot directly access elements inside an iframe.

In Playwright, we handle iframes using page.frameLocator(), which is the recommended approach. It automatically waits for the iframe to load and allows us to locate elements inside it. If we need direct access to the Frame object for operations like evaluate() or checking the frame URL, we can use contentFrame() or page.frame().

What is a Checkbox?

A **Checkbox** allows users to select **one or more options** from a list.

Example:

☑ Cricket
☑ Football
☐ Hockey

A user can select **multiple checkboxes** at the same time.

What is a Radio Button?

A **Radio Button** allows users to select **only one option** from a group.

Example:

◉ Male
○ Female
○ Other

Selecting one radio button automatically deselects the previously selected option.

Checkbox Methods

1. check()

Used to select a checkbox.

Syntax

await locator.check();

2. uncheck()

Used to unselect a checkbox.

Syntax

await locator.uncheck();

3. isChecked()

Returns the checked status.

Syntax

const status = await locator.isChecked();

Returns:

- true
- false

Assertions

Validate Checked

await expect(locator).toBeChecked();

Validate Unchecked

await expect(locator).not.toBeChecked();

Radio Button Handling

Radio buttons are selected using the same method.
await locator.check();
No uncheck() method is used because radio buttons work as a group.

Playwright Mouse & Keyboard Actions

What are Mouse & Keyboard Actions?

Mouse and Keyboard Actions are advanced user interactions used to simulate real user behavior while interacting with a web application. These actions allow automation scripts to perform operations such as clicking, hovering, dragging, scrolling, typing, pressing keyboard keys, and using keyboard shortcuts.

Playwright provides **built-in methods** to perform these actions without requiring an **Actions** class. This makes Playwright automation simpler, faster, and more readable.

Why Do We Need Mouse & Keyboard Actions?

Many modern web applications contain interactive UI components that cannot be automated using only a normal click or text entry.

Mouse and Keyboard Actions help automate these complex user interactions.

Common Real-Time Scenarios

- Opening a context menu using Right Click.
- Displaying hidden menus using Mouse Hover.
- Performing Drag and Drop operations.
- Rearranging dashboard widgets.
- Using keyboard shortcuts such as Copy, Paste, Cut, and Select All.
- Navigating forms using the Tab key.
- Scrolling through long web pages.
- Interacting with sliders, canvases, and custom UI controls.

Mouse Actions in Playwright

Playwright provides built-in support for various mouse operations.

Common Mouse Actions

- Single Click
- Double Click
- Right Click (Context Click)
- Mouse Hover
- Drag and Drop
- Mouse Move
- Mouse Wheel (Scroll)
- Click and Hold

Keyboard Actions in Playwright

Playwright also provides built-in support for keyboard interactions.

Common Keyboard Actions

- Type Text
- Press Individual Keys
- Keyboard Shortcuts
- Copy & Paste
- Cut & Paste
- Select All
- Enter
- Tab
- Escape
- Backspace
- Delete
- Shift Key
- Control / Command (Meta) Keys

Real-Time Applications of Mouse Actions

Mouse Actions are widely used in automation testing for the following scenarios:

- Opening Context Menus using Right Click.
- Navigating Hidden Dropdown Menus.
- Displaying Tooltips.
- Dragging Products into Shopping Carts.
- Rearranging Dashboard Widgets.
- Moving Slider Controls.
- Performing Signature Pad Operations.

Real-Time Applications of Keyboard Actions

Keyboard Actions are commonly used for:

- Filling Login Forms.
- Entering Search Text.
- Navigating Forms using the Tab Key.
- Performing Copy & Paste operations.
- Selecting All Text.
- Using Keyboard Shortcuts.

Common Mouse Methods

Method	Purpose
`click()`	Performs a single mouse click
`dblclick()`	Performs a double click
`click({ button: "right" })`	Performs a right click
`hover()`	Moves the mouse over an element
`dragTo()`	Performs drag and drop
`page.mouse.move()`	Moves the mouse to specific coordinates
`page.mouse.down()`	Presses the left mouse button
`page.mouse.up()`	Releases the mouse button
`page.mouse.wheel()`	Scrolls the page vertically or horizontally

Common Keyboard Methods

Method	Purpose
`fill()`	Enters text into an input field
`type()`	Types text character by character
`press()`	Presses a keyboard key on an element
`keyboard.press()`	Presses a key globally on the page
`keyboard.down()`	Holds a key down
`keyboard.up()`	Releases a held key
`inputValue()`	Returns the current value of an input field

Mouse Actions vs Keyboard Actions

Mouse Actions	Keyboard Actions
Interact using the mouse pointer	Interact using keyboard keys
Used for clicking and hovering	Used for typing and shortcuts
Works with mouse methods	Works with keyboard methods
Simulates mouse movements	Simulates keyboard input

Interview Questions

Q1. What are Mouse and Keyboard Actions in Playwright?

Mouse and Keyboard Actions are built-in Playwright features used to simulate real user interactions such as clicking, hovering, dragging, typing, pressing keys, and using keyboard shortcuts.

Q2. Does Playwright use an Actions class like Selenium?

No.

Playwright does not require an Actions class. It provides built-in methods such as click(), dblclick(), hover(), dragTo(), and press().

Q3. Which method is used to perform Mouse Hover?

```
await locator.hover();
```

Q4. Which method is used for Drag and Drop?

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

Q5. Which method performs a Right Click?

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

Q6. Which method performs a Double Click?

```
await locator.dblclick();
```

Q7. Which methods are used for Keyboard Actions?

- fill()
- type()
- press()
- keyboard.press()

- keyboard.down()
- keyboard.up()

Q8. What is the difference between locator.press() and keyboard.press()?

locator.press()

- Presses a key on a specific element.
- The element receives keyboard focus automatically.

keyboard.press()

- Presses a key globally on the active page.
- Useful when no specific element needs to be targeted.

What is an Assertion?

An **Assertion** is a verification point in a test that checks whether the **actual result matches the expected result**.

Assertions help ensure that the application behaves as expected. If an assertion fails, Playwright reports the test as failed.

Example:

```
await expect(page).toHaveTitle("Dashboard");
```

This verifies that the page title is **"Dashboard"**.

Types of Assertions in Playwright

Playwright provides two types of assertions:

1. **Hard Assertion**
2. **Soft Assertion**

1. Hard Assertion

What is a Hard Assertion?

A **Hard Assertion** is the default assertion in Playwright. It uses the expect() method.

If a Hard Assertion fails:

- The current test **stops immediately**.
- Remaining test steps are **not executed**.
- The test is marked as **Failed**.

This behavior prevents the execution of further steps when a critical validation fails.

Example

```
await expect(page.locator("#login")).toBeVisible();

console.log("This line executes only if assertion passes.");
```

If the element is **not visible**, the test stops immediately and the console.log() statement is never executed.

When Should You Use Hard Assertions?

Use Hard Assertions for **critical validations** where continuing the test makes no sense if the validation fails.

Examples:

- Login page verification
- Successful login
- Page navigation
- URL validation
- Page title validation
- Mandatory button visibility
- Critical form fields
- Payment confirmation page

Advantages

- Stops invalid test execution immediately.
- Prevents cascading failures.
- Easier to identify the exact failure point.
- Best for critical validations.

Disadvantages

- Remaining validations are skipped.
- Only the first failure is reported.

Real-Time Example

Login Test

Step 1 → Open Login Page

Step 2 → Verify Username Field

Step 3 → Verify Password Field

Step 4 → Verify Login Button

Step 5 → Click Login

If the **Login Button is missing**, there is no point in clicking it. Therefore, a **Hard Assertion** is appropriate.

2. Soft Assertion

What is a Soft Assertion?

A **Soft Assertion** uses the expect.soft() method.

If a Soft Assertion fails:

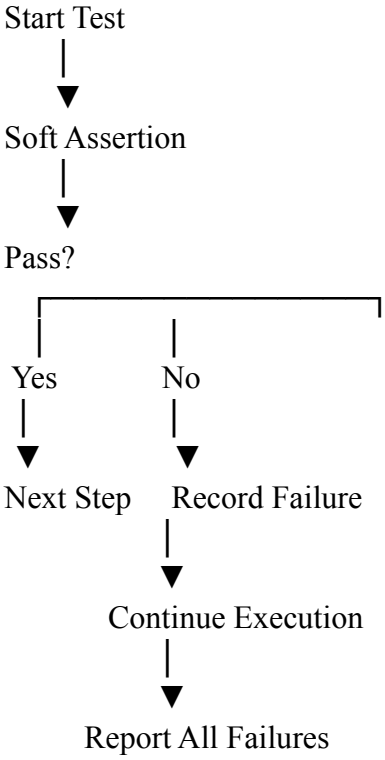
- The failure is recorded.
- Test execution **continues**.
- Remaining test steps are executed.
- All failed soft assertions are reported at the end of the test.

Soft Assertions allow multiple validations within the same test without stopping execution after the first failure.

Syntax

await expect.soft(locator).toBeVisible();

Execution Flow



Example

await expect.soft(page.locator("#username")).toBeVisible();

await expect.soft(page.locator("#password")).toBeVisible();

await expect.soft(page.locator("#login")).toHaveText("Login");

console.log("This line executes even if one assertion fails.");

Even if one validation fails, the remaining validations continue to execute.

When Should You Use Soft Assertions?

Use Soft Assertions when validating **multiple independent UI elements**.

Examples:

- Dashboard widgets
- Table data validation
- Product cards
- User profile details
- Reports
- Form labels
- Multiple menu options
- Multiple cards on a homepage

Advantages

- Continues execution after failures.
- Reports multiple failures in a single test.
- Reduces the need to rerun tests for additional failures.
- Useful for comprehensive UI validation.

Disadvantages

- Test continues even if an important validation fails.
- Not suitable for critical validations like login or payment flow.

Real-Time Example

Imagine a dashboard with:

- Welcome Message
- User Name
- Email Address
- Profile Picture
- Notifications
- Settings Icon

Even if the **Profile Picture** is missing, you still want to verify the remaining dashboard elements. In this case, **Soft Assertions** are appropriate.

Why are Assertions Important?

Without Assertion	With Assertion

Test only performs actions	Test verifies expected behavior
Bugs may go unnoticed	Bugs are detected immediately
No validation	Proper validation
Not suitable for regression	Essential for regression testing

Verification vs Validation

Verification	Validation
Are we building the product right?	Are we building the right product?
Static activity	Dynamic activity
Code Reviews	Executing Test Cases
Performed by Developers	Performed by Testers

Hard Assertion vs Soft Assertion

Feature	Hard Assertion	Soft Assertion
Method	`expect()`	`expect.soft()`
Stops Execution on Failure	✅ Yes	❌ No
Continues Test Execution	❌ No	✅ Yes
Reports Multiple Failures	❌ No	✅ Yes
Best For	Critical Validations	Multiple Independent Validations
Failure Reporting	Immediately	At the End of Test

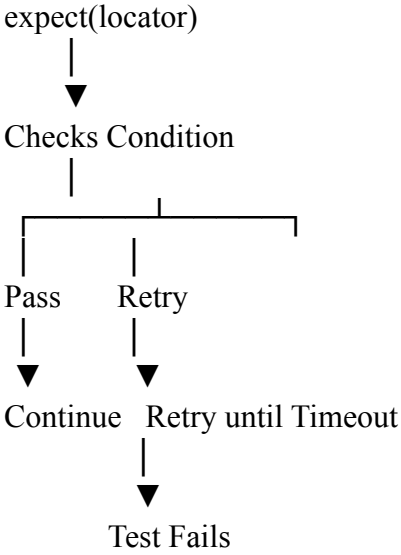
2. Playwright Assertion Library – expect()

What is expect()?

expect() is Playwright's built-in assertion library used to verify application behavior.

import { test, expect } from "@playwright/test";

How expect() Works



Auto Waiting

Locator assertions automatically wait until:

- Element appears
- Text changes
- URL changes
- Title changes

No need to use: await page.waitForTimeout(3000);

before assertions.

Retry Mechanism

Default assertion timeout: 5000 ms

Playwright retries every few milliseconds until:

- Assertion Passes
- Timeout Expires

Changing Assertion Timeout

Per Assertion

```
await expect(locator).toBeVisible({
   timeout:10000
```

```
});
```

Global. -

```
expect:{
   timeout:10000
}
```

3. Generic Assertions

Generic Assertions are used to validate **JavaScript values** such as Numbers, Strings, Arrays, Objects, and Boolean values.

Note: Generic Assertions do **NOT** require await because they do not support Auto-Waiting.

3.1 toEqual()

Purpose: Performs deep comparison of Arrays and Objects.

Real-Time Use:

- API Response Validation
- JSON Validation
- Compare Arrays
- Compare Objects

Syntax:

```
expect([1, 2]).toEqual([1, 2]);
```

3.2 toStrictEqual()

Purpose: Performs deep comparison including object type and structure.

Real-Time Use:

- Object Validation
- API Response Validation
- Class Object Comparison

Syntax:

```
expect(user).toStrictEqual(expectedUser);
```

3.3 toContain()

Purpose: Checks whether a String or Array contains a specific value.

Real-Time Use:

- Search Results
- Product Lists
- Browser Lists
- Text Validation

Syntax (String):

```
expect("Playwright").toContain("Play");
```

Syntax (Array):

```
expect(["Chrome", "Firefox"]).toContain("Chrome");
```

3.4 toContainEqual()

Purpose: Checks whether an Array contains a matching Object.

Real-Time Use:

- API Response Validation
- Employee Records
- Product Lists
- User Objects

Syntax:

```
expect(users).toContainEqual({ id: 2 });
```

3.5 toBeTruthy()

Purpose: Checks whether a value is truthy.

Truthy Values:

- true
- 1
- "Playwright"
- []
- {}

Real-Time Use:

- Boolean Validation
- API Response
- Flag Verification

Syntax:

```
expect(1).toBeTruthy();
```

3.6 toBeFalsy()

Purpose: Checks whether a value is falsy.

Falsy Values:

- false
- 0
- ""
- null
- undefined
- NaN

Real-Time Use:

- Boolean Validation
- Empty Values
- API Validation

Syntax:

```
expect(0).toBeFalsy();
```

Number Assertions

3.7 toBeGreaterThan()

Purpose: Checks whether a number is greater than the expected value.

Real-Time Use:

- Product Count
- Row Count
- Response Time
- Order Amount

Syntax:

```
expect(price).toBeGreaterThan(100);
```

3.8 toBeGreaterThanOrEqual()

Purpose: Checks whether a number is greater than or equal to the expected value.

Real-Time Use:

- Minimum Marks
- Minimum Balance

- Stock Quantity

Syntax:

```
expect(price).toBeGreaterThanOrEqual(100);
```

3.9 toBeLessThan()

Purpose: Checks whether a number is less than the expected value.

Real-Time Use:

- Response Time
- Product Price
- Discount Validation

Syntax:

```
expect(price).toBeLessThan(500);
```

3.10 toBeLessThanOrEqual()

Purpose: Checks whether a number is less than or equal to the expected value.

Real-Time Use:

- Maximum Price
- Maximum Quantity
- Response Time Validation

Syntax:

```
expect(price).toBeLessThanOrEqual(500);
```

3.11 toHaveLength()

Purpose: Checks the length of a String or an Array.

Real-Time Use:

- Product Count
- Search Results
- Username Length
- Password Validation

Syntax (String):

```
expect("Playwright").toHaveLength(10);
```

Syntax (Array):

```
expect([1, 2, 3]).toHaveLength(3);
```

4. Locator Assertions

Locator assertions are used to validate **Web Elements (Locators)**.

Unlike Generic Assertions, Locator Assertions **automatically wait** until the expected condition is satisfied or the timeout expires.

Note: Locator Assertions always require await.

4.1 toBeVisible()

Purpose: Checks whether an element is visible on the webpage.

Real-Time Use:

- Login Button
- Search Box
- Success Message
- Popup

Syntax:

```
await expect(locator).toBeVisible();
```

4.2 toBeHidden()

Purpose: Checks whether an element is hidden.

Real-Time Use:

- Loading Spinner
- Loader
- Closed Popup
- Hidden Menu

Syntax:

```
await expect(locator).toBeHidden();
```

4.3 toBeEnabled()

Purpose: Checks whether an element is enabled and can be interacted with.

Real-Time Use:

- Login Button

- Submit Button
- Save Button
- Checkout Button

Syntax:

```
await expect(locator).toBeEnabled();
```

4.4 toBeDisabled()

Purpose: Checks whether an element is disabled.

Real-Time Use:

- Disabled Submit Button
- Disabled Save Button
- Read-only Forms

Syntax:

```
await expect(locator).toBeDisabled();
```

4.5 toBeEditable()

Purpose: Checks whether an input field is editable.

Real-Time Use:

- Username Field
- Email Field
- Search Box
- Address Field

Syntax:

```
await expect(locator).toBeEditable();
```

4.6 toBeEmpty()

Purpose: Checks whether an element or input field is empty.

Real-Time Use:

- Empty Textbox
- Empty Comment Box
- Empty Search Field

Syntax:

```
await expect(locator).toBeEmpty();
```

4.7 toBeChecked()

Purpose: Checks whether a Checkbox or Radio Button is selected.

Real-Time Use:

- Remember Me
- Terms & Conditions
- Gender Selection

Syntax:

```
await expect(locator).toBeChecked();
```

4.8 toContainText()

Purpose: Validates partial text inside an element.

Real-Time Use:

- Success Message
- Error Message
- Toast Notification
- Alert Message

Syntax:

```
await expect(locator).toContainText("Success");
```

4.9 toHaveText()

Purpose: Validates the exact text of an element.

Real-Time Use:

- Page Heading
- Labels
- Button Text
- Table Headers

Syntax:

```
await expect(locator).toHaveText("Login Successful");
```

4.10 toHaveValue()

Purpose: Validates the value inside an input field.

Real-Time Use:

- Username
- Email
- Search Box
- Mobile Number

Syntax:

```
await expect(locator).toHaveValue("Admin");
```

4.12 toHaveAttribute()

Purpose: Validates an HTML attribute and its value.

Real-Time Use: Validate type, href, placeholder, disabled, etc.

Syntax:

```
await expect(locator).toHaveAttribute("type", "text");
```

4.13 toHaveCSS()

Purpose: Validates the CSS property value of an element.

Real-Time Use: Validate button color, font size, background color, etc.

Syntax:

```
await expect(locator).toHaveCSS("color", "rgb(255, 0, 0)");
```

4.14 toHaveCount()

Purpose: Validates the total number of matching elements.

Real-Time Use: Validate table rows, products, search results, menu items, etc.

Syntax:

```
await expect(locator).toHaveCount(10);
```

5. Page Assertions

5.1 toHaveTitle()

Purpose: Validates the browser page title.

Real-Time Use: Login validation, navigation validation, dashboard verification.

Syntax (Exact Match):

await expect(page).toHaveTitle("Dashboard");

Syntax (Regular Expression):

await expect(page).toHaveTitle(/Dashboard/);

5.2 toHaveURL()

Purpose: Validates the current page URL.

Real-Time Use: Login, logout, redirects, page navigation.

Syntax (Exact Match):

await expect(page).toHaveURL("https://example.com/dashboard");

Syntax (Regular Expression):

await expect(page).toHaveURL(/dashboard/);

6. Hard Assertion

Purpose: Stops the test execution immediately if the assertion fails.

Default Assertion: expect()

Syntax:

await expect(locator).toBeVisible();

Example:

await expect(page.locator("#login")).toBeVisible();

✔ Test Stops

Remaining steps do not execute.

7. Soft Assertions

await expect.soft(locator) .toBeVisible();

If it fails

✔ Failure is recorded

✔ Test continues

✔ All failures shown at end

Hard vs Soft Assertion

Hard	Soft
Stops Test	Continues Test
expect()	expect.soft()
Critical Validation	UI Validation

8. Negation Assertions

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

await expect(locator)
.not.toBeChecked();

await expect(locator)
.not.toHaveText("Error");
```

9. Assertion Timeouts

Timeout	Default
Assertion	5 sec
Navigation	30 sec
Action	Unlimited

10. Locator Assertions vs Generic Assertions

Locator Assertions	Generic Assertions
Auto Wait	No Auto Wait
await Required	No await
Retry	No Retry
Locator/Page	JS Values

12. Comparison Tables

toBe() vs toEqual()

toBe()	**toEqual()**
Strict Equality	Deep Equality
Primitive Values	Objects & Arrays

toContainText() vs toHaveText()

toContainText()	**toHaveText()**
Partial Match	Exact Match

Hard vs Soft Assertion

Hard	**Soft**
Stops Test	Continues Test
expect()	expect.soft()

Locator vs Generic Assertion

Locator	**Generic**
Auto Wait	No Auto Wait
Retry	No Retry
await Required	No await

Web Tables in Playwright

1. What is a Web Table?

A **Web Table**, also called an **HTML Table**, is a structure used on a web page to organize and display data in **rows and columns**.

HTML tables are mainly created using the following tags:

```
<table>
   <tr>
      <th>Book</th>
      <th>Author</th>
      <th>Price</th>
   </tr>

   <tr>
      <td>Learn JS</td>
      <td>Amit</td>
      <td>300</td>
   </tr>
</table>
```

Important HTML Table Tags

Tag	Meaning
`<table>`	Defines the table
`<tr>`	Defines a table row
`<th>`	Defines a table header
`<td>`	Defines table data/cell
`<thead>`	Contains table header rows
`<tbody>`	Contains table body rows
`<tfoot>`	Contains table footer rows

2. Basic Web Table Structure

Consider the following table:

Book	Author	Subject	Price
Learn JS	Amit	JavaScript	300
Learn Selenium	Mukesh	Selenium	500
Learn Playwright	Naveen	Playwright	700

This table contains:

- **4 columns**
- **4 rows including the header**
- **3 data rows**

Understanding Row and Column

```
            Columns
        1        2        3        4
      ---------------------------------------------
Row 1 |   Book     Author     Subject     Price
      ---------------------------------------------
Row 2 | Learn JS    Amit      JavaScript    300
      ---------------------------------------------
Row 3 | Learn      Mukesh     Selenium      500
      ---------------------------------------------
Row 4 | Learn      Naveen     Playwright    700
      ---------------------------------------------
```

3. Why Do We Need Web Table Handling?

In automation testing, we rarely just verify that a table exists.

Usually, we need to interact with or validate the data inside the table.

Common operations include:

- Count total rows
- Count total columns
- Print complete table
- Read a particular row
- Read a particular column
- Find a specific value
- Verify specific data
- Find a row based on a value
- Read data from the same row
- Click a checkbox inside a row
- Click Edit button inside a row
- Click Delete button inside a row

- Validate prices
- Validate status
- Validate multiple rows
- Compare table data with expected data

4. Types of Web Tables

Web tables can commonly be handled as:

1. Static Web Table

The table data remains mostly fixed.

Example:

Book	Author	Price
Learn JS	Amit	300
Selenium	Mukesh	500

The data is already present when the page loads.

2. Dynamic Web Table

The table data can change dynamically.

For example:

- Data comes from an API
- Data changes after refresh
- Rows are added dynamically
- Rows are removed dynamically
- Pagination changes the displayed records
- Sorting changes the row order
- Filtering changes the displayed records

Example:

Name	CPU	Memory
Chrome	20%	40%
Firefox	15%	35%
Edge	10%	30%

The values may change every time the page loads.

5. Locating a Web Table

Suppose the HTML is:

<table id="employeeTable">

We can locate it using:

```
const table = page.locator("#employeeTable");
```

Using XPath:

```
const table = page.locator("//table[@id='employeeTable']");
```

6. Count Rows

To count all rows:

```
const rows = page.locator("#employeeTable tbody tr");
const rowCount = await rows.count();
console.log("Total Rows :", rowCount);
```

Explanation

#employeeTable

→ Finds the table.

tbody

→ Finds the table body.

tr

→ Finds all rows.

count()

→ Returns the number of matching rows.

7. Count Columns

To count columns:

```
const columns = page.locator("#employeeTable thead th");

const columnCount = await columns.count();

console.log("Total Columns :", columnCount);
```

If the table does not use <thead>, you can use:

```
const columns = page.locator("#employeeTable tr:first-child th");
```

8. Read Complete Table

We can use nested loops to read the complete table.

```
const rows = page.locator("#employeeTable tbody tr");

const rowCount = await rows.count();

for (let i = 0; i < rowCount; i++) {

    const cells = rows.nth(i).locator("td");

    const columnCount = await cells.count();

    for (let j = 0; j < columnCount; j++) {

        const text = await cells.nth(j).textContent();

        console.log(text?.trim());
    }
}
```

Logic

```
Table
  ↓
Rows
  ↓
Each Row
  ↓
Cells
  ↓
```

Each Cell
↓
Cell Text

9. Read Table Row by Row

A cleaner approach is to create an array for each row:

```
const rows = page.locator("#employeeTable tbody tr");

const rowCount = await rows.count();

for (let i = 0; i < rowCount; i++) {

   const cells = rows.nth(i).locator("td");

   const columnCount = await cells.count();

   const rowData = [];

   for (let j = 0; j < columnCount; j++) {

      const text = await cells.nth(j).textContent();

      rowData.push(text ? text.trim() : "");
   }

   console.log(rowData.join(" | "));
}
```

Output:

```
Learn JS | Amit | JavaScript | 300
Learn Selenium | Mukesh | Selenium | 500
Learn Playwright | Naveen | Playwright | 700
```

10. Read a Particular Row

Suppose we want the first data row:

```
const firstRow = page.locator("#employeeTable tbody tr").first();

const rowText = await firstRow.innerText();
```

```
console.log(rowText);
```

Using nth():

```
const secondRow = page.locator("#employeeTable tbody tr").nth(1);

console.log(await secondRow.innerText());
```

Important

Playwright nth() starts from **0**.

```
nth(0) → First row
nth(1) → Second row
nth(2) → Third row
```

11. Read a Particular Column

Suppose we want all book names.

```
const books = page.locator("#employeeTable tbody tr td:nth-child(1)");

const count = await books.count();

for (let i = 0; i < count; i++) {

    console.log(await books.nth(i).innerText());
}
```

Output:

```
Learn JS
Learn Selenium
Learn Playwright
```

12. Find Specific Data

Suppose we want to find:

```
Learn Selenium
```

We can loop through the rows:

```
const rows = page.locator("#employeeTable tbody tr");

const rowCount = await rows.count();

for (let i = 0; i < rowCount; i++) {

   const bookName = await rows.nth(i).locator("td").nth(0).innerText();

   if (bookName.trim() === "Learn Selenium") {

      console.log("Book Found :", bookName);

      break;
   }
}
```

13. Find Data and Read Another Column

This is one of the **most important web-table interview concepts**.

Suppose we need:

Find "Learn Selenium" and get its price.

Table:

Book	Author	Subject	Price
Learn JS	Amit	JavaScript	300
Learn Selenium	Mukesh	Selenium	500
Learn Playwright	Naveen	Playwright	700

Code:

```
const rows = page.locator("#employeeTable tbody tr");

const rowCount = await rows.count();

for (let i = 0; i < rowCount; i++) {

   const cells = rows.nth(i).locator("td");

   const bookName = await cells.nth(0).innerText();

   if (bookName.trim() === "Learn Selenium") {
```

```
        const price = await cells.nth(3).innerText();

        console.log("Book :", bookName);
        console.log("Price :", price);

        break;
    }
}
```

Important Concept

We first find the **matching row**.

Then we read another cell from the **same row**.

```
Find Book
   ↓
Find Matching Row
   ↓
Read Price From Same Row
```

This concept is very important for automation interviews.

14. Click Edit/Delete Button in a Specific Row

Suppose a table contains:

Name	Role	Action
Rahul	QA	Edit Delete
Amit	Dev	Edit Delete

Requirement:

Find Rahul and click Delete.

```
const rows = page.locator("#employeeTable tbody tr");

const rowCount = await rows.count();

for (let i = 0; i < rowCount; i++) {

    const row = rows.nth(i);

    const name = await row.locator("td").nth(0).innerText();
```

```
    if (name.trim() === "Rahul") {

        await row.getByRole("button", { name: "Delete" }).click();

        break;
    }
}
```

This is better than finding a generic Delete button because we are clicking Delete **inside Rahul's row**.

15. Validate Table Data

Playwright assertions can be used to validate table data.

```
const table = page.locator("#employeeTable");

await expect(table).toBeVisible();

await expect(table.locator("thead th").nth(0))
    .toHaveText("Book");

await expect(table.locator("tbody tr").first().locator("td").nth(0))
    .toHaveText("Learn JS");
```

16. Get All Text From Table

Playwright provides:

```
allTextContents()
```

Example:

```
const books = page.locator("#employeeTable tbody tr td:nth-child(1)");

const bookNames = await books.allTextContents();

console.log(bookNames);
```

Output:

```
[
    "Learn JS",
```

```
    "Learn Selenium",
    "Learn Playwright"
]
```

17. textContent() vs innerText()

textContent()

Returns the text content from the element.

```
const text = await locator.textContent();
```

innerText()

Returns the visible text.

```
const text = await locator.innerText();
```

Simple Difference

textContent()
→ Gets text content from DOM

innerText()
→ Gets visible rendered text

For table automation, both are commonly useful.

18. Static Table Example

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

test("Static Web Table", async ({ page }) => {

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

    const table = page.locator('table[name="BookTable"]');

    await expect(table).toBeVisible();

    const rows = table.locator("tbody tr");

    const rowCount = await rows.count();
```

```
    console.log("Total Rows :", rowCount);

    for (let i = 0; i < rowCount; i++) {

        const cells = rows.nth(i).locator("td");

        const columnCount = await cells.count();

        const rowData = [];

        for (let j = 0; j < columnCount; j++) {

            const text = await cells.nth(j).textContent();

            rowData.push(text ? text.trim() : "");
        }

        console.log(rowData.join(" | "));
    }
});
```

19. Dynamic Table Example

For dynamic tables, we should **not assume that the number of rows or data is always fixed**.

Instead:

```
const rows = page.locator("#taskTable tbody tr");

const rowCount = await rows.count();

console.log("Total Rows :", rowCount);

for (let i = 0; i < rowCount; i++) {

    const cells = rows.nth(i).locator("td");

    const cellCount = await cells.count();

    const rowData = [];

    for (let j = 0; j < cellCount; j++) {

        const text = await cells.nth(j).innerText();
```

```
        rowData.push(text.trim());
    }

    console.log(rowData.join(" | "));
}
```

The advantage is that the code works based on the **current number of rows and columns**.

20. Common Web Table Operations

```
1. Locate table
      ↓
2. Count rows
      ↓
3. Count columns
      ↓
4. Read table data
      ↓
5. Find required row
      ↓
6. Read required column
      ↓
7. Perform action
      ↓
8. Validate result
```

21. Important Playwright Commands for Web Tables

```
// Locate table
page.locator("table");

// Count rows
await rows.count();

// Count columns
await columns.count();

// Get nth row
rows.nth(0);

// Get nth cell
cells.nth(0);

// Get text
```

```
await locator.textContent();

// Get visible text
await locator.innerText();

// Get all text
await locator.allTextContents();

// Check visibility
await expect(table).toBeVisible();

// Validate text
await expect(locator).toHaveText("Learn JS");

// Validate partial text
await expect(locator).toContainText("JS");

// Click button inside row
await row.getByRole("button", { name: "Delete" }).click();
```

22. Important Interview Questions

Q1. What is a Web Table?

A Web Table is an HTML structure used to organize information into rows and columns.

Q2. How do you count rows in Playwright?

```
const rowCount = await page.locator("table tbody tr").count();
```

Q3. How do you count columns?

```
const columnCount = await page.locator("table thead th").count();
```

Q4. How do you read all rows?

Use a loop over the rows:

```
for (let i = 0; i < rowCount; i++) {
   console.log(await rows.nth(i).innerText());
}
```

Q5. How do you find a specific value in a table?

Loop through the rows and compare the required cell value.

Q6. How do you get another value from the same row?

First identify the matching row, then locate the required cell inside that row.

```
const row = rows.nth(i);

const name = await row.locator("td").nth(0).innerText();

if (name === "Rahul") {
   const price = await row.locator("td").nth(3).innerText();
}
```

Q7. How do you click Delete for a particular user?

Find the user's row first and click the Delete button inside that row.

Q8. What is the difference between static and dynamic tables?

Static table: Data is generally fixed and available when the page loads.

Dynamic table: Data can change based on API responses, user actions, filtering, sorting, pagination, or other runtime conditions.

Q9. What is nth() in Playwright?

nth() returns the element at a specified **zero-based index**.

```
rows.nth(0); // First row
rows.nth(1); // Second row
```

Q10. What is the most important approach for table automation?

Find the row based on unique data first, then perform the required action or validation within that row.

Example:

```
Find "Rahul"
    ↓
Find Rahul's row
    ↓
Find Delete button inside Rahul's row
    ↓
Click Delete
    ↓
Validate result
```

This approach is much more reliable than using a fixed row number such as tr[3], especially for dynamic tables.

```
// ====================================================
// WINDOW HANDLING - IMPORTANT POINTS
// ====================================================

/*
1. Every Window/Tab in Playwright is represented as a Page Object.

2. Parent Window  -> page
   Child Window   -> popup

3. Playwright does NOT use switchTo().window() like Selenium.

4. page.waitForEvent("popup") waits for a popup opened by the
current Page.
   Return Type -> Promise<Page>

5. Promise.all() is used to wait for the popup event and perform
   the action that opens the popup together.

   Promise.all() allows us to handle multiple asynchronous
operations together and wait for all of them to complete. In
Playwright window handling, we use it to start listening for the
   popup and perform the action that opens the popup at the same
time.

6. popup.waitForLoadState() waits until the child page reaches
   the required loading state.

7. popup.close() closes only the child window.

8. page.context().pages() returns all open Page objects/windows
   inside the BrowserContext.

9. page.context().pages().length returns the total number of
   open windows/pages.
```

```
10. bringToFront() brings a particular Page to the front.

11. No switchTo().window() is required in Playwright.
    We directly use the Page Object of the required window.

12. Avoid hard-coded waitForTimeout().
    Prefer Playwright events, auto-waiting and assertions.
*/
```

1. Introduction to Window Handling

What is Window Handling?

Window Handling is the process of handling multiple browser windows, tabs, or popups that open during automation execution.

Whenever an application opens a new tab or window, the automation script must know:

- Which window was opened.
- How to interact with it.
- How to return to the parent window.

Why Do We Need Window Handling?

Many modern applications open secondary windows during execution.

Examples:

- Google OAuth Login
- Facebook Login
- LinkedIn Login
- Razorpay Payment Gateway
- Stripe Checkout
- PayPal Payment
- OTP Verification Window
- PDF Reports
- Terms & Conditions
- Help Pages

Without proper window handling, Playwright continues working on the parent page while the actual operation happens inside the child window, causing the test to fail.

Window vs Tab vs Popup

Type	Description	Example
Window	Separate browser window	`window.open()`
Tab	New browser tab	`<a target="_blank">`
Popup	Any child page opened from parent	OAuth Login

Important

In Playwright,

Window = Tab = Popup

Internally all are represented by a **Page Object**.

Real-Time Examples

- Amazon Product Details
- Google Login
- LinkedIn Company Page
- Facebook Login
- Bank OTP Window
- Payment Gateway
- Invoice PDF

Why is Window Handling Important?

- Used in almost every enterprise application.
- Required for Payment Gateways.
- Required for OAuth Login.
- Required for PDF Validation.
- Frequently asked in Playwright Interviews.

2. Window Handling in Playwright

Does Playwright Use Window Handles?

No.

Playwright does **NOT** use Window Handles like Selenium.

Selenium uses:

```
driver.getWindowHandles();
driver.switchTo().window(handle);
```

Playwright uses an event-driven approach.

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

Playwright directly returns the **Page Object**.

Selenium vs Playwright

Selenium	Playwright
getWindowHandles()	waitForEvent("popup")
Window Handle	Page Object
switchTo() required	No switching
Manual handling	Automatic
Polling	Event Driven

Why is Playwright Better?

- No Window Handles
- No switchTo()
- Direct Page Object
- Cleaner Code
- Less Boilerplate
- Faster Execution
- Prevents Race Conditions

3. Core Concepts

Browser

The browser engine (Chromium, Firefox, WebKit).

Example

Browser

BrowserContext

A BrowserContext is an isolated browser session.

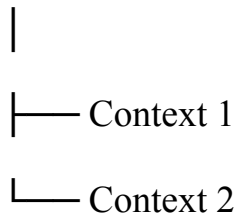
Think of it as an Incognito Window.

Each BrowserContext has:

- Separate Cookies
- Separate Local Storage
- Separate Session Storage
- Separate Login Session

Example

Browser



Page

Every browser tab or window is represented by a **Page Object**.

Page
↓
Parent Tab
↓
Child Tab
↓
Popup

Hierarchy

Browser

BrowserContext

Why BrowserContext Matters?

- Multiple users in one test.
- Independent sessions.
- Better parallel execution.
- Popups belong to the same BrowserContext.

4. Popup Events

What is Popup Event?

Whenever a page opens another page,

Playwright fires a **popup** event.

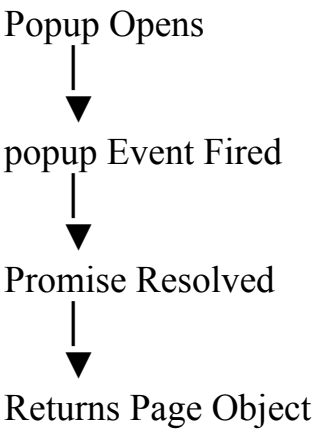
Syntax

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

When is popup Event Fired?

- window.open()
- target="_blank"
- JavaScript popup
- Payment Gateway
- OAuth Login
- Social Media Links

Internal Working



Why do we use waitForEvent("popup")?

It tells Playwright:

"Whenever a popup opens from this page, notify me."

5. Promise.all() (Detailed)

What is Promise.all()?

Promise.all() executes multiple asynchronous operations simultaneously.

It waits until every Promise completes successfully.

Why is it Required?

Window Handling consists of two asynchronous operations.

Operation 1

Register Popup Listener

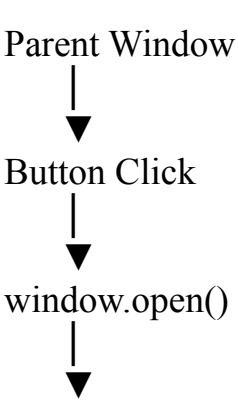
```
page.waitForEvent("popup")
```

Operation 2

Click the Button

```
page.click("#open");
```

Internal Flow



Why Must Both Start Together?

If click executes before the listener,

Playwright misses the popup event.

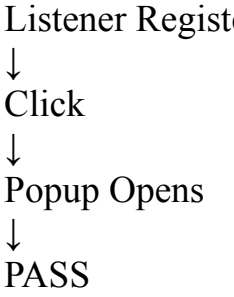
Result:

Timeout.

Race Condition

A Race Condition occurs when two asynchronous operations compete, and the result depends on which finishes first.

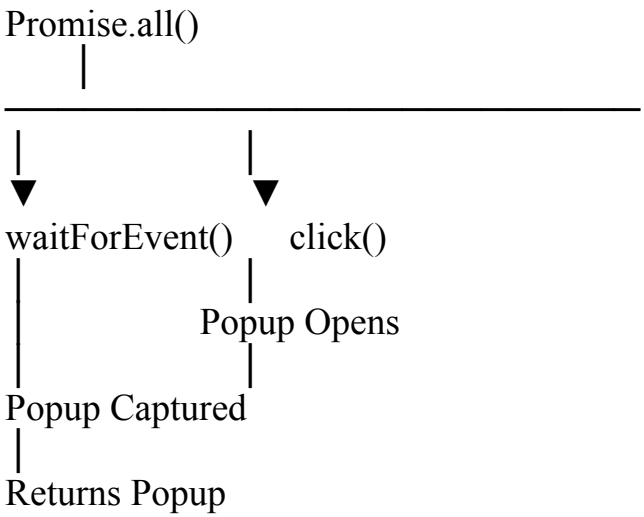
Listener Wins



Click Wins

Click
↓
Popup Opens
↓
Listener Starts
↓
FAIL

How Promise.all() Fixes It



Everything starts together.

No Race Condition.

What does page.waitForEvent("popup") return?

It returns:

Promise<Page>
After popup opens:

Page

Why const [popup]?

Promise.all() always returns an array.

Example

const [popup] = await Promise.all([...]);

Equivalent to

const arr = await Promise.all(...);

const popup = arr[0];

This is called **Array Destructuring**.

What is popup?

popup is **NOT** a Window Handle.

popup is a **Page Object**.

Parent
↓
page
Child
↓
popup

Why popup.waitForLoadState()?

Opening a popup does not mean it has completely loaded.

await popup.waitForLoadState();

Waits until:

- HTML Loaded
- DOM Ready
- CSS Loaded
- JavaScript Loaded

Default

await popup.waitForLoadState("load");

Get Popup Details

Title

await popup.title();

URL

popup.url();

6. Window Handling Methods

page.waitForEvent("popup")

Waits for a popup from a specific page.

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

context.waitForEvent("page")

Waits for any new page inside BrowserContext.

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

context.newPage()

Creates a new blank page.

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

page.bringToFront()

Brings a page to the foreground.

```
await popup.bringToFront();
```

page.close()

Closes the page.

```
await popup.close();
```

page.title()

Returns page title.
await popup.title();

page.url()

Returns page URL.
popup.url();

page.waitForLoadState()

Waits until page finishes loading.
await popup.waitForLoadState();

7. Working with Multiple Windows

Handle Two Windows

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

Handle Multiple Windows

```
const pages = context.pages();
console.log(pages.length);
```

Switch Window Focus

```
await popup.bringToFront();
await page.bringToFront();
```

Close One Window

```
await popup.close();
```

Close All Windows

```
for (const p of context.pages()) {
   await p.close();
}
```

8. Comparison Tables

Selenium vs Playwright

Selenium	Playwright
Window Handle	Page Object
switchTo()	Not Required
Polling	Event Driven
Manual	Automatic

page.waitForEvent() vs context.waitForEvent()

page.waitForEvent()	context.waitForEvent()
Specific Parent	Entire Context
Popup Only	Any New Page

Sequential vs Promise.all()

Sequential	Promise.all()
Race Condition	Safe
May Timeout	Reliable
Not Recommended	Recommended

9. Interview Questions

Q1. What is Window Handling?

Handling multiple browser windows, tabs, or popups using Playwright's Page objects.

Q2. Does Playwright use Window Handles?

No.

It uses Page Objects.

Q3. Which event handles popups?

- page.waitForEvent("popup")
- context.waitForEvent("page")

Q4. Why Promise.all()?

To start the popup listener and click simultaneously, preventing Race Conditions.

Q5. What is Race Condition?

When two asynchronous operations compete, and the result depends on which finishes first.

Q6. What does page.waitForEvent("popup") return?

Promise<Page>

Q7. What is BrowserContext?

An isolated browser session similar to an Incognito Window.

Q8. How do you switch windows in Playwright?

No explicit switching is required.

Simply use the returned **Page Object** (popup).

Q9. How do you close a popup?

await popup.close();

Q10. How do you get all open windows?

```
const pages = context.pages();
console.log(pages.length);
```

Playwright: Browser, BrowserContext, Page & newContext()

Understanding **Browser, BrowserContext, Page, newContext() and newPage()** is important for Playwright because these objects define how Playwright manages the browser, sessions, tabs, and windows.
The basic hierarchy is:

Browser
↓
BrowserContext
↓
Page
↓
Tab / Window

1. Browser

What is Browser?
Browser represents the actual browser instance launched by Playwright.
Playwright supports browsers such as:

- Chromium
- Firefox
- WebKit

When Playwright launches Chromium, Firefox, or WebKit, the resulting browser instance is represented by the Browser object.

```
const browser = ...
```

The Browser is the **top-level object** in this hierarchy.
A single Browser can contain multiple BrowserContexts.

Browser
│
├── BrowserContext 1
│
├── BrowserContext 2
│
└── BrowserContext 3

Why do we need Browser?
The Browser is mainly used when we need to control browser-level operations or create multiple independent BrowserContexts.
For example:

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

Here, the Browser is being used to create a new BrowserContext.

2. BrowserContext

What is BrowserContext?
A **BrowserContext represents an isolated browser session**.
It is similar to having a separate browser session where the user's session-related information is maintained independently.
A BrowserContext can have its own:

- Cookies
- Local Storage
- Session Storage
- Authentication state
- Cache
- Pages

For example:

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

This creates a new isolated BrowserContext.
Important concept
An isolated context means that information belonging to one context does not automatically get shared with another context.
For example:

```
Browser
│
├── Admin Context
│     └── Admin session
│
└── Customer Context
      └── Customer session
```

The Admin Context and Customer Context are separate sessions.
If the Admin logs into an application, the Customer Context does not automatically use the Admin's cookies or authentication session.

3. Why do we need BrowserContext?
BrowserContext is particularly useful when we want to test **multiple users or multiple independent sessions**.
For example, imagine an application with:

Admin User
Customer User

We can create two contexts:

```
const adminContext = await browser.newContext();
const customerContext = await browser.newContext();
```

Now we have:

Browser
│
├── Admin Context
│ └── Admin session
│
└── Customer Context
└── Customer session

This allows both users to be tested independently.
Real-world example
Suppose:

- Admin logs into the application.
- Customer also logs into the application.

If both users use the same session, their cookies and authentication information could interfere with each other.
Using separate BrowserContexts keeps their sessions isolated.

4. What is browser.newContext()?

newContext() is a **method of the Browser object**.
Its purpose is to create a new BrowserContext.
Syntax

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

Flow:

browser
↓
newContext()
↓
BrowserContext

Important point
browser.newContext() **does not create another browser application**.
For example:

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

This gives:

One Browser
│
├── Context 1

└── Context 2

There is still only **one Browser instance**, but there are two isolated sessions.

5. Multiple BrowserContexts

We can create multiple BrowserContexts from the same Browser.

```
const context1 = await browser.newContext();
const context2 = await browser.newContext();
const context3 = await browser.newContext();
```

Structure:

```
Browser
│
├── Context 1
├── Context 2
└── Context 3
```

Each context can have different:

- Cookies
- Storage
- Authentication
- Pages
- Session information

This makes BrowserContext very useful for **multi-user testing**.

6. Page

What is Page?
A **Page represents a browser tab or window**.
For example:

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

creates a new Page inside the BrowserContext.
The Page is the object that we normally use for UI automation.
For example:

```
await page.goto("https://example.com");
await page.locator("#username").fill("admin");
await page.getByRole("button", { name: "Login" }).click();
```

These operations are performed on a Page.

7. context.newPage()

newPage() is a method of BrowserContext.
It creates a new Page inside that context.
Syntax

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

Flow:

```
Browser
  ↓
BrowserContext
  ↓
context.newPage()
  ↓
Page
```

A BrowserContext can contain multiple Pages.
For example:

```
const page1 = await context.newPage();
const page2 = await context.newPage();
const page3 = await context.newPage();
```

Structure:

```
Browser
 │
 └── BrowserContext
      │
      ├── Page 1
      ├── Page 2
      └── Page 3
```

Each Page can represent a different browser tab or window.

8. Multiple Pages in One Context

Suppose you create:

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

const page1 = await context.newPage();
const page2 = await context.newPage();
const page3 = await context.newPage();
```

You now have:

Browser
│
└── BrowserContext
    │
    ├── Page 1 → Google
    ├── Page 2 → Facebook
    └── Page 3 → Amazon

For example:

```
await page1.goto("https://www.google.com");
await page2.goto("https://www.facebook.com");
await page3.goto("https://www.amazon.com");
```

All three Pages belong to the same BrowserContext.

9. Page and Window Handling

This concept is very important for **Window Handling**.
In Playwright, a browser tab or window is represented by a **Page object**.
For example:

Browser
│
└── BrowserContext
    │
    ├── Page → Parent Window
    ├── Page → Child Window
    ├── Page → Facebook
    └── Page → YouTube

If a Parent Window opens a Child Window, the Child Window becomes another Page.
Example:

```
const [popup] = await Promise.all([
   page.waitForEvent("popup"),
   page.locator("button").click()
]);
```

Here:

popup

is a **Page object representing the Child Window**.
Therefore, we can directly perform:

```
await popup.title();
```

```
popup.url();
popup.locator("selector");
await popup.close();
```

There is no need for Selenium-style window switching.

10. context.pages()

context.pages() returns all currently open Page objects belonging to that BrowserContext.
Syntax

```
const pages = context.pages();
```

It returns an array of Page objects.
For example, if the context has:

Parent
LinkedIn
Facebook
YouTube

then:

```
const pages = context.pages();
```

contains four Page objects.
To get the number of Pages:

```
console.log(context.pages().length);
```

Output:
4

11. Difference Between newContext() and newPage()

This is one of the most important differences to remember.
browser.newContext()
Creates a **new isolated browser session**.

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

Think:

New User Session

context.newPage()
Creates a **new tab/window inside that session**.

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

Think:
New Tab
So:

```
browser.newContext()
      ↓
New isolated session

context.newPage()
      ↓
New tab/window
```

12. Browser vs BrowserContext vs Page

Object	Meaning	Example
`browser`	Actual browser instance	Chromium
`BrowserContext`	Isolated browser session	Admin session
`Page`	Browser tab/window	Google tab
`browser.newContext()`	Creates isolated session	New user session
`context.newPage()`	Creates tab/window	New tab
`context.pages()`	Gets all open Pages	Array of Page objects

13. Complete Example

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

test("Browser Context Page Example", async ({ browser }) => {

   // Browser
   console.log("Browser Started");

   // Create BrowserContext
   const context = await browser.newContext();

   // Create Page
   const page = await context.newPage();

   // Open application
   await page.goto("https://www.google.com");
```

```
    console.log("Page Title :", await page.title());
    console.log("Page URL :", page.url());

    // Get all Pages
    console.log("Total Pages :", context.pages().length);

    // Close BrowserContext
    await context.close();
});
```

Flow:

Browser
↓
browser.newContext()
↓
BrowserContext
↓
context.newPage()
↓
Page
↓
Browser Tab

14. Multiple Users Example

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

test("Multiple Users Using BrowserContext", async ({ browser }) => {

    // Create Admin Session
    const adminContext = await browser.newContext();

    // Create Customer Session
    const customerContext = await browser.newContext();

    // Create Admin Page
    const adminPage = await adminContext.newPage();

    // Create Customer Page
    const customerPage = await customerContext.newPage();

    await adminPage.goto("https://example.com");
    await customerPage.goto("https://example.com");

    console.log("Admin URL :", adminPage.url());
    console.log("Customer URL :", customerPage.url());
```

```
    await adminContext.close();
    await customerContext.close();
});
```

Structure:

```
Browser
 │
 ├── Admin Context
 │      └── Admin Page
 │
 └── Customer Context
        └── Customer Page
```

The important point is that **Admin and Customer have independent browser sessions**.

Final trick

Browser
→ Actual browser instance

BrowserContext
→ Isolated browser session

browser.newContext()
→ Creates a new isolated session

Page
→ Browser tab/window

context.newPage()
→ Creates a new tab/window

context.pages()
→ Gets all open tabs/windows

Interview answer
Browser represents the actual browser instance. BrowserContext represents an isolated browser session with its own cookies, storage, and authentication state. Page represents an individual browser tab or window. browser.newContext() creates a new isolated session, while context.newPage() creates a new tab or window inside that session.

Playwright File Upload and Download

This is an important Playwright interview topic. You can explain it in two parts:

1. **File Upload**
2. **File Download**

1. File Upload

What is File Upload?

File upload means selecting a file from the system and uploading it through an application's file input.

Examples:

- Upload profile photo
- Upload resume
- Upload PDF
- Upload Aadhaar/PAN document
- Upload Excel/CSV file

Playwright provides the **setInputFiles()** method for handling file uploads.

Basic Syntax

```
await page.locator("input[type='file']").setInputFiles(filePath);
```

Here:

- locator() identifies the file input.
- setInputFiles() selects the file.
- filePath is the path of the file on the system.

File Upload Example

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

test("File Upload", async ({ page }) => {

   await page.goto("https://the-internet.herokuapp.com/upload");

   const filePath = path.join(__dirname, "../TestData/AdityaDeolalikar.pdf");

   await page.locator("#file-upload").setInputFiles(filePath);

   await page.locator("#file-submit").click();

   await expect(page.locator("h3")).toHaveText("File Uploaded!");

});
```

Execution Flow

Open Application

↓
Locate File Input
↓
setInputFiles(filePath)
↓
File Selected
↓
Click Upload
↓
Validate Upload

2. What if File Chooser is used?
Sometimes the application doesn't allow us to directly use setInputFiles() in the way we expect, and clicking a button opens a file chooser.
Playwright provides the **filechooser event**.
Syntax

```
const [fileChooser] = await Promise.all([
   page.waitForEvent("filechooser"),
   page.locator("button").click()
]);

await fileChooser.setFiles(filePath);
```

Flow

Click Upload Button
↓
File Chooser Opens
↓
Capture filechooser Event
↓
setFiles()
↓
File Selected

For most normal <input type="file"> elements, however, **setInputFiles() is simpler and preferred**.

3. File Download
What is File Download?
File download means downloading a file from an application to the local system.
Examples:

- Download PDF
- Download invoice
- Download report
- Download Excel file
- Download CSV file

Playwright provides the **download event** to handle downloads.

4. Basic Download Syntax

```
const [download] = await Promise.all([
   page.waitForEvent("download"),
   page.locator("Download Button").click()
]);
```

Why Promise.all()?
Because we need to:

Wait for Download
+
Click Download

at the same time.
This prevents a race condition where the download starts before Playwright begins listening for it.

Playwright Grouping (test.describe)

1. What is Grouping?

Grouping means putting related test cases together into a single block called a **suite**. Playwright provides the test.describe() method to group related test cases.

Grouping helps organize tests based on modules, features, or scenarios, making the automation framework easier to understand and maintain.

Why do we use Grouping?

Benefit	Explanation
Readable Reports	Tests appear under a common suite name in the HTML report.
Shared Setup and Teardown	Hooks defined inside a `describe()` block run only for the tests in that suite.
Bulk Skip or Focus	Skip or execute an entire suite using a single modifier.
Serial or Parallel Execution	Execute all tests in a suite sequentially or simultaneously.
Better Organization	Group tests by module such as Login, Cart, Payment, Dashboard, etc.

Real-Time Analogy

Think of **test.describe()** as a **folder** and the individual **test()** methods as **files** inside that folder.

Example:

Login Module
  Login Test
  Invalid Login Test
  Forgot Password Test

Without grouping, all test cases appear separately, making the project difficult to maintain.

2. test.describe() – The Grouping Block

Syntax

```
test.describe("Suite Name", () => {

  test("Test Case 1", async ({ page }) => {

  });
```

```
    test("Test Case 2", async ({ page }) => {

    });

});
```

Example

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

test.describe("Login Page Tests", () => {

    test("Should show error for invalid credentials", async ({ page }) => {

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

        await page.fill("#user", "wrong");
        await page.fill("#pass", "wrong");

        await page.click("button[type='submit']");

        await expect(page.locator(".error"))
            .toHaveText("Invalid credentials");

    });

    test("Should login successfully with valid credentials", async ({ page }) => {

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

        await page.fill("#user", "admin");
        await page.fill("#pass", "admin123");

        await page.click("button[type='submit']");

        await expect(page).toHaveURL(/dashboard/);

    });

});
```