Vì sao là Playwright chứ không phải Selenium

- **Tự chờ (auto-wait).** Selenium bắt bạn tự viết lệnh chờ khắp nơi. Playwright tự chờ element hiện ra, hết bị che, bấm được rồi mới bấm. Đây là lý do lớn nhất khiến test Playwright ít "lúc xanh lúc đỏ".
- **Công cụ debug tốt.** Có chế độ xem lại từng bước như xem video tua ngược (Trace Viewer).
- **Cài một phát là chạy.** Một câu lệnh, có sẵn cả Chrome, Firefox, Safari.
- **Ghi thao tác ra code.** Bạn bấm trên trình duyệt, nó viết code hộ bạn (Codegen) — cực hợp cho người mới.

PHẦN 01

Chuẩn bị máy tính

Chuẩn bị máy tính

Làm theo đúng thứ tự dưới đây. Khoảng 20 phút. Hướng dẫn cho Windows, vì đó là máy bạn đang dùng.

Bước 1 — Cài Node.js

Node.js là chương trình giúp máy tính chạy được code JavaScript ở ngoài trình duyệt. Playwright chạy trên nền nó.

1. Vào nodejs.org, tải bản có chữ **LTS** (bản ổn định).
2. Cài như mọi phần mềm khác — bấm Next đến hết.
3. Mở **PowerShell** (bấm phím Windows, gõ "powershell") và kiểm tra:

```
> node -v
v22.14.0
> npm -v
10.9.2
```

Nếu hiện ra số phiên bản như trên là xong. Nếu báo "not recognized", hãy đóng PowerShell và mở lại.

Bước 2 — Cài VS Code

VS Code là nơi bạn viết code. Tải ở code.visualstudio.com. Sau khi cài, mở lên và cài thêm 1 tiện ích:

- Bấm biểu tượng Extensions ở thanh trái (hoặc `Ctrl` + `Shift` + `X`)
- Gõ **Playwright Test for VSCode** → Install. Đây là tiện ích chính thức, sau này bạn sẽ chạy test bằng cách bấm nút tam giác thay vì gõ lệnh.

Bước 3 — Tạo project đầu tiên

Trong PowerShell, gõ từng dòng:

```
> cd C:\Users\Admin\Desktop
```

```
> mkdir hoc-playwright
> cd hoc-playwright
> npm init playwright@latest
```

Nó sẽ hỏi vài câu. Trả lời như sau:

Câu hỏi	Chọn	Vì sao
TypeScript or JavaScript?	**TypeScript**	Gõ sai tên hàm là VS Code báo đỏ ngay, không phải chạy mới biết.
Where to put your end-to-end tests?	**tests**	Mặc định, giữ nguyên.
Add a GitHub Actions workflow?	**false**	Đây là phần chạy tự động trên server, để sau.
Install Playwright browsers?	**true**	Bắt buộc — đây là các trình duyệt để test.

Bước 4 — Hiểu những gì vừa được tạo ra

Mở thư mục `hoc-playwright` bằng VS Code. Bạn sẽ thấy:

hoc-playwright/ cấu trúc project

```
hoc-playwright/
├── node_modules/          ← thư viện tải về. KHÔNG bao giờ sửa, không đọc.
├── tests/
│   └── example.spec.ts    ← bài test mẫu. Code của BẠN nằm ở đây.
├── tests-examples/        ← ví dụ thêm, xoá được.
├── playwright.config.ts   ← bảng điều khiển: chạy trình duyệt nào, chờ bao lâu...
├── package.json           ← khai báo project dùng thư viện gì.
└── .gitignore
```

BA TỪ BẠN SẼ GẶP HÀNG NGÀY

- `npm` — người đi chợ. Nó tải thư viện về (`npm install`).
- `npx` — người sai vặt. Nó chạy một công cụ đã tải về (`npx playwright test`).

- `node_modules` — cái kho. Nặng vài trăm MB, sinh ra tự động, mất thì chạy `npm install` là có lại.

Bước 5 — Chạy thử

```
> npx playwright test
Running 6 tests using 6 workers
  6 passed (4.2s)

> npx playwright test --ui   # mở giao diện, nên dùng cái này
```

Lệnh thứ hai mở ra **UI Mode** — một cửa sổ có danh sách test bên trái, trình duyệt bên phải. Đây là nơi bạn sẽ sống trong vài tuần tới. Bấm nút ▶ cạnh một test để chạy nó.

TỰ LÀM THỬ

1. Chạy `npx playwright test --ui`.
2. Bấm chạy test *has title*. Xem nó chuyển xanh.
3. Mở `tests/example.spec.ts`, sửa chữ `'Playwright'` thành `'Cypress'`, lưu lại, chạy lại.
4. Nhìn kỹ dòng báo lỗi màu đỏ. Bạn vừa thấy một test FAIL thật — và đó là điều tốt.

PHẦN 02

Đủ code để bắt đầu

Bạn không cần học hết JavaScript. Bạn cần đúng 7 thứ dưới đây — chúng

chiếm khoảng 95% code trong một bộ test. Đọc một lượt, không cần thuộc; quay lại tra khi cần.

1. Biến — đặt tên cho một giá trị

```
biến

const email = 'tester@newwave.vn';    // const = không đổi nữa. Dùng 95% trường hợp
let soLanThu = 0;                      // let = sẽ đổi giá trị sau này.
soLanThu = 1;                          // hợp lệ vì là let
```

Quy tắc: cứ dùng `const`. Chỉ đổi sang `let` khi VS Code báo lỗi vì bạn gán lại giá trị.

2. Chuỗi — và cách ghép chuỗi

```
chuỗi

const host = 'https://genesis-test.vn';

// Dấu backtick ` cho phép nhét biến vào giữa chuỗi bằng ${...}
const url = `${host}/sign-in`;    // → https://genesis-test.vn/sign-in
```

Dấu `` ` `` (backtick) nằm cạnh phím số 1. Bạn sẽ dùng nó rất nhiều để ghép URL.

3. Hàm — đóng gói một nhóm việc và đặt tên

```
hàm

// Cách viết truyền thống
function cong(a, b) {
  return a + b;
}

// Arrow function — cách viết ngắn, bạn sẽ thấy dạng này khắp nơi
const cong = (a, b) => a + b;

cong(2, 3);  // gọi hàm → 5
```

Dấu `⇒` đọc là "arrow". Khi bạn thấy `async () ⇒ { ... }` trong bài test, đó chính là một hàm không tên được truyền vào cho Playwright chạy.

4. Object — một cái "hồ sơ" gồm nhiều trường

object

```
const user = {
  email: 'tester@newwave.vn',
  password: 'Test@12345',
  role: 'Quản trị viên',
};

user.email;   // đọc một trường → 'tester@newwave.vn'
```

Object là cách bạn gom dữ liệu test lại cho gọn, thay vì rải rác các biến rời.

5. Class — bản thiết kế; đây là nền của Page Object

Hãy hình dung `class` là *bản vẽ một màn hình*. Bản vẽ ghi: màn hình này có những element nào, và làm được những hành động gì. Từ bản vẽ đó, mỗi bài test tạo ra một "bản sao" để dùng.

class

```
class ManHinhDangNhap {
  // constructor = việc làm lúc tạo ra bản sao
  constructor(page) {
    this.page = page;              // this = chính bản sao này
  }

  // method = một hành động màn hình này làm được
  async dangNhap(email, matKhau) {
    await this.page.getByLabel('Email').fill(email);
  }
}

// Dùng:
```

```
const manHinh = new ManHinhDangNhap(page);
await manHinh.dangNhap('a@b.com', '123456');
```

6. async / await — thứ quan trọng nhất trong phần này

Mọi việc trên trình duyệt đều *tốn thời gian*: mở trang mất 2 giây, bấm nút xong trang phải load lại. Máy tính mặc định không chờ — nó ra lệnh xong là chạy tiếp ngay dòng dưới.

`await` có nghĩa: "**đứng đây chờ việc này xong đã, rồi mới đi tiếp**". Và một hàm có chứa `await` bên trong thì phải được đánh dấu `async`.

✓ CÓ AWAIT

```
await page.goto('/sign-in');
await page.getByLabel('Email')
          .fill('a@b.com');
await page.getByRole('button')
          .click();
```

Mở trang xong mới điền, điền xong mới bấm. Đúng thứ tự.

✗ QUÊN AWAIT

```
page.goto('/sign-in');
page.getByLabel('Email')
    .fill('a@b.com');
page.getByRole('button')
    .click();
```

Điền vào lúc trang chưa mở xong. Test lúc xanh lúc đỏ không rõ lý do.

LỖI SỐ 1 CỦA NGƯỜI MỚI

Quên `await`. Nếu test của bạn "chạy máy này thì được, máy kia thì hỏng", hoặc "chạy 10 lần đỏ 3 lần" — 80% khả năng là thiếu một chữ `await` ở đâu đó. Quy tắc thô nhưng hiệu quả: *hễ dòng nào có chữ `page` hoặc `expect` thì phải có `await` đứng đầu.*

7. import / export — chia code ra nhiều file

```
import / export

// file: src/pages/LoginPage.ts — cho phép file khác dùng class này
export class LoginPage { ... }

// file: tests/login.spec.ts — lấy về dùng
import { LoginPage } from '../src/pages/LoginPage';
```

Dấu `../` nghĩa là "lùi ra thư mục cha", `./` nghĩa là "cùng thư mục". VS Code thường tự điền hộ bạn.

Còn TypeScript thì sao?

TypeScript chỉ là JavaScript cộng thêm phần *ghi chú kiểu dữ liệu*. Bạn nói rõ "biến này là chuỗi", "hàm này không trả về gì":

```
TypeScript

const email: string = 'a@b.com';        // : string = đây là chuỗi
const soLan: number = 3;                  // : number = đây là số

async login(email: string, pass: string): Promise<void> { ... }
//                                          ↑ Promise<void> = hàm async không trả
```

Đổi lại, VS Code sẽ gợi ý cho bạn: gõ `page.` là hiện ra danh sách mọi thứ `page` làm được, và gạch đỏ ngay khi bạn gõ sai tên. Với người mới, đây là món quà chứ không phải gánh nặng.

TỰ LÀM THỬ

Mở `tests/example.spec.ts` và chỉ làm một việc: đọc từng dòng, chỉ ra đâu là `const`, đâu là arrow function, đâu là `await`. Chưa cần hiểu nó làm gì.

PHẦN 03

Mổ xẻ một bài test

Đây là một bài test hoàn chỉnh, ngắn nhất có thể. Chúng ta sẽ đọc từng dòng.

tests/dang-nhap.spec.ts — bài test đầy đủ

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

test('Đăng nhập thành công với tài khoản hợp lệ', async ({ page }) => {
  await page.goto('https://demo.playwright.dev/todomvc');

  await page.getByPlaceholder('What needs to be done?').fill('Viết test đầu tiên');
  await page.getByPlaceholder('What needs to be done?').press('Enter');

  await expect(page.getByText('Viết test đầu tiên')).toBeVisible();
});
```

Dòng	Nghĩa là gì
`import { test, expect }`	Lấy về 2 công cụ: `test` để khai báo một bài test, `expect` để kiểm chứng kết quả.
`test('tên', async ... )`	Khai báo một bài test. Chữ trong dấu nháy là **tên hiển thị trong báo cáo** — hãy viết như tên test case của bạn, bằng tiếng Việt cũng được.
`({ page })`	Playwright đưa cho bạn một **tab trình duyệt sạch**, tên là `page`. Mỗi test có tab riêng, không dính dữ liệu của test khác.
`page.goto(...)`	Mở địa chỉ. Tương đương gõ URL vào thanh địa chỉ rồi Enter.
`.fill(...)` / `.press(...)`	Hành động: điền chữ vào ô, bấm phím.

`await expect(...).toBeVisible()`	Kiểm chứng: "tôi mong element này hiện ra". Nếu không hiện → test đỏ.

Ba công cụ tổ chức bạn sẽ dùng ngay

`describe / beforeEach / step`

```
// describe = gom các test cùng một chức năng vào một nhóm
test.describe('Genesis Id > Đăng nhập', () => {

  // beforeEach = Pre-Condition. Chạy TRƯỚC MỖI test trong nhóm.
  test.beforeEach(async ({ page }) => {
    await page.goto('/sign-in');
  });

  test('Đăng nhập sai mật khẩu thì báo lỗi', async ({ page }) => {

    // test.step = một Step trong test case. Hiện thành mục riêng trong báo cáo.
    await test.step('Nhập sai mật khẩu và submit', async () => {
      await page.getByLabel('Email').fill('tester@newwave.vn');
      await page.getByLabel('Mật khẩu').fill('SaiRoi@123');
      await page.getByRole('button', { name: 'Đăng nhập' }).click();
    });

    await test.step('Hiện thông báo lỗi, vẫn ở màn hình đăng nhập', async () => {
      await expect(page.getByText('Email hoặc mật khẩu không chính xác')).toBeVisi
      await expect(page).toHaveURL(/\/sign-in/);
    });
  });
});
```

VÌ SAO `TEST.STEP` ĐÁNG DÙNG

Nó làm báo cáo lỗi đọc được bởi người không biết code. Khi test đỏ, thay vì "fail ở dòng 47", báo cáo ghi *"Hiện thông báo lỗi, vẫn ở màn hình đăng nhập — FAILED"*. PM và dev đọc là hiểu ngay. Bạn map thẳng Step trong file TC sang `test.step`.

Quy tắc vàng: mỗi test độc lập

Test số 2 **không được** phụ thuộc vào việc test số 1 đã chạy. Lý do: Playwright chạy nhiều test song song, và thứ tự không đảm bảo. Nếu test "Sửa sản phẩm" cần một sản phẩm tồn tại, thì chính nó phải tự tạo sản phẩm đó trong phần chuẩn bị — chứ không trông chờ test "Tạo sản phẩm" chạy trước.

TỰ LÀM THỬ

1. Tạo file `tests/todo.spec.ts`, chép bài test đầu tiên ở trên vào.
2. Chạy `npx playwright test --ui` và cho nó chạy.
3. Thêm một test thứ hai trong cùng `describe`: thêm 2 việc rồi kiểm chứng bộ đếm hiển thị *"2 items left"*.
4. Bọc mỗi phần bằng `test.step` và xem báo cáo thay đổi thế nào.

PHẦN 04 · CHƯƠNG QUAN TRỌNG NHẤT

Locator — trái tim của automation

80% thời gian viết test là đi tìm locator. 80% test hỏng là do locator sai. Nếu bạn chỉ đọc kỹ một phần trong tài liệu này, hãy đọc phần này.

Trước hết: trang web trong mắt máy tính

Bạn nhìn thấy một nút màu xanh ghi "Đăng nhập". Máy tính nhìn thấy một đoạn văn bản có cấu trúc, gọi là **HTML**:

```
HTML thật của một form đăng nhập

<form>
  <h1>Đăng nhập</h1>
  <label for="email">Email</label>
```

```
  <input id="email" name="email" placeholder="Nhập email" />
  <label for="pwd">Mật khẩu</label>
  <input id="pwd" name="password" type="password" />
  <button type="submit" class="btn btn-primary">Đăng nhập</button>
</form>
```

Mỗi `<input>`, `<button>` là một **element**. Toàn bộ cây element đó gọi là **DOM**. Bấm `F12` trên bất kỳ trang web nào, chọn tab *Elements* — bạn đang nhìn vào DOM.

Locator là gì — và điểm khác biệt quan trọng

Locator là **cách bạn chỉ vào một element**. Nhưng ở Playwright có một điểm đặc biệt phải hiểu ngay:

> LOCATOR LÀ LỜI HỨA, KHÔNG PHẢI KẾT QUẢ
>
> Khi bạn viết `page.getByRole('button', { name: 'Lưu' })`, Playwright **chưa** đi tìm gì cả. Nó chỉ ghi lại "khi nào cần, hãy tìm cái nút tên Lưu". Đến lúc bạn `.click()`, nó mới tìm — và nếu chưa thấy, nó *tìm lại liên tục* cho đến khi thấy hoặc hết giờ.
>
> Đây chính là lý do Playwright ít bị flaky: locator không "chết" khi trang render lại. Bạn có thể khai báo locator ngay từ constructor, trước cả khi trang được mở.

Bảng ưu tiên — dùng theo đúng thứ tự này

Có nhiều cách chỉ vào cùng một nút. Không phải cách nào cũng tốt. Nguyên tắc: **ưu tiên cách mà người dùng nhận ra element**, tránh cách phụ thuộc vào chi tiết kỹ thuật mà dev có thể đổi bất cứ lúc nào.

Ưu tiên	Cách viết	Dùng khi	Độ bền
1	`getByRole('button', { name: 'Lưu' })`	Nút, link, ô nhập, heading, checkbox — gần như mọi thứ bấm được	**Rất cao**
2	`getByLabel('Email')`	Ô nhập trong form có nhãn	**Rất**

2	`getByLabel('Email')`	Ô nhập trong form có nhãn	**Rất cao**
3	`getByPlaceholder('Nhập email')`	Ô nhập không có nhãn, chỉ có chữ mờ bên trong	Cao
4	`getByText('Không có dữ liệu')`	Đoạn chữ, thông báo, nội dung tĩnh	Trung bình
5	`getByTestId('submit-btn')`	Khi dev đã gắn `data-testid`. Bền nhất nếu có.	**Rất cao**
6	`locator('input[name="email"]')`	CSS — khi 5 cách trên không dùng được	Trung bình
7	`locator('//div[3]/span[2]')`	XPath — **gần như luôn có cách tốt hơn**	Rất thấp

> MẸO CHO TESTER: ĐÂY CŨNG LÀ BÀI TEST ACCESSIBILITY MIỄN PHÍ
>
> Nếu bạn *không thể* viết `getByRole('button', { name: 'Xoá' })` vì cái nút đó thật ra chỉ là một `<div>` có icon, thì đó là một **bug thật**: người dùng đọc màn hình bằng trình đọc màn hình sẽ không dùng được nút đó. Hãy log bug, đừng lặng lẽ chuyển sang XPath.

Ba cách tìm locator trong thực tế

Cách 1 — Codegen: bạn bấm, nó viết code

Cách dễ nhất cho người mới. Nó mở một trình duyệt, bạn thao tác bình thường, nó sinh code tương ứng:

```
> npx playwright codegen https://demo.playwright.dev/todomvc
```

Bấm vài thao tác và xem code hiện ra bên cạnh. **Nhưng đừng chép nguyên xi** — codegen đôi khi chọn locator xấu. Hãy coi nó là bản nháp, rồi bạn sửa lại theo bảng ưu tiên ở trên.

Cách 2 — Pick locator trong VS Code / UI Mode

Trong UI Mode (`npx playwright test --ui`) có nút **Pick locator**. Bấm vào, rồi rê chuột lên trang — Playwright tô sáng element và hiện luôn locator đề xuất. Đây là cách bạn sẽ dùng hàng ngày.

Cách 3 — Tự đọc DOM bằng F12

Bấm `F12` → chọn công cụ mũi tên → click vào element. Nhìn HTML của nó và tự quyết định: nó có `role` gì, có `label` không, có `data-testid` không. Đây là kỹ năng bạn cần khi hai cách trên không cho locator tốt.

Khi có nhiều element giống nhau: lọc và nối

Đây là tình huống thực tế nhất: một cái bảng có 20 dòng, mỗi dòng đều có nút "Xoá". Bạn muốn xoá đúng dòng của "Dự án Ecopark".

lọc và nối locator — kỹ thuật quan trọng nhất chương này

```
// SAI: có 20 nút Xoá, Playwright không biết chọn cái nào → báo lỗi strict mode
await page.getByRole('button', { name: 'Xoá' }).click();

// ĐÚNG: tìm đúng DÒNG trước, rồi tìm nút Xoá BÊN TRONG dòng đó
const dong = page.getByRole('row').filter({ hasText: 'Dự án Ecopark' });
await dong.getByRole('button', { name: 'Xoá' }).click();

// filter cũng lọc được theo element con:
page.getByRole('row').filter({ has: page.getByRole('checkbox', { checked: true }) })

// Và loại trừ:
page.getByRole('listitem').filter({ hasNotText: 'Đã huỷ' });
```

Cách đọc: `A.getByRole(...)` nghĩa là "**tìm bên trong A**". Nối như vậy gọi là *chaining*, và nó luôn tốt hơn viết một CSS selector dài loằng ngoằng.

Lỗi "strict mode violation" — bạn chắc chắn sẽ gặp

```
Error: strict mode violation: getByRole('button', { name: 'Xoá' }) ...
```

```
Error: strict mode violation: getByRole('button', { name: 'Xoá' })
resolved to 20 elements:
  1) <button class="btn-del">Xoá</button> aka getByRole('row').first()...
  2) <button class="btn-del">Xoá</button> ...
```

Nghĩa là: *"locator của bạn trúng 20 element, tôi không đoán bừa"*. Đây là **tính năng, không phải lỗi** — nó chặn bạn khỏi việc vô tình bấm nhầm dòng.

✓ CÁCH SỬA ĐÚNG

```
page.getByRole('row')
    .filter({ hasText: 'Ecopark' })
    .getByRole('button', { name: 'Xoá' })
```

Mô tả chính xác element bạn muốn. Dữ liệu đổi thứ tự vẫn đúng.

✗ CÁCH CHỮA CHÁY

```
page.getByRole('button', { name: 'Xoá' })
    .first()
```

Tắt cảnh báo chứ không giải quyết. Hôm nào dòng đầu bảng đổi là xoá nhầm dữ liệu.

Chỉ dùng `.first()`, `.last()`, `.nth(2)` khi *thứ tự chính là điều bạn đang test* — ví dụ "sản phẩm mới nhất phải nằm đầu danh sách".

Locator bền và locator dễ vỡ

✓ BỀN

```
getByRole('button', { name: 'Lưu' })
getByLabel('Số điện thoại')
getByTestId('project-row')
locator('input[name="email"]')
```

Bám vào ý nghĩa và chức năng. Dev đổi màu, đổi khung, đổi vị trí — test vẫn chạy.

✗ DỄ VỠ

```
locator('.css-1x2h9k')
```

```
locator('div > div > div:nth-child(3)')
locator('//*[@id="root"]/div[2]/div[1]/button')
locator('.flex.items-center.px-4.rounded-lg')
```

Bám vào chi tiết trình bày. Chỉ cần dev bọc thêm một thẻ div là hỏng hàng loạt test.

Đọc code locator thật của dự án

Đây là code đang chạy trong `playwright/genesis-e2e` của team bạn — bây giờ bạn đọc được rồi:

src/pages/LoginPage.ts — code thật

```
this.heading        = page.getByRole('heading', { name: 'Đăng nhập' });
this.emailInput     = page.locator('input[name="email"]');
this.passwordInput  = page.locator('input[name="password"]');
this.submitButton   = page.getByRole('button', { name: 'Đăng nhập', exact: true })
this.errorMessage   = page.getByText('Email hoặc mật khẩu không chính xác').first()

// Nút hiện/ẩn mật khẩu không có tên → đi lên thẻ cha rồi tìm nút bên trong
this.togglePasswordButton = this.passwordInput.locator('..').getByRole('button');
```

- `exact: true` ở nút submit: vì trên trang có cả heading "Đăng nhập" lẫn nút "Đăng nhập". Không có `exact` thì `getByRole` khớp cả chuỗi con — thêm nó vào để tránh trúng nhầm.
- `.locator('..')`: dấu hai chấm trong CSS/XPath nghĩa là "đi lên thẻ cha". Dùng khi element bạn cần không tự mô tả được nó là gì, phải mượn element bên cạnh làm mốc.
- `.first()` ở thông báo lỗi: chấp nhận được, vì đây là thông báo lỗi duy nhất về mặt nghiệp vụ — có thể ứng dụng render nó ở 2 chỗ (toast + inline).

TỰ LÀM THỬ — BÀI QUAN TRỌNG NHẤT TRONG TÀI LIỆU

1. Mở `npx playwright codegen https://www.saucedemo.com`.
2. Đăng nhập bằng `standard_user` / `secret_sauce`. Xem code sinh ra.
3. Với *mỗi* locator codegen sinh ra, tự hỏi: nó nằm ở mức mấy trong bảng ưu

tiên? Có viết lại thành `getByRole` hoặc `getByLabel` được không?

4. Thêm sản phẩm vào giỏ. Cố tình viết `getByRole('button', { name: 'Add to cart' })` để gặp lỗi strict mode.
5. Sửa nó bằng `filter({ hasText: ... })` để chọn đúng sản phẩm "Sauce Labs Backpack".

PHẦN 05

Hành động & kiểm chứng

Có locator rồi thì làm gì với nó, và làm sao chấm điểm đúng/sai.

Các hành động thường dùng

Hành động	Code	Ghi chú
Bấm	`await loc.click()`	Có `.dblclick()` và `.click({ button: 'right' })`
Điền ô nhập	`await loc.fill('abc')`	Xoá sạch rồi điền. Nhanh, ổn định — **dùng cái này**.
Gõ từng ký tự	`await loc.pressSequentially('abc')`	Chỉ dùng khi ô có gợi ý tự động cần kích hoạt theo từng phím.
Xoá ô	`await loc.clear()`	
Tick checkbox	`await loc.check()` / `.uncheck()`	Tự biết trạng thái hiện tại, không bấm nhầm ngược.
Chọn dropdown	`await loc.selectOption('HN')`	Chỉ cho thẻ `<select>` gốc. Dropdown tự vẽ thì phải click như bình thường.

Rê chuột	`await loc.hover()`	Để hiện menu hoặc tooltip.
Bấm phím	`await loc.press('Enter')`	`'Tab'`, `'Escape'`, `'Control+A'` …
Upload file	`await loc.setInputFiles('hd.pdf')`	Không cần mở hộp thoại chọn file.
Lấy chữ ra	`const t = await loc.textContent()`	Có `await` vì phải hỏi trình duyệt.

Auto-waiting: vì sao bạn không cần lệnh chờ

Trước mỗi hành động, Playwright tự kiểm tra element đã *sẵn sàng* chưa: đã xuất hiện trong DOM, đang hiển thị, không bị element khác che, đã ngừng chuyển động, và đang bật (không bị disabled). Nếu chưa, nó chờ và thử lại — mặc định tới 15–30 giây tuỳ cấu hình.

ĐỪNG BAO GIỜ DÙNG LỆNH NGỦ

`await page.waitForTimeout(3000)` là dấu hiệu của một bộ test sắp mục ruỗng. Máy nhanh thì bạn phí 3 giây mỗi lần; máy chậm thì 3 giây vẫn không đủ và test vẫn đỏ. Bạn vừa làm test chậm hơn *vừa* không hết flaky.

✗ CHỜ MÙ

```
await page.click('#luu');
await page.waitForTimeout(3000);
const t = await page.textContent('.msg');
expect(t).toBe('Lưu thành công');
```

Chờ cứng 3 giây rồi chụp một khoảnh khắc. Chậm hơn cần thiết mà vẫn có ngày đỏ.

✓ CHỜ THEO ĐIỀU KIỆN

```
await page.getByRole('button', { name: 'Lưu' }).click();
await expect(page.getByText('Lưu thành công'))
  .toBeVisible();
```

Hiện sau 0,2 giây thì đi tiếp ngay; sau 8 giây vẫn bắt được. Nhanh hơn và chắc hơn.

Web-first assertion — và cái bẫy lớn

`await expect(locator).toBeVisible()` có một siêu năng lực: nó **thử lại liên tục** cho tới khi đúng hoặc hết giờ. Nhưng chỉ khi bạn viết đúng dạng:

cái bẫy

```
// ✓ ĐÚNG — locator nằm TRONG expect. Có retry.
await expect(page.getByText('Thành công')).toBeVisible();

// ✗ SAI — chụp giá trị TRƯỚC rồi mới expect. Không retry. Flaky.
const hienRa = await page.getByText('Thành công').isVisible();
expect(hienRa).toBe(true);
```

Câu thần chú: **để locator vào bên trong `expect`, đừng lấy giá trị ra trước.**

Các assertion hay dùng

Muốn kiểm chứng	Code
Element hiện ra / biến mất	`toBeVisible()` / `toBeHidden()`
Nút bật / mờ	`toBeEnabled()` / `toBeDisabled()`
Chữ chính xác / chứa chữ	`toHaveText('Xong')` / `toContainText('Xong')`
Giá trị trong ô nhập	`toHaveValue('a@b.com')`
Ô trống	`toBeEmpty()`
Checkbox đang tick	`toBeChecked()`
Số lượng element	`expect(rows).toHaveCount(20)`
URL / tiêu đề trang	`expect(page).toHaveURL(/\/don-hang/)` · `toHaveTitle(/Genesis/)`

Phủ định bất kỳ cái nào	thêm `.not` : `expect(loc).not.toBeVisible()`

KIỂM CHỨNG CHO ĐỦ

Một cái bẫy nghề nghiệp: chỉ kiểm tra "màn hình chuyển sang trang B" mà quên kiểm tra dữ liệu có đúng không. Test case của bạn ghi Expected Result thế nào thì code phải kiểm đủ thế ấy — chuyển trang *và* đúng tên khách hàng *và* đúng số tiền.

TỰ LÀM THỬ

1. Trên saucedemo, viết test: đăng nhập → thêm 2 sản phẩm → mở giỏ hàng.
2. Kiểm chứng: badge giỏ hàng hiện số `2` , giỏ có đúng `toHaveCount(2)` dòng, và tên sản phẩm khớp.
3. Viết thêm test đăng nhập bằng `locked_out_user` và kiểm chứng thông báo lỗi hiện ra.

PHẦN 06

Debug khi test đỏ

Test đỏ là chuyện bình thường hàng ngày. Kỹ năng cần có là đọc được nó đỏ vì đâu — bug thật, hay code test sai.

Bốn công cụ, theo thứ tự nên dùng

1. UI Mode — mặc định của bạn

```
> npx playwright test --ui
```

Cho bạn: danh sách test, xem lại từng bước sau khi chạy, tua ngược thời gian để xem trang trông thế nào ở mỗi bước, và **Pick locator**. Người mới nên dành 90% thời gian ở đây.

2. Xem test chạy bằng mắt thường

```
> npx playwright test --headed              # mở trình duyệt thật cho bạn nhìn
> npx playwright test dang-nhap --debug     # chạy từng bước một, có nút Next
```

3. Trace Viewer — khi test đỏ trên CI mà máy bạn chạy lại xanh

Trace là bản ghi đầy đủ: từng bước, ảnh chụp DOM tại mỗi bước, network, console log. Trong `playwright.config.ts` của dự án đã bật sẵn `trace: 'retain-on-failure'` — nghĩa là test nào đỏ sẽ tự lưu lại.

```
> npx playwright show-trace test-results/dang-nhap/trace.zip
```

4. Báo cáo HTML — thứ bạn gửi cho dev

```
> npx playwright show-report
```

Có ảnh chụp màn hình lúc lỗi và video. Đây chính là bằng chứng để đính vào bug report — đỡ hẳn công quay màn hình bằng tay.

Đọc thông báo lỗi

Lỗi bạn thấy	Nghĩa thật là	Làm gì
`Timeout 15000ms exceeded waiting for locator(...)`	Không tìm thấy element trong 15 giây	Locator sai, HOẶC trang thật sự không hiện element đó → có thể là **bug thật**. Xem trace để biết.
`strict mode violation: resolved to N`	Locator trúng nhiều element	Thu hẹp bằng `filter()` hoặc chaining. Xem lại Phần 04.

`elements`		
`element is not visible`	Tìm thấy trong DOM nhưng đang bị ẩn	Có thể chưa mở tab/accordion chứa nó, hoặc thiếu một bước trước đó.
`element is outside of the viewport`	Nằm ngoài màn hình	Thường Playwright tự cuộn. Nếu vẫn lỗi, kiểm tra xem có popup/overlay che không.
`Expected: "5" Received: "4"`	Kiểm chứng sai giá trị	Đây thường là **bug thật**. Chụp lại và log defect.
`Target page, context or browser has been closed`	Trang bị đóng giữa chừng	Gần như luôn là thiếu `await` ở đâu đó phía trên.

> CÂU HỎI PHẢI TRẢ LỜI MỖI KHI TEST ĐỎ
>
> **"Đây là bug của sản phẩm, hay là lỗi của code test?"** Đừng bao giờ sửa code test cho xanh trước khi trả lời được câu này. Sửa test để né một bug thật là cách nhanh nhất biến bộ automation thành đồ trang trí.

PHẦN 07

Xây framework

Đến đây bạn viết được test đơn lẻ. Framework là thứ giúp 200 bài test không biến thành 200 mớ hỗn độn. Phần này dùng chính code trong `playwright/genesis-e2e` của dự án làm ví dụ.

Vấn đề mà framework giải quyết

Giả sử bạn có 30 bài test, bài nào cũng phải đăng nhập, nên bài nào cũng có dòng

nay:

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

Một hôm dev đổi `name="email"` thành `name="username"`. Bạn phải sửa 30 file. Tuần sau đổi cái khác — lại 30 file. Bộ test chết dần vì không ai buồn sửa.

Page Object Model: mỗi màn hình một file

Ý tưởng rất đơn giản: **gom hết locator và hành động của một màn hình vào một class.** Ai cần thao tác với màn hình đó thì gọi class. Dev đổi locator → sửa đúng một chỗ.

✗ KHÔNG CÓ POM

```
test('đăng nhập', async ({ page }) => {
  await page.goto('/sign-in');
  await page.locator('input[name="email"]')
            .fill('a@b.com');
  await page.locator('input[name="password"]')
            .fill('123456');
  await page.locator('.btn-primary').click();
});
```

Đọc không biết đang test nghiệp vụ gì. Locator lặp ở mọi file.

✓ CÓ POM

```
test('đăng nhập', async ({ loginPage }) => {
  await loginPage.goto();
  await loginPage.login('a@b.com', '123456');
  await loginPage.expectLoaded();
});
```

Đọc như test case. PM cũng hiểu. Locator nằm gọn một nơi.

Cấu trúc thư mục — lấy từ dự án thật

```
playwright/genesis-e2e/                                    cấu trúc thật của team
```

```
genesis-e2e/
├── src/
│   ├── config/paths.ts          ← đường dẫn dùng chung
│   ├── fixtures/test-fixtures.ts ← "giao hàng tận nơi" page object cho test
│   └── pages/
│       ├── BasePage.ts          ← phần chung của mọi màn hình
│       ├── LoginPage.ts         ← 1 màn hình = 1 file
│       ├── UserManagementPage.ts
│       ├── OpsHeaderComponent.ts ← component dùng lại ở nhiều màn hình
│       └── RoleSwitcherComponent.ts
├── tests/
│   ├── auth.setup.ts            ← đăng nhập 1 lần, lưu session lại
│   ├── login.spec.ts            ← 1 chức năng = 1 file spec
│   ├── logout.spec.ts
│   └── ops/session-reuse.spec.ts
├── .env.test                    ← URL + tài khoản, KHÔNG commit lên git
└── playwright.config.ts
```

BasePage — phần chung của mọi màn hình

Mọi màn hình đều cần: biết URL của mình, biết khi nào mình đã load xong. Viết một lần ở đây, các màn hình khác kế thừa (`extends`).

src/pages/BasePage.ts — code thật, rút gọn

```ts
export abstract class BasePage {
  protected constructor(protected readonly page: Page) {}

  // abstract = "lớp con BẮT BUỘC phải khai báo cái này"
  abstract readonly url: string;
  protected abstract get readyLocator(): Locator;

  async goto(): Promise<void> {
    await this.page.goto(this.url, { waitUntil: 'domcontentloaded' });
  }

  async waitUntilLoaded(): Promise<void> {
    await expect(this.readyLocator).toBeVisible();
  }
}
```

`readyLocator` là một ý hay đáng học: mỗi màn hình tự khai báo *"element nào chứng minh tôi đã render xong"*. Với màn Đăng nhập thì đó là cái heading "Đăng nhập". Nhờ vậy không màn hình nào cần lệnh chờ thủ công.

Một Page Object hoàn chỉnh

src/pages/LoginPage.ts — code thật, rút gọn

```
export class LoginPage extends BasePage {
  readonly url = `${process.env.URL_ID}/sign-in/`;

  // 1) Khai báo element theo notation Screen > Element
  readonly emailInput: Locator;
  readonly passwordInput: Locator;
  readonly submitButton: Locator;

  constructor(page: Page) {
    super(page);
    this.emailInput    = page.locator('input[name="email"]');
    this.passwordInput = page.locator('input[name="password"]');
    this.submitButton  = page.getByRole('button', { name: 'Đăng nhập', exact: true
  }

  // 2) Hành động nghiệp vụ — KHÔNG kiểm chứng kết quả
  async login(email: string, password: string): Promise<void> {
    await this.emailInput.fill(email);
    await this.passwordInput.fill(password);
    await this.submitButton.click();
  }

  // 3) Kiểm chứng về TRẠNG THÁI MÀN HÌNH thì được phép để ở đây
  async expectLoaded(): Promise<void> {
    await expect(this.emailInput).toBeVisible();
    await expect(this.submitButton).toBeEnabled();
  }
}
```

> NGUYÊN TẮC VÀNG CỦA POM

> Hàm `login()` ở trên **cố tình không** kiểm chứng xem đăng nhập có thành công hay không. Comment trong code thật của dự án ghi rõ điều đó: *"Does NOT assert the outcome — the test decides"*.
>
> Lý do: cùng một hành động `login()` được dùng cho cả test "đăng nhập đúng thì vào trang chủ" lẫn test "đăng nhập sai thì báo lỗi". Nếu Page Object tự khẳng định kết quả, bạn không viết được case âm. **Page Object mô tả màn hình làm được gì; spec quyết định kết quả mong đợi.**

Fixtures — để test khỏi phải tự khởi tạo page object

Không có fixture, mỗi test phải viết `const loginPage = new LoginPage(page);`. Fixture khai báo một lần, sau đó test chỉ cần "gọi tên" là có:

src/fixtures/test-fixtures.ts — ý tưởng

```
export const test = base.extend<{ loginPage: LoginPage }>({
  loginPage: async ({ page }, use) => {
    await use(new LoginPage(page));   // tạo sẵn, giao cho test
  },
});
```

tests/login.spec.ts — và test chỉ việc nhận

```
import { test, expect } from '../src/fixtures/test-fixtures';

test('Đăng nhập thành công', async ({ loginPage, userManagementPage, validUser }) =
  await loginPage.login(validUser.email, validUser.password);
  await userManagementPage.expectLoaded();
});
```

Để ý dòng `import`: nó lấy `test` từ file fixture của dự án, **không** lấy từ `@playwright/test`. Đây là điểm người mới hay nhầm khi thêm file spec mới.

playwright.config.ts — bảng điều khiển

playwright.config.ts — các mục quan trọng

playwright.config.ts — các mục quan trọng

```ts
export default defineConfig({
  testDir: './tests',
  fullyParallel: true,                // chạy song song cho nhanh
  retries: process.env.CI ? 2 : 0,    // trên CI, đỏ thì thử lại 2 lần
  timeout: 60_000,                    // 1 test tối đa 60 giây
  expect: { timeout: 15_000 },        // 1 assertion chờ tối đa 15 giây

  use: {
    baseURL: process.env.URL_ID,      // để goto('/sign-in') là đủ
    trace: 'retain-on-failure',       // đỏ mới lưu trace
    screenshot: 'only-on-failure',
    video: 'retain-on-failure',
  },
});
```

VỀ RETRIES

Chạy lại khi đỏ giúp CI đỡ ồn, nhưng nó cũng *giấu* test flaky. Nếu một test cứ phải chạy lần 2 mới xanh, hãy coi đó là nợ kỹ thuật cần sửa, đừng để yên.

Dữ liệu và môi trường: đừng viết cứng vào code

.env.test — không commit lên git

```
URL_ID=https://id-test.genesis.vn
URL_OPS=https://ops-test.genesis.vn
USER_EMAIL=tester@newwave.vn
USER_PASSWORD=...
```

Trong code đọc bằng `process.env.URL_ID`. Nhờ vậy cùng một bộ test chạy được trên TEST và STAGING chỉ bằng cách đổi file env. Nhớ thêm `.env.*` vào `.gitignore`.

QUY TẮC CỦA DỰ ÁN

Dữ liệu test luôn là dữ liệu giả. **Không bao giờ** copy dữ liệu khách hàng thật vào test — kể cả tên, số điện thoại, CCCD. Đây là quy định đã ghi trong `qa-flow-rules.md` của team.

`flow-notes.md` của team.

Đăng nhập một lần, dùng lại cho mọi test

Nếu 50 test đều phải đăng nhập lại từ đầu, bạn mất vài phút chỉ để gõ mật khẩu. Cách làm chuẩn: đăng nhập *một lần* ở bước setup, lưu cookie ra file, các test sau nạp thẳng cookie đó.

Trong dự án Genesis, file `tests/auth.setup.ts` làm việc này và ghi ra `.auth/user.json`. Nhưng có một chi tiết đáng học — comment thật trong config của team:

playwright.config.ts — ràng buộc thật của dự án

```
/* Thứ tự project là cố ý: auth-flows → setup → ops.
 * Tài khoản test Genesis chỉ giữ MỘT phiên đăng nhập — mỗi lần login
 * hay đổi vai trò đều làm cookie cũ hết hiệu lực. Nên session cache
 * phải được tạo SAU tất cả những test có đăng nhập. */
```

Đây là bài học framework quan trọng nhất và không sách nào dạy: **ràng buộc của hệ thống thật quyết định kiến trúc test.** Bạn phải hiểu nghiệp vụ mới thiết kế được framework đúng — và đó chính là thứ bạn, với vai trò QA, mạnh hơn dev.

Checklist một framework khoẻ mạnh

- Mỗi màn hình một page object; locator **không** xuất hiện trong file spec.
- Page object chứa hành động; spec chứa kiểm chứng nghiệp vụ.
- Không có `waitForTimeout` ở bất kỳ đâu.
- Không có URL, email, mật khẩu viết cứng — tất cả qua `.env`.
- Mỗi test tự chuẩn bị dữ liệu của mình, chạy độc lập, chạy được ở bất kỳ thứ tự nào.
- Tên test đọc như tên test case, người không biết code cũng hiểu.
- Chạy lại 5 lần liên tiếp vẫn xanh cả 5.

TỰ LÀM THỬ

1. Quay lại bộ test saucedemo của bạn. Tạo thư mục `src/pages/`.
2. Viết `LoginPage.ts`: chứa 3 locator (email, password, nút Login) và hàm `login()`.
3. Viết `ProductsPage.ts`: có hàm `addToCart(tenSanPham)` dùng `filter()`.
4. Sửa lại các spec cũ để dùng 2 page object này. So sánh độ dễ đọc trước/sau.
5. Đưa URL và tài khoản ra file `.env`.

PHẦN 08

Lộ trình 6 tuần

Mỗi tuần khoảng 4–5 giờ. Đây là một chuỗi có thứ tự — tuần sau dựa trên kết quả tuần trước, đừng nhảy cóc.

TUẦN 1

Chạy được, đọc được

Phần 01 + 02 + 03. Cài đặt, tạo project, đọc hiểu một bài test. Chưa cần tự viết từ đầu — mục tiêu là chép, sửa, chạy, và không sợ terminal nữa.

Kết quả: sửa được test mẫu và giải thích được từng dòng.

TUẦN 2

Test đầu tay trên trang thật

Dùng demo.playwright.dev/todomvc. Viết 5 test: thêm việc, sửa việc, xoá việc, tick hoàn thành, lọc danh sách. Dùng codegen làm nháp rồi viết lại bằng tay.

Kết quả: 5 test xanh, tự viết, không copy.

TUẦN 3

Cày locator

Đọc lại Phần 04 thật kỹ. Chuyển sang saucedemo.com — có bảng, dropdown, giỏ hàng. Ép mình dùng `getByRole` trước, chỉ xuống CSS khi thật sự bí. Cố tình gây lỗi strict mode rồi sửa.

Kết quả: chọn đúng 1 dòng trong bảng nhiều dòng bằng `filter()`.

TUẦN 4

Page Object

Phần 07. Tái cấu trúc toàn bộ test saucedemo sang POM. Thêm fixture. Đưa dữ liệu ra `.env`. Đây là tuần biến bạn từ "người viết script" thành "người xây framework".

Kết quả: file spec không còn một locator nào.

TUẦN 5

Đọc code của team

Mở `playwright/genesis-e2e`. Đọc `BasePage.ts`, `LoginPage.ts`, `login.spec.ts`, rồi `playwright.config.ts`. Chạy suite. Chưa sửa gì — chỉ đọc và ghi lại câu hỏi.

Kết quả: một danh sách câu hỏi để hỏi người viết suite.

TUẦN 6

Đóng góp thật

Chọn *một* test case manual đơn giản của Genesis mà bạn đã viết, và tự động hoá nó: thêm page object nếu cần, thêm spec, chạy xanh, nhờ review.

Kết quả: bài test đầu tiên của bạn nằm trong suite của dự án.

BA LỜI KHUYÊN VỀ CÁCH HỌC

- **Gõ, đừng copy.** Gõ tay chậm hơn nhưng tay bạn nhớ cú pháp. Copy-paste thì tuần sau vẫn phải tra lại.
- **Cố tình làm hỏng.** Xoá một chữ `await` và xem chuyện gì xảy ra. Học từ lỗi

nhanh hơn học từ code chạy đúng.

- **Mỗi buổi một mục tiêu nhỏ, chạy được.** "Hôm nay chọn đúng dòng trong bảng" tốt hơn "hôm nay học Playwright".

PHẦN 09

Tra cứu nhanh

Phần để mở lại khi đang làm việc.

Từ điển thuật ngữ

`DOM`	Cây các thẻ HTML tạo nên trang web. Bấm F12 → tab Elements để xem.
`Element`	Một thành phần trên trang: một nút, một ô nhập, một dòng bảng.
`Locator`	Cách chỉ vào một element. Ở Playwright nó là "lời hứa tìm", tìm lại mỗi lần dùng.
`Selector`	Chuỗi mô tả element theo cú pháp CSS hoặc XPath, ví dụ `input[name="email"]`.
`Assertion`	Câu kiểm chứng — phần "Expected Result" trong code. Viết bằng `expect`.
`Spec`	File chứa các bài test, đuôi `.spec.ts`.
`Suite`	Toàn bộ tập hợp test của dự án.

`Flaky`	Test lúc xanh lúc đỏ dù code không đổi. Kẻ thù số một.
`Headless / Headed`	Chạy ẩn không hiện trình duyệt / chạy hiện trình duyệt cho bạn nhìn.
`Fixture`	Thứ Playwright chuẩn bị sẵn và "giao tận tay" cho test, ví dụ `page` hay `loginPage`.
`POM`	Page Object Model — mỗi màn hình gói thành một class.
`Trace`	Bản ghi đầy đủ một lần chạy test, xem lại được từng bước như tua video.
`CI`	Máy chủ tự động chạy test mỗi khi có code mới được đẩy lên.
`storageState`	File lưu cookie/session để test sau khỏi phải đăng nhập lại.

Lệnh hay dùng

Locator — bản rút gọn