WRITING RELIABLE AUTOMATED TESTS

A guide to writing Playwright tests that aren't flaky.

CARDIFFQA.COM

Best Practices for Stability

✓ Run tests 100 times

Before submitting a test, run it many times to make sure it is stable and can be run in parallel.

```
# Run the test 100 times
npx playwright test tests/ecommerce.spec.ts --grep "should add item to cart" --re-
peat-each 100
```

✗ Don't use page.waitForTimeout

Avoid waiting for a fixed time. It makes tests slow and unreliable. Always wait for a specific state (e.g., an element appearing).

```
// Bad - Never do this
await page.waitForTimeout(5000);
```

Finding Elements

✓ Use user-facing locators

Find elements by their role (like "button") or text. This is how a real user sees the page.

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

✓ Use test IDs to find elements

Use data-testid attributes and getByTestId() to find elements. This works even if the page design or layout changes.

```
// Good
await page.getByTestId("submit-button").click();
```

✗ Don't use brittle selectors

Avoid using XPath or specific CSS classes. If the page layout changes slightly, these tests will break.

```
// Bad
await page.click(".product-list > div:nth-child(3) .price span:last-child");
```

Validating Outcomes

✓ Use web-first assertions

These assertions wait automatically until the condition is met.

```
// Good
await expect(page.getByTestId("result")).toHaveText("Complete!");

// Bad
const text = await page.getByTestId("result").textContent();
expect(text).toBe("Complete!");
```

✓ Test for ranges of values

If a result changes every time (like heart rate that updates), check that it falls within a correct range instead of looking for an exact match.

```
// Good - these are reasonable outputs.
expect(bpm).toBeGreaterThan(40);
expect(bpm).toBeLessThanOrEqual(210);
```

Parallelization

✓ Keep tests independent

Each test should run on its own. The result of one test should not change the result of another. All tests must be able to run in parallel.

```
test("test A", async ({ page }) => {
// ... does something
});

test("test B", async ({ page }) => {
// ... runs safely regardless of test A
});
```

Pitfalls to Avoid

✗ Don't rely on retries to fix bugs

Retrying a test hides the problem. Only use retries for issues you can't control. It is better to fix the root cause.

✓ Use Promise.all to prevent race conditions

Use this when you need to wait for a response at the same time you perform an action (like a click).

```
// Good
await Promise.all([
 page.waitForResponse("**/data"), // Start waiting
 page.getByTestId("my-button").click(), // Click triggers the response
]);
```

✓ Use toPass and poll to avoid instant failures

Use toPass to keep trying a set of steps until they work. Use poll to keep checking a value (like an API check) until it matches what you want.

```
// Keeps retrying until the expectation passes
await expect(async () => {
 const newCount = await articles.count();
 expect(newCount).toBeGreaterThan(initialCount);
}).toPass({ timeout: 3000 });

// Using poll to wait for a specific value
await expect
.poll(async () => {
 const response = await page.request.get("/status");
 return response.json().status;
 })
.toBe("completed");
```

✗ Don't wait for a response after the action

If the response comes back too fast, your test will miss it and fail.

```
// Bad
await page.getByTestId("submit-button").click();
// The response might have already happened!
await page.waitForResponse("**/api/data");
```

✗ Don't rely only on afterEach for cleanup

If afterEach fails or crashes, the next test will start with leftover data from the previous test. This can cause the next test to fail even though it did nothing wrong.

```
// Bad - If cleanup API call fails, next test starts dirty
test.afterEach(async () => {
 await fetch("/api/init");
});

// Good - Use beforeEach to reset state via API, even if cleanup failed
test.beforeEach(async () => {
 await fetch("/api/init");
});
```

Network Dependencies

✗ Don't test external sites

Only test what you control. Don't test links to other websites. They can change and break your test.

```
// Bad - Testing a site you don't own
await page.goto("https://google.com");
```

✓ Mock external APIs

Use page.route() to fake responses from other servers. This makes tests faster and reliable.

```
// Good - Mock the data
await page.route("**/api/third-party/data", route =>
 route.fulfill({ status: 200 }));
```

Additional Tips

✓ Quarantine flaky tests

If a test is flaky, tag it so it doesn't break the main build while you fix it.

```
test("@quarantined flaky test", async ({ page }) => {
// ...
});
```

✓ Use throttling to find timing bugs

Test your app with slow internet or a slow CPU. This helps you find bugs that only happen when things load slowly.

Keep your test environment the same

Make sure every place you run tests is exactly the same. The settings and tools should be just like the real website users see.

Lint for missing awaits

Use tools to find missing 'await' keywords. If you forget 'await', the test continues before the action finishes.

```
// Bad - Missing await
page.click("button");

// Good
await page.click("button");
```