Playwright Automation Handbook — JavaScript

A compact practical handbook to get started and be productive with Playwright for end-to-end testing using JavaScript. Includes setup, patterns, examples, CI tips, and best practices.

Table of Contents

1. Introduction

Playwright is a modern end-to-end testing framework that enables reliable automation for web applications across Chromium, Firefox, and WebKit. It's designed for speed, reliability, and cross-browser parity.

This handbook focuses on using Playwright with JavaScript (Node.js) and the built-in Playwright Test Runner (often invoked via `npx playwright test`).

2. Setup & Installation

Prerequisites: Node.js (LTS recommended).

Install Playwright and its browsers in a new project directory.

```
# initialize project
npm init -y

# install Playwright test runner and browsers
npm i -D @playwright/test

# install browsers (recommended)
npx playwright install

# optionally add a test script in package.json
# "scripts": { "test": "npx playwright test" }
```

3. Project Structure

A typical Playwright project layout:

```
my-playwright-project/
■■ tests/              # test files (e.g., login.spec.js)
■■ pages/              # page objects (POM)
■■ fixtures/           # custom fixtures
■■ playwright.config.js
■■ package.json
■■ README.md
```

4. Playwright Test Runner (pw-test)

Create a simple test file under the tests directory. Files named *.spec.js or *.test.js are picked by default.

```
// tests/example.spec.js
const { test, expect } = require('@playwright/test');

test('homepage has title', async ({ page }) => {
  await page.goto('https://example.com');
  await expect(page).toHaveTitle(/Example Domain/);
});
```

5. Locators & Selectors

Playwright's `locator` API is recommended over raw selectors — it handles waiting and re-querying automatically.

Common selector types: text, css, xpath, role, data-testid, test-id, and test attributes.

```
// using locator
const loginButton = page.locator('text=Log in');

// role selector (preferred for accessibility)
await page.getByRole('button', { name: 'Submit' }).click();

// attribute
await page.locator('[data-testid="username"]').fill('user@example.com');
```

6. Actions & Assertions

Playwright supports actions like click, fill, selectOption, hover, waitFor, etc.

Assertions use expect() from @playwright/test and integrate with auto-waiting.

```
await page.fill('#email', 'me@example.com');
await page.click('button[type=submit]');
await expect(page.locator('.welcome')).toContainText('Welcome');
```

7. Fixtures & Test Hooks

Fixtures provide powerful dependency injection for tests. Use `test.extend()` to create custom fixtures.

Hooks: test.beforeAll, test.afterAll, test.beforeEach, test.afterEach for setup/teardown.

```
// example fixture
const base = require('@playwright/test');

const test = base.test.extend({
  apiContext: async ({ request }, use) => {
    # create API context or perform auth
    await use(request);
  }
});

test('uses apiContext', async ({ apiContext }) => {
  const r = await apiContext.get('/api/status');
  expect(r.ok()).toBeTruthy();
});
```

8. Parallelism, Projects & Browsers

Playwright runs tests in parallel by worker processes. Configure `workers` in playwright.config.js.

Define `projects` for combinations of browsers or device settings (e.g., Desktop Chromium, Firefox, Mobile emulation).

```
// playwright.config.js
module.exports = {
  projects: [
    { name: 'chromium', use: { browserName: 'chromium' } },
    { name: 'firefox',  use: { browserName: 'firefox' } },
  ],
  workers: 4,
};
```

9. Debugging, Tracing & Screenshots

Use `PWDEBUG=1 npx playwright test` to open the Playwright inspector for debugging.

Enable tracing to capture a trace of a failing test for later inspection.

```
# enable trace for a single test run
npx playwright test --trace on

# capture screenshot in test
await page.screenshot({ path: 'screenshots/fail.png', fullPage: true });
```

10. Page Object Model (POM)

POM helps organize selectors and actions for page-specific behavior. Keep locators and methods in classes.

```
// pages/loginPage.js
class LoginPage {
```

```
  /**
   * @param {import('@playwright/test').Page} page
   */
  constructor(page) {
    this.page = page;
    this.username = page.locator('[data-testid="username"]');
    this.password = page.locator('[data-testid="password"]');
    this.submit = page.getByRole('button', { name: 'Sign in' });
  }
  async login(user, pass) {
    await this.username.fill(user);
    await this.password.fill(pass);
    await this.submit.click();
  }
}
module.exports = { LoginPage };
```

11. API Testing with Playwright

Playwright provides APIRequestContext for HTTP requests — useful for setup/teardown or API-level checks.

```
// example using request fixture
test('api health', async ({ request }) => {
  const r = await request.get('https://api.example.com/health');
  expect(await r.json()).toHaveProperty('status', 'ok');
});
```

12. CI/CD Integration

Common CI steps: install dependencies, install browsers, run `npx playwright test --reporter=html` and publish artifacts (screenshots, traces, HTML report).

Example: GitHub Actions uses `actions/setup-node` then `npx playwright install --with-deps`.

```
# GitHub Actions job (snippet)
- uses: actions/checkout@v3
- uses: actions/setup-node@v4
  with:
    node-version: 18
- run: npm ci
- run: npx playwright install --with-deps
- run: npx playwright test --reporter=html
- name: Upload Playwright Report
  uses: actions/upload-artifact@v4
  with:
    name: playwright-report
    path: playwright-report/
```

13. Reporting & Test Results

Playwright includes reporters: list, dot, line, json, html. The HTML reporter is useful for interactive failure inspection.

Generate HTML report with `npx playwright show-report` after a run.

```
# run tests and generate html report
npx playwright test --reporter=html
npx playwright show-report
```

14. Best Practices & Tips

1. Prefer `locator()` and `getByRole()` for resilient selectors.

2. Keep tests deterministic: avoid time-based waits; use waitFor* when needed.

3. Use fixtures for shared setup and tear-down to keep tests isolated.

4. Use Page Object Model for maintainability on larger suites.

5. Run tests in CI with a fixed Node and Playwright version to ensure reproducibility.

6. Capture traces and screenshots on failures to ease debugging.

15. Example: End-to-End Test Suite

A small example combining POM and tests for a login flow.

```
// tests/login.spec.js
const { test, expect } = require('@playwright/test');
const { LoginPage } = require('../pages/loginPage');

test('user can log in', async ({ page }) => {
  const login = new LoginPage(page);
  await page.goto('https://example.com/login');
  await login.login('user@example.com', 'password123');
  await expect(page).toHaveURL(/dashboard/);
});
```

16. Troubleshooting

Common issues and fixes:

- Tests flaky: increase timeout or improve selector reliability; use `await expect(locator).toBeVisible()` before actions.

- Browser installation failures on CI: use `npx playwright install --with-deps` and ensure required system packages are available.

- Authentication flows: consider using API calls to obtain auth tokens and set storageState for faster tests.

17. Resources & Further Reading

Official docs: https://playwright.dev — (guides, API reference, and examples).

Playwright GitHub repo and community examples.

Use Playwright's codegen for quick selector generation: `npx playwright codegen `.

Credits

Generated with assistance from ChatGPT — Playwright Automation Handbook (JavaScript). Customize and extend for your team's needs.