🎭 What is Playwright

Playwright is a tool to automate browsers — like Chrome, Firefox, or Edge — for testing web applications.

Think of it as a robot tester that opens your website, clicks buttons, fills forms, and checks if everything works properly.

🧩 Key Features of Playwright (with simple examples & analogies)

1. Cross-Browser Testing

✅ Meaning: Run your tests on different browsers — Chromium, Firefox, WebKit (Safari).

🧠 Analogy: Like checking if your movie looks good on a TV, laptop, and phone — you want your website to work everywhere.

🌍 Example: You test your e-commerce checkout on Chrome and Safari automatically — to ensure “Add to Cart” works on both.

2. Cross-Platform Support

✅ Meaning: Works on Windows, macOS, and Linux.

🧠 Analogy: Like a mobile charger that works in any socket across countries.

💻 Example: Your team uses Windows and macOS — both can run the same Playwright tests easily.

3. Multiple Language Support

✅ Meaning: You can write Playwright tests in JavaScript, TypeScript, Python, Java, or .NET.

🧠 Analogy: It’s like being multilingual — Playwright understands many “languages”.

💡 Example: A Python developer and a JavaScript developer can both test the same web app using Playwright.

4. Auto-Wait (Smart Waiting)

✅ Meaning: Playwright waits automatically for elements to load before acting.

🧠 Analogy: Like waiting for the traffic signal to turn green before driving — Playwright doesn’t rush.

🕒 Example: When you click “Login”, Playwright waits until the next page actually appears before checking content.

5. Headless & Headed Modes

✅ Meaning: Can run with or without showing the browser UI.

🧠 Analogy: Headless = robot working silently behind the curtain; Headed = you watch the robot do it live.

🎬 Example: During daily automated runs, you use headless mode (faster); for debugging, you use headed mode (visible).

6. Built-in Test Runner (Playwright Test)

✅ Meaning: It has its own framework to organize, run, and report tests.

🧠 Analogy: Like a sports tournament manager — it schedules matches (tests), records scores (results), and gives a report.

📋 Example: You group your login, payment, and search tests — Playwright Test runs them all and shows a summary.

7. Parallel Execution

✅ Meaning: Run multiple tests at the same time to save time.

🧠 Analogy: Like having multiple cash counters open at a supermarket — all customers (tests) get served faster.

⚡ Example: Instead of testing 10 pages one by one, Playwright tests them all in parallel.

8. Fixtures & Test Hooks

✅ Meaning: Common setup or cleanup logic that runs automatically before/after tests.

🧠 Analogy: Like preparing the exam hall before students enter (setup) and cleaning it after (teardown).

🧪 Example: Before every test, open browser → go to homepage. After test, close browser.

9. Network Interception / Mocking

✅ Meaning: You can fake or block network calls to test offline or specific scenarios.

🧠 Analogy: Like pretending there’s a slow internet connection to see how your app behaves.

📱 Example: Test how your weather app behaves if the API returns an error or slow response.

10. Tracing and Debugging Tools

✅ Meaning: You can record videos, screenshots, and logs of test runs.

🧠 Analogy: Like having a CCTV recording to find out why something failed.

🎥 Example: When a test fails, you replay the trace to see what went wrong — maybe a button didn’t load.

11. Emulation (Devices, Viewports, Geolocation)

✅ Meaning: Simulate different devices and locations.

🧠 Analogy: Like wearing VR glasses that make you feel like you’re in another country using a different phone.

📍 Example: Test how your website looks on an iPhone 14 or how it behaves when user is in London vs. India.

12. Automatic Code Generation (Codegen)

✅ Meaning: It can record your actions and automatically generate test scripts.

🧠 Analogy: Like a stenographer writing notes as you talk — it writes code as you click.

🎮 Example: You open Playwright Codegen, click through your web app, and it gives you ready test code.

13. Screenshots and Video Recording

✅ Meaning: Capture what’s happening during test execution.

🧠 Analogy: Like taking photos or recording videos of a process to show proof it worked.

📸 Example: When running nightly tests, it automatically saves a screenshot of failed steps.

14. Continuous Integration (CI) Friendly

✅ Meaning: Works easily with CI/CD tools like GitHub Actions, Jenkins, or Azure DevOps.

🧠 Analogy: Like a chef who fits perfectly into any kitchen setup.

🔁 Example: Every time you push new code to GitHub, Playwright tests run automatically.

**What is a Fixture in Playwright?**

Fixture = Pre-arranged setup or helper that provides a ready-to-use object for your test.

✅ Simple Meaning:

A fixture gives you something “ready-made” to use in your test (like a browser, a page, or test data).

🧠 Analogy:

Think of fixtures like a hotel staff preparing your room before you check in.

You don’t have to set up the bed, towels, or lights — they’re already ready for you.

You just use the room.

💡 In testing:

Instead of opening the browser, creating pages, or logging in manually in every test — Playwright does it for you using fixtures.

# 🔹 Browser Fixture

What it is:

\* The `browser` fixture is Playwright’s way of giving you a ready-to-use browser engine (Chromium, Firefox, or WebKit).

\* From this browser, you can open contexts (independent sessions) and pages (tabs).

🧠 Analogy: The Hotel

| Playwright Object | Analogy |

| -- | -- |

| Browser | The entire hotel building 🏨 |

| Context | A private room in the hotel 🛏️ |

| Page | The bed in that room 🛌 (ready to use) |

\* Browser = one hotel building

\* Contexts = separate rooms (isolated, private)

\* Pages = beds inside the rooms (ready to use immediately)

✅ You can have multiple rooms (contexts) in one hotel (browser), and multiple beds (pages) in each room.

When to Use `browser`:

\* When you want multiple independent users in the same test.

# 🔹 Explaining the Code Using the Hotel Analogy

```javascript

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

test('Browser fixture example with multiple contexts', async ({ browser }) => {

// Create first context (User 1)

const contextUser1 = await browser.newContext();

const pageUser1 = await contextUser1.newPage();

await pageUser1.goto('https://example.com');

console.log('User 1 Page Title:', await pageUser1.title());

// Create second context (User 2)

const contextUser2 = await browser.newContext();

const pageUser2 = await contextUser2.newPage();

await pageUser2.goto('https://example.com');

console.log('User 2 Page Title:', await pageUser2.title());

// Close contexts (cleanup)

await contextUser1.close();

await contextUser2.close();

});

```

🔹 Step by Step

1. `browser` → The hotel building

\* Playwright gives you a ready hotel (no need to build it yourself).

2. `browser.newContext()` → Open a private room in the hotel

\* `contextUser1` = Room for User 1

\* `contextUser2` = Room for User 2

\* These rooms are isolated — no shared cookies, localStorage, or cache

3. `context.newPage()` → Open a bed (tab) in that room

\* `pageUser1` = User 1’s bed (ready to use)

\* `pageUser2` = User 2’s bed

4. `page.goto()` → The user lays on the bed and goes somewhere (visits the website)

5. `console.log(page.title())` → Check what’s on the bed (page content)

6. `context.close()` → Clean up the room after the guest leaves

\* All cookies, storage, and session data are removed

🔹 Key Takeaways

\* One browser instance → many contexts (rooms) → many pages (beds)

\* Using `browser` allows manual control over sessions

\* You can simulate multiple independent users in the same test

🖥️ Output of This Test

```

User 1 Page Title: Example Domain

User 2 Page Title: Example Domain

```

\* Both users see the same website

\* Their sessions are completely isolated (cookies, storage don’t mix)

What is a `page` fixture?

In Playwright:

\* A browser is like a building.

\* A browser context is like a floor in the building, isolated from other floors.

\* A page is like a room on that floor where you can do things (open apps, type, click, etc.).

The `page` fixture gives you a fresh room to work in for each test. Playwright automatically creates a new page in a browser context when you use it, so you don’t have to manually create it every time.

2️⃣ Real-time analogy

Imagine you are testing different users logging into a website:

\* Browser = the building

\* Browser context = separate floor for each user (so they don’t share cookies)

\* Page = the room where a user sits and interacts with the website

So, if you have 3 tests:

\* Each test gets a new page, ensuring one user doesn’t mess with another user’s session.

3️⃣ Example Code Using `page` Fixture

```javascript

// tests/example.spec.js

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

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

// 'page' is automatically created for you

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

// Type in search box

await page.fill('input[name="q"]', 'Playwright testing');

// Press Enter

await page.press('input[name="q"]', 'Enter');

// Wait for search results

await page.waitForSelector('h3');

// Assert first result contains 'Playwright'

const firstResult = await page.locator('h3').first().textContent();

expect(firstResult).toContain('Playwright');

});

```

✅ Key points:

\* `page` is automatically created by Playwright.

\* Each test gets a fresh page.

\* You don’t need `browser.newPage()` manually unless you want multiple pages in a single test.

What is `browserName` fixture?

\* In Playwright, `browserName` is a built-in fixture that tells your test which browser is currently running.

\* It does not launch a browser itself — it just gives you the name of the browser that Playwright already launched for this test.

Analogy

\* Imagine you’re in a testing building (browser fixture analogy):

\* Building = browser (Chrome, Firefox, or WebKit)

\* Floor = context

\* Room = page

\* Now, `browserName` is like a label on the building door that tells you:

> “This is Chrome building” or “This is Firefox building.”

\* You use it when you want to run some logic differently depending on the browser.

2️⃣ Real-time use case

Sometimes, a website behaves slightly differently in Chrome vs Safari vs Firefox.

\* Maybe a button selector changes.

\* Maybe a hover effect works differently.

Using `browserName`, you can write conditional logic in tests.

3️⃣ Example Code

```javascript

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

test('Browser-specific test', async ({ page, browserName }) => {

console.log('Running on browser:', browserName);

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

if (browserName === 'chromium') {

console.log('Special handling for Chrome');

await page.click('#chrome-button');

} else if (browserName === 'firefox') {

console.log('Special handling for Firefox');

await page.click('#firefox-button');

} else if (browserName === 'webkit') {

console.log('Special handling for Safari/WebKit');

await page.click('#safari-button');

}

const title = await page.title();

expect(title).toBe('Example Domain');

});

```

✅ Key points:

\* `browserName` is read-only — you don’t set it in the test.

\* Playwright automatically provides it based on which browser the test runner is using.

\* Useful for cross-browser conditional testing.

4️⃣ Running the test in multiple browsers

You can run the same test in multiple browsers using Playwright config:

```javascript

// playwright.config.js

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

export default defineConfig({

projects: [

{ name: 'chromium', use: { browserName: 'chromium' } },

{ name: 'firefox', use: { browserName: 'firefox' } },

{ name: 'webkit', use: { browserName: 'webkit' } },

],

});

```

\* Playwright will run the same test 3 times, each time providing a different `browserName` fixture.

\* Your test can now adapt to each browser dynamically.

What is baseURL Fixture

The default website address that all your tests can use.

Defined once in your config.

🧠 Analogy:

Like setting your home location in a GPS app.

Then you just enter “go to park” instead of typing the full address.

🧰 Why Use It:

To simplify test URLs.

So that if the site URL changes, you only change it once (in config).

🧪 Example:

playwright.config.js

export default {

use: {

baseURL: 'https://example.com',

},

test file:

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

test('Base URL fixture example', async ({ page, baseURL }) => {

await page.goto(`${baseURL}/about`);

console.log('Current URL:', await page.url());

});

🖥️ Output:

Current URL: https://example.com/about

🧩 Behind the Scenes:

The baseURL is automatically injected into each test.

You can use relative paths to make tests cleaner

Got it! Let’s break down `request` fixture in Playwright with analogy, real-time use, and code.

What is the `request` fixture?

\* `request` is a built-in Playwright fixture for API testing.

\* It lets you send HTTP requests (GET, POST, PUT, DELETE, etc.) directly without opening a browser.

\* Perfect for backend testing, checking APIs, or setting up test data.

Analogy

Think of it like sending a letter to the server instead of visiting the website in person:

\* Browser → physically visiting the site

\* Request → sending a direct message to the server asking for data

\* Response → the server’s reply to your request

So, you can test the backend without even opening a page.

2️⃣ Example Use Case

\* You want to check if a user API works correctly.

\* You can send a request, check the response status, and validate data.

3️⃣ Example Code

```javascript

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

test('API test using request fixture', async ({ request }) => {

// Send GET request

const response = await request.get('https://jsonplaceholder.typicode.com/posts/1');

// Check status code

expect(response.status()).toBe(200);

// Get JSON body

const data = await response.json();

console.log('Post title:', data.title);

// Assert specific data

expect(data.id).toBe(1);

});

```

🔹 Tags in Playwright — Theoretical Explanation

1. What Are Tags?

\* Tags are labels or annotations you attach to tests to categorize or group them.

\* They help you run only specific types of tests without running the entire suite.

\* Tags are just strings in test names (like `@smoke`) or using Playwright’s metadata features.

2. Why Use Tags?

\* Organization: Group tests by type (`smoke`, `regression`, `integration`, `system`)

\* Selective Execution: Run only relevant tests (`npx playwright test --grep @smoke`)

\* Faster Feedback: Avoid running unnecessary tests during development or CI pipelines

\* Better Maintenance: Easier to debug, maintain, and scale large test suites

3. Analogy (Hotel / Library / Movies)

\* Hotel Analogy: Each test is a guest 🎫

\* `@smoke` = VIP guests (critical tests)

\* `@integration` = Guests connecting multiple services (integration tests)

\* `@system` = Guests checking the whole hotel (end-to-end system tests)

\* You can choose which guests to serve first based on the sticker (tag) — no need to handle everyone at once.

# 🔹 Example: Using Tags in Playwright

```javascript

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

// Smoke Test

test('@smoke Login page opens', async ({ page }) => {

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

await expect(page).toHaveTitle('Login');

});

// Integration Test

test('@integration API integration test', async ({ request }) => {

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

expect(response.ok()).toBeTruthy();

});

// System Test

test('@system Full checkout flow', async ({ page }) => {

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

await page.fill('#cart', '2');

await page.click('#checkout');

await expect(page.locator('.success')).toHaveText('Order Placed');

});

```

🔹 Running Tests by Tag

```bash

# Run only smoke tests

npx playwright test --grep @smoke

# Run only integration tests

npx playwright test --grep @integration

# Run system tests

npx playwright test --grep @system

# Run smoke OR system tests

npx playwright test --grep "@smoke|@system"

```

🔹 Summary

\* Tags = labels to organize tests (`@smoke`, `@integration`, `@system`)

\* Without tags: Must run all tests → slow, messy

\* With tags: Run selectively → faster, organized, scalable

\* Analogy: Stickers on movie tickets, books, or guests in a hotel → pick only what you need

What are Annotations

# 🔹 1️⃣ `test.skip`

Definition

\* Marks a test to be skipped temporarily. Playwright will not run it.

Analogy

\* Imagine a movie ticket 🎫 for a show that is canceled today. Nobody watches it until it’s fixed.

Real-Time Usage

\* Feature isn’t implemented yet.

\* Bug is blocking the test.

\* External service (like payment gateway) is down.

Code Example

```javascript

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

test.skip('Checkout test skipped - payment gateway down', async ({ page }) => {

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

await expect(page.locator('.success')).toHaveText('Order Placed');

});

```

What Happens Without `test.skip`?

\* Playwright runs the test → fails because the feature isn’t ready or service is down.

# 🔹 2️⃣ `test.only`

Definition

\* Runs only this test and ignores all other tests in the suite.

Analogy

\* VIP ticket holder 🎟️ gets special focus, everyone else waits.

Real-Time Usage

\* Debugging a failing test.

\* Running a single critical test during development.

Code Example

```javascript

test.only('Login test focused for debugging', async ({ page }) => {

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

await page.fill('#username', 'user');

await page.fill('#password', 'pass');

await page.click('#login');

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

});

```

What Happens Without `test.only`?

\* All tests in the suite run → slower feedback → harder to focus on a single failing test.

# 🔹 3️⃣ `test.describe`

Definition

\* Groups related tests together for a feature or module.

Analogy

\* A movie series 🎬 with multiple episodes under one franchise label.

Real-Time Usage

\* Organize tests for login, checkout, payment, etc.

\* Makes reporting and maintenance easier.

Code Example

```javascript

test.describe('Login Feature Tests', () => {

test('Login with valid credentials', async ({ page }) => {

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

await page.fill('#username', 'user');

await page.fill('#password', 'pass');

await page.click('#login');

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

});

test('Login with invalid credentials', async ({ page }) => {

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

await page.fill('#username', 'wronguser');

await page.fill('#password', 'wrongpass');

await page.click('#login');

await expect(page.locator('.error')).toHaveText('Invalid credentials');

});

});

```

What Happens Without `test.describe`?

\* Tests are not grouped → harder to organize large suites → reporting is less clear.

# 🔹 What is `test.slow` in Playwright?

\* `test.slow` is an annotation that tells Playwright:

> “This test is expected to take longer than normal, so don’t fail it just because it’s slow.”

\* By default, Playwright sets a timeout for each test (usually 30 seconds).

\* Long-running tests might take more than this, and without `test.slow`, they would fail automatically.

🔹 Analogy: Long Movie 🍿

\* Most movies last 2 hours.

\* One movie is an epic 5-hour film.

\* You mark it as slow, so the theater manager doesn’t cancel it for taking too long.

\* In Playwright: the slow test is like the epic movie — Playwright will allow extra time to finish.

🔹 Real-Life Examples of Slow Tests

1. End-to-End Payment Flow

\* Calls multiple APIs and third-party services.

\* Takes 40+ seconds sometimes.

2. File Import/Export

\* Large files can take time to upload/download.

3. Integration or Performance Tests

\* Waiting for multiple backend systems to respond.

🔹 Code Example

```javascript

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

// Mark this test as slow

test.slow('End-to-End Payment Flow', async ({ page }) => {

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

await page.fill('#cart', '3');

await page.click('#checkout');

// Simulate slow API call

await page.waitForTimeout(10000); // 10 seconds delay

await expect(page.locator('.success')).toHaveText('Order Placed');

});

```

🔹 What Happens Without `test.slow`?

\* Playwright uses the default timeout (30 seconds).

\* If the test takes longer → it fails, even if everything is correct.

# 🔹 What is a Flaky Test?

\* A flaky test is a test that sometimes passes and sometimes fails, even if the code hasn’t changed.

\* The results are unreliable, which makes debugging and CI pipelines frustrating.

🔹 Why Do Flaky Tests Happen?

1. Timing/Asynchronous issues

\* E.g., page takes longer to load, but the test tries to click a button too soon.

2. External dependencies

\* Third-party APIs, servers, or network requests fail occasionally.

3. Environment differences

\* Works on developer machine, fails on CI server.

4. Race conditions

\* Two actions depend on each other, but the timing is inconsistent.

🔹 Analogy: Flaky Movie 🎬

\* Imagine a movie theater projector that sometimes plays the movie perfectly, sometimes freezes.

\* Even though the movie file is fine, viewers can’t rely on watching it properly every time.

\* Similarly, flaky tests sometimes pass, sometimes fail, even when the app is correct.

🔹 Real-Time Example of Flaky Test

```javascript

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

test('Flaky login test', async ({ page }) => {

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

// Sometimes this element appears quickly, sometimes slowly

await page.click('#login-button');

// Test fails occasionally if the page isn't ready yet

await expect(page.locator('.dashboard')).toBeVisible();

});

```

\* Problem: Sometimes `.dashboard` appears after a delay → test fails → flaky.

🔹 How to Handle Flaky Tests

1. Mark with `test.fixme`

Ex:

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

test('Stable test - should load homepage', async ({ page }) => {

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

await expect(page).toHaveTitle(/Example Domain/);

});

test.fixme('Flaky test - sometimes fails due to network delay', async ({ page }) => {

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

await expect(page.locator('h1')).toHaveText('Example Domain');

});

Run tests in specific browser

We have 5 test cases in total, but only 2 of them should run in Firefox. In Playwright, this is done using test annotations and the `projects` configuration in your `playwright.config.ts`. Let me explain step by step.

Step 1: Configure browsers in `playwright.config.ts`

```ts

import { defineConfig, devices } from '@playwright/test';

export default defineConfig({

projects: [

{

name: 'chromium',

use: { ...devices['Desktop Chrome'] },

},

{

name: 'firefox',

use: { ...devices['Desktop Firefox'] },

},

{

name: 'webkit',

use: { ...devices['Desktop Safari'] },

},

],

});

```

✅ This defines 3 browser “projects”: Chromium, Firefox, and WebKit.

Step 2: Mark tests to run only in Firefox

Playwright has `test.only`, `test.skip`, and `test.describe`. To run only in a specific browser, use the `test.skip` with a condition:

```ts

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

// This test runs in all browsers except Firefox

test('Test case 1 - All except Firefox', async ({ browserName }) => {

test.skip(browserName === 'firefox', 'Skip in Firefox');

// test code

console.log('Running in', browserName);

});

// This test runs only in Firefox

test('Test case 2 - Firefox only', async ({ browserName }) => {

test.skip(browserName !== 'firefox', 'Run only in Firefox');

// test code

console.log('Running in', browserName);

});

```

\* `test.skip(condition, reason)` skips a test if the condition is true.

\* `browserName` is automatically provided by Playwright in each project.

Step 3: Run all tests

```bash

npx playwright test

```

\* The 2 Firefox-only tests will run only in Firefox.

\* The remaining tests will run in all browsers (or depending on your skip logic).

CONFIG FILE explanation

**🧩 1. testDir**

**➤ What it does**

Specifies the **folder where all your test files are located**.  
Playwright looks inside this directory to find and run tests.

**➤ Default**

testDir: './tests'

**➤ Examples**

testDir: './e2e' // Store your tests inside a folder named e2e

testDir: './specs' // Alternative folder name

**➤ Notes**

* It can be **relative to the config file**.
* You can have multiple folders if needed (by setting multiple configs or using projects)

**🧩 2. timeout**

**➤ What it does**

Sets the **maximum duration allowed for each test** to finish before Playwright stops it.

**➤ Default**

timeout: 30000 // 30 seconds

**➤ Example**

timeout: 60000 // Increases timeout to 1 minute per test

**➤ Notes**

* If your test exceeds this time, it fails with a timeout error.
* You can override per-test:
* test('slow test', async ({ page }) => { ... }).setTimeout(120000);

**🧩 3. expect.timeout**

**➤ What it does**

Specifies how long Playwright should **wait for an assertion** (e.g., expect(page).toHaveTitle()) before giving up.

**➤ Default**

expect: {

timeout: 5000 // 5 seconds

}

**➤ Example**

expect: {

timeout: 10000 // Wait up to 10s for elements to appear

}

**➤ Why needed**

Some UI elements take time to load or animate — this setting helps you avoid false failures due to slow rendering.

**🧩 4. fullyParallel**

**➤ What it does**

Controls whether **tests inside the same file** should run **in parallel** or **sequentially**.

**➤ Default**

fullyParallel: true

**➤ Example**

fullyParallel: false // Runs tests one after another within a file

**➤ When to use**

* ✅ true: Best for independent tests (faster runs).
* ❌ false: Use if tests share state (e.g., same user session or database).

**🧩 5. retries**

**➤ What it does**

Defines **how many times a failed test will automatically re-run** before marking as failed.

**➤ Default**

retries: process.env.CI ? 2 : 0

**➤ Examples**

retries: 0 // No retry (default for local)

retries: 2 // Try again twice if test fails

**➤ Why use it**

Useful for **flaky tests** that sometimes fail due to network delays or UI timing issues.

**➤ Tip**

You can also set per-project or per-test retries.

**🧩 6. workers**

**➤ What it does**

Specifies how many **test files run in parallel** (each in its own browser process).

**➤ Default**

workers: process.env.CI ? 1 : undefined

**➤ Examples**

workers: 1 // Run tests one by one

workers: 4 // Run 4 tests in parallel

workers: undefined // Let Playwright decide (usually CPU cores)

**➤ Notes**

* More workers = faster execution, but heavier CPU usage.
* CI systems often use 1 to ensure stability.

**🧩 7. reporter**

**➤ What it does**

Controls the **format of test results** and where they’re displayed or stored.

**➤ Common values**

| **Reporter** | **Description** |
| --- | --- |
| 'list' | Default; simple list of tests and results |
| 'dot' | Prints dots for each test (compact) |
| 'line' | Prints progress on one line (useful for CI logs) |
| 'html' | Generates a beautiful HTML report |
| 'json' | Saves results to a JSON file |
| 'junit' | XML output for CI tools like Jenkins |

**➤ Examples**

reporter: 'html' // Basic HTML report

reporter: [['html', { outputFolder: 'my-report' }]] // Custom folder

reporter: [['list'], ['junit', { outputFile: 'results.xml' }]] // Multiple reporters

**➤ How to view HTML report**

After run:

npx playwright show-report

**🧩 8. headless**

**➤ What it does**

Controls whether the browser runs **with or without a visible UI window**.

**➤ Example**

use: {

headless: true

}

**➤ Possible values**

| **Value** | **Description** |
| --- | --- |
| True | Browser runs **in background** (default, faster, for CI). |
| False | Browser runs **visibly** — useful for debugging locally. |

**➤ Example usage**

* **CI/CD pipelines:** true (no need for UI)
* **Local debugging:** false (you can see what happens)

use: {

headless: process.env.CI ? true : false

}

**🧩 9. viewport**

**➤ What it does**

Defines the **browser window size (width × height)** that each test starts with.

**➤ Example**

use: {

viewport: { width: 1280, height: 720 }

}

**➤ Common values**

| **Device** | **Width × Height** |
| --- | --- |
| Desktop | { width: 1280, height: 720 } |
| Laptop | { width: 1366, height: 768 } |
| Mobile | { width: 375, height: 667 } (iPhone 8) |

**➤ Notes**

* You can override per test:
* test.use({ viewport: { width: 1920, height: 1080 } });
* Or simulate real devices using:
* use: { ...devices['iPhone 13'] }

**🧩 10. screenshot**

**➤ What it does**

Controls **when Playwright captures screenshots** during test runs.

**➤ Example**

use: {

screenshot: 'only-on-failure'

}

**➤ Possible values**

| **Value** | **Meaning** |
| --- | --- |
| 'off' | Never take screenshots |
| 'only-on-failure' | Capture screenshot **only if a test fails** |
| 'on' | Capture a screenshot **after every test** (pass or fail) |

**➤ Why useful**

* 'only-on-failure' → Saves disk space, but still helps debug failed tests.
* 'on' → Good for detailed reporting or demos.

**➤ Screenshot location**

Saved automatically inside your test-results/ folder.

**🧩 11. video**

**➤ What it does**

Controls **whether Playwright records a video of each test execution**.

**➤ Example**

use: {

video: 'retain-on-failure'

}

**➤ Possible values**

| **Value** | **Description** |
| --- | --- |
| 'off' | No videos recorded |
| 'on' | Record all tests (can use a lot of storage) |
| 'retain-on-failure' | Record but **keep only failed test videos** |
| 'on-first-retry' | Record video only when test is retried |

**➤ Why useful**

* Helps visually debug what went wrong.
* 'retain-on-failure' is best for CI — efficient and informative.

**➤ Where saved**

In the test-results/ directory, automatically linked in the HTML report.

**🧩 12. trace**

**➤ What it does**

Enables **Playwright Tracing**, a powerful debugging feature that records:

* Screenshots, DOM snapshots, console logs, network calls, etc.
* You can replay everything step-by-step later.

**➤ Example**

use: {

trace: 'on-first-retry'

}

**➤ Possible values**

| **Value** | **Description** |
| --- | --- |
| 'off' | Don’t record traces |
| 'on' | Always record traces for every test |
| 'on-first-retry' | Record trace only when test fails first time (default good practice) |
| 'retain-on-failure' | Keep traces only for failed tests |

**➤ Why useful**

It gives you **a visual timeline** of what the test did.  
You can replay it like a video but with **inspectable DOM + network + console data**.

**➤ How to view**

After a failed test:

npx playwright show-trace trace.zip

**🧩 What is a frame (iframe)?**

A **frame** (or **iframe**) is like a **small webpage inside another webpage**.  
Example: A login box or ad area that loads from another site.

👉 You can’t directly click or type inside it —  
you must **first tell Playwright to go inside that frame**.

**🥇 1. Using frameLocator() → easiest and best way**

You directly point to the frame and the element you want inside it.

const frame = page.frameLocator('iframe#loginFrame');

await frame.locator('input#username').fill('John');

✅ **Best for most cases**  
✅ No need to “switch”  
✅ Works with nested frames

👉 Use this when you **know the iframe selector**.

**🥈 2. Using page.frame() → when frame has a name or URL**

You get the frame by its **name**, **URL**, or **index**.

const frame = page.frame({ name: 'loginFrame' });

// or: const frame = page.frame({ url: /login/ });

await frame.locator('input#username').fill('John');

✅ Useful when iframe has a **name or known URL**  
⚙️ You have to manually find the frame first

👉 Use this when the iframe **doesn’t have a clear selector**, but you know its **URL or name**.

**🧩 What Are Alerts / Pop-ups?**

In browsers, JavaScript can show small popups like:

1. alert("Message") → just shows a message.
2. confirm("Are you sure?") → OK / Cancel.
3. prompt("Enter name:") → asks for input.

They **block the page** until you click a button.  
Playwright handles them using an **event listener**.

**✅ 1. Handle an alert() popup**

**Example:**

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

test('Handle alert popup', async ({ page }) => {

await page.goto('https://the-internet.herokuapp.com/javascript\_alerts');

// Wait for the alert and accept it

page.once('dialog', async (dialog) => {

console.log('Alert text:', dialog.message());

await dialog.accept(); // Clicks OK

});

await page.click('button[onclick="jsAlert()"]');

});

✅ **dialog.message()** → gets alert text  
✅ **dialog.accept()** → clicks **OK**

**✅ 2. Handle a confirm() popup**

**Example:**

test('Handle confirm popup', async ({ page }) => {

await page.goto('https://the-internet.herokuapp.com/javascript\_alerts');

page.once('dialog', async (dialog) => {

console.log('Confirm text:', dialog.message());

await dialog.dismiss(); // Click Cancel

});

await page.click('button[onclick="jsConfirm()"]');

});

✅ Use dialog.accept() for OK  
✅ Use dialog.dismiss() for Cancel

**✅ 3. Handle a prompt() popup (with input)**

**Example:**

test('Handle prompt popup', async ({ page }) => {

await page.goto('https://the-internet.herokuapp.com/javascript\_alerts');

page.once('dialog', async (dialog) => {

console.log('Prompt text:', dialog.message());

await dialog.accept('Playwright User'); // Enter text and click OK

});

await page.click('button[onclick="jsPrompt()"]');

});

✅ You can send text to prompt using dialog.accept('text').

Visual Comparison

**🧠 What’s Visual Comparison?**

Visual comparison = checking that **what’s on the screen looks the same** as before.  
It’s like “spot the difference” for websites.

🟢 **If page looks same → Test Passes**  
🔴 **If page looks different → Test Fails and shows diff image**

**🧩 Simple Example Code**

// visual.spec.js

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

test('Visual comparison example', async ({ page }) => {

// Step 1: Open a webpage

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

// Step 2: Take screenshot and compare

await expect(page).toHaveScreenshot('homepage.png');

});

**🪄 What Happens Behind the Scenes**

**🏃‍♂️ First Test Run**

* Playwright visits https://example.com
* Takes a screenshot
* Saves it as a **baseline** → homepage.png in \_\_screenshots\_\_ folder  
  ✅ Test passes (baseline created)

**🏃‍♀️ Second Test Run**

* Playwright takes a new screenshot again
* Compares it with the **baseline image**
* If both images match → ✅ pass
* If there’s a small layout or color change → ❌ fail and show diff

**📁 Folder Example**

tests/

├── visual.spec.js

├── \_\_screenshots\_\_/

│ └── homepage.png <-- baseline image

└── test-results/

├── homepage-diff.png <-- highlights changed pixels

└── homepage-actual.png <-- new screenshot

**🖼 Example Diff View (Conceptually)**

| **Baseline** | **New** | **Diff** |
| --- | --- | --- |
| Old version of page | New version of page | Highlighted red area showing changed part |

**⚙️ Update Baseline When UI Changes Intentionally**

If you *expect* the UI to change (like a design update):

npx playwright test --update-snapshots

This replaces the old baseline with the new one.

**🧩 Compare Only a Specific Element**

If you want to compare just part of the page (e.g., a button):

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

test('Compare button look', async ({ page }) => {

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

const button = page.locator('a');

await expect(button).toHaveScreenshot('button.png');

});