

# Playwright

Version 1.64 · verified 2026-11-02

End-to-end testing with Playwright Test 1.64 — locators, retrying assertions, fixtures, auth state, mocking, traces, CI sharding and flaky tests.

## Contents


Mental model ..... 2

Setup and config ..... 2

Locators ..... 4

Web-first assertions ..... 6

Fixtures and page objects ..... 7

Reusing authentication ..... 9

Network mocking ..... 11

Debugging: UI mode and traces ..... 13

Parallelism and sharding in CI ..... 14

Flaky-test triage ..... 15

Further reading ..... 17

## Mental model

Every Playwright action and web-first assertion waits for the page: `click()` waits until the locator resolves to exactly one element that is visible, stable, enabled and receiving events, and `expect(locator)` retries until it passes or times out. A locator is a recipe re-run on every use, not a handle to a DOM node, so it never goes stale. If you reach for `waitForTimeout`, the locator or the assertion is wrong — fix that instead of adding sleep.

## Setup and config

| Command | Does |
| --- | --- |
| `npm init playwright@latest` | Scaffold config, example test |
| `npx playwright install` | Download browser builds |
| `npx playwright test` | Run all tests, headless |
| `npx playwright test --project=firefox` | One project only |
| `npx playwright show-report` | Open the HTML report |

Playwright 1.64 supports Node 22, 24 and 26. A **project** is a named set of options (browser, device, `storageState`) applied to the test files; the same test runs once per project. `webServer` starts your app before the run and waits for its URL, so tests can use relative `page.goto('/')` against `baseURL`.

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

export default defineConfig({
  fullyParallel: true,
  forbidOnly: !!process.env.CI,
  retries: process.env.CI ? 2 : 0,
  use: { baseURL: 'http://localhost:3000',
    trace: 'on-first-retry' },
  webServer: { command: 'npm run start',
    url: 'http://localhost:3000',
    reuseExistingServer: !process.env.CI },
  projects: [{ name: 'chromium',
```

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

> **Note:** Component testing changed in 1.62 to a “stories and gallery” model driven by the built-in `mount` fixture. The `@playwright/experimental-ct-*` packages are no longer published.

## Locators

| Locator | Prefer for |
|---|---|
| `getByRole('button', { name })` | Anything interactive — first choice |
| `getByLabel('Email')` | Form controls |
| `getByText('Saved')` | Non-interactive text |
| `getByTestId('cart')` | An explicit test contract |
| `locator('css=…')` | Last resort |

Locate the way a user or screen reader perceives the page: role and accessible name first. A role locator that

cannot be written is often an accessibility bug in the page, not in the test. Narrow with chaining (`list.getByRole('listitem')`) or `filter({ hasText })` instead of indexes.

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

test('add to cart', async ({ page }) => {
  await page.goto('/products')
  const row = page.getByRole('listitem')
    .filter({ hasText: 'Blue mug' })
  await row.getByRole('button', { name: 'Add' })
    .click()
})
```

> **Gotcha:** Locators are strict: an action on a locator that matches two elements throws instead of picking one. `first()` and `nth()` silence it but are discouraged — when the page changes, they click the wrong element without failing.

## Web-first assertions

| Assertion | Retries until |
|---|---|
| `toBeVisible()` | Element is visible |
| `toHaveText('Saved')` | Text matches |
| `toHaveCount(3)` | Locator matches 3 elements |
| `toHaveURL(/\/orders/)` | Page URL matches |
| `expect.soft(…)` | Records failure, keeps going |

Assertions on a locator or page re-check until they pass or the expect timeout runs out — 5 s by default, separate from the 30 s test timeout. Assert the outcome the user sees after an action; that wait replaces every manual sleep.

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

test('save settings', async ({ page }) => {
  await page.goto('/settings')
  await page.getByLabel('Name').fill('Ana')
  await page.getByRole('button',
    { name: 'Save' }).click()
  await expect(page.getByRole('status'))
    .toHaveText('Saved')
})
```

> **Gotcha:** `expect(await loc.isVisible()).toBe(true)` checks once and returns immediately — no retry. Put the locator inside `expect` and `await` the assertion: `await expect(loc).toBeVisible()`.

## Fixtures and page objects

| Concept | Meaning |
|---|---|
| `{ page }` argument | Built-in fixture, fresh per test |

| Concept | Meaning |
| --- | --- |
| `test.extend({ … })` | Define your own fixtures |
| `await use(value)` | Hand value to test; code after is teardown |
| `{ scope: 'worker' }` | Share across tests in a worker |

Every test gets a fresh browser context, so tests do not share cookies or storage. Fixtures are set up only when a test asks for them. Keep page objects thin — named locators plus a few user-level actions — and expose them as fixtures rather than constructing them in every test.

```
import { test as base, type Page }
  from '@playwright/test'
class Auth {
  constructor(private p: Page) {}
  async as(user: string, pass: string) {
    await this.p.goto('/login')
    await this.p.getByLabel('User').fill(user)
    await this.p.getByLabel('Pass').fill(pass)
```

```
    await this.p.getByRole('button',
      { name: 'Sign in' }).click()
  }
}
const test = base.extend<{ auth: Auth }>({
  auth: ({ page }, use) => use(new Auth(page)),
})
```

> **Tip:** Keep assertions in tests, not in page objects. A page object that asserts hides what the test actually checks.

## Reusing authentication

| Piece | Role |
|---|---|
| `auth.setup.ts` | Logs in once, saves state |
| `storageState: 'playwright/.auth/user.json'` | Tests start logged in |
| `dependencies: ['setup']` | Setup runs first |

| Piece | Role |
| --- | --- |
| `playwright/.auth` in `.gitignore` | Keep session secrets out of Git |

Log in once in a **setup project**, save cookies and local storage with `context().storageState({ path })`, and point every browser project at that file. Logging in through the UI in each test is slow and turns the login page into a single point of failure for the whole suite.

```
import { test as setup } from '@playwright/test'

const file = 'playwright/.auth/user.json'

setup('authenticate', async ({ page }) => {
  await page.goto('/login')
  await page.getByLabel('User').fill('qa')
  await page.getByLabel('Pass').fill('secret')
  await page.getByRole('button',
    { name: 'Sign in' }).click()
```

```
  await page.waitForURL('/dashboard')
  await page.context()
    .storageState({ path: file })
})
```

> **Warning:** One shared account works only while tests do not change server-side state. When a test edits settings that another parallel test reads, give each worker its own account.

## Network mocking

| API | Does |
|---|---|
| `page.route(url, handler)` | Intercept matching requests |
| `route.fulfill({ json })` | Answer without hitting the server |
| `route.continue()` | Let it through, optionally modified |

| API | Does |
| --- | --- |
| `route.abort()` | Simulate a network failure |
| `routeFromHAR(file)` | Replay recorded responses |

Mock the boundary you do not own (third-party APIs, payment providers, flaky staging services), not your own backend — that is what the end-to-end test exists to cover.

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

test('rates from API', async ({ page }) => {
  await page.route('**/api/rates', (route) =>
    route.fulfill({ json: [{ code: 'EUR' }] }))
  await page.goto('/rates')
  await expect(page.getByText('EUR'))
    .toBeVisible()
})
```

> **Gotcha:** Register routes **before** `page.goto()`. Requests fired during page load go out before a route added afterwards exists, and the test silently hits the real API.

## Debugging: UI mode and traces

| Command | Does |
|---|---|
| `npx playwright test --ui` | Watch, time-travel, pick locators |
| `npx playwright test --debug` | Step through with the Inspector |
| `npx playwright show-trace trace.zip` | Open a recorded trace |
| `--trace=on` | Force tracing for this run |
| `npx playwright codegen` | Record actions as code |

A trace records every action with DOM snapshots, console, network and source. `trace: 'on-first-retry'` captures one only when a test fails and is retried, which

keeps passing runs cheap. On CI, upload the HTML report as an artifact; failed tests link their traces.

```
npx playwright test tests/cart.spec.ts:12 --ui
npx playwright test --last-failed --trace=on
npx playwright show-report
```

## Parallelism and sharding in CI

| Setting | Effect |
| --- | --- |
| `fullyParallel: true` | Tests in one file run in parallel too |
| `workers: 1` on CI | Docs' default for stability |
| `--shard=2/4` | Run the 2nd quarter of tests |
| `reporter: 'blob'` | Per-shard raw results |
| `npx playwright merge-reports` | One HTML report from blobs |

Workers are separate processes, so tests in one file run in order unless `fullyParallel` is set. To scale beyond

one machine, shard across CI jobs; with `fullyParallel`, shards are balanced per test rather than per file. Each shard writes a blob report and a final job merges them.

```
# each CI job, matrix index 1..4
npx playwright test --shard=1/4 --reporter=blob
# final job, after downloading all blobs
npx playwright merge-reports --reporter html \
  ./all-blob-reports
```

> **Tip:** `test.only` left in a commit makes CI run one test and pass. `forbidOnly: !!process.env.CI` turns that into a failure.

## Flaky-test triage

| Tool | Use |
|---|---|
| `retries: 2` | Pass-on-retry is reported as **flaky** |
| `--repeat-each=20` | Reproduce a suspect locally |

| Tool | Use |
| --- | --- |
| `--shuffle` | Expose order dependence (1.64) |
| `--fail-on-flaky-tests` | Make flakes fail the build |
| `{ lock: 'name' }` | Never run two holders at once (1.63) |

Retries keep the pipeline green but are a reporting tool, not a fix: a flaky test is one that failed and then passed, and the report lists it separately. Triage from the trace of the failed attempt. The usual causes are a missing web-first assertion before the next action, shared server state between parallel tests, and an unmocked third-party call.

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

test('edit global settings',
  { lock: 'user-settings' },
  async ({ page }) => {
```

```
  await page.goto('/admin/settings')
})
```

> **Warning:** A retry hides real races too. Track the flaky count over time and fix or quarantine; a suite where every failure gets retried stops telling you anything.

## Further reading

- Playwright — Best practices
- Playwright — Locators
- Playwright — Fixtures
- Playwright — Authentication
- Playwright — Sharding
- Playwright — Release notes