30 Câu Hỏi Phỏng Vấn Playwright — Phiên Bản Thực Chiến

Mỗi câu gồm: ngữ cảnh interviewer hỏi, câu trả lời ở mức ứng viên có kinh nghiệm, và bẫy/follow-up thường gặp.

Câu 1. Tại sao team em chọn Playwright thay vì Cypress hoặc Selenium?

Vì sao họ hỏi: Lọc ứng viên "biết dùng" và ứng viên "hiểu tại sao dùng".

Trả lời: Em không nói Playwright "tốt hơn" — em nói nó phù hợp hơn với bài toán của team:
- **Cross-browser thật sự:** Cypress chạy trên một bản Chromium đã được vá, Playwright chạy trực tiếp WebKit, Firefox, Chromium qua CDP. Sản phẩm em có user dùng Safari trên iPhone nên đây là yêu cầu cứng.
- **Auto-wait + Web-first assertions:** `expect(locator).toBeVisible()` tự retry trong timeout — giảm ~80% flaky test so với Selenium em từng dùng.
- **Tốc độ + cách ly:** `BrowserContext` rẻ hơn việc bật/tắt browser. Suite ~400 case chạy parallel xong trong ~4 phút trên CI.

Follow-up hay gặp: *"Khi nào em không chọn Playwright?"* → Khi cần test trên Safari thật trên iOS device, hoặc project legacy IE11.

Câu 2. Khi nào em thực sự cần `waitFor`, và khi nào auto-wait là đủ?

Vì sao họ hỏi: Junior thường await mọi thứ kèm `waitForTimeout(3000)`. Câu này lọc người mới.

Trả lời: Mặc định em KHÔNG dùng `waitFor`. Các action (`click`, `fill`) và assertion (`toBeVisible`) đã có auto-wait. Em chỉ dùng `waitFor` trong 3 trường hợp:
- **Chờ phần tử biến mất:** `await page.locator('.spinner').waitFor({ state: 'hidden' })` trước khi click tiếp.
- **Chờ điều kiện không gắn với locator:** Ví dụ chờ biến global trên `window` — `page.waitForFunction(() => window.appReady === true)`.
- **Chờ network response cụ thể:** `page.waitForResponse(resp =>`

`resp.url().includes('/api/orders') && resp.status() === 200)` — hữu ích sau submit form mà UI chưa đổi rõ.

Quy tắc cứng: nếu phải dùng `waitForTimeout(N)`, 99% là code đang giấu một race condition. Em sẽ tìm điều kiện thực sự cần chờ thay vì chờ mò.

Câu 3. Test chạy song song có ca thất bại lẻ tẻ. Em debug thế nào?

Vì sao họ hỏi: Flaky test giết niềm tin vào automation nhanh hơn bất cứ thứ gì.

Trả lời: Em làm theo quy trình, không đoán:
1. **Thu thập bằng chứng:** Bật `trace: 'on-first-retry'`. Lần fail kế tiếp có `trace.zip` — mở bằng `npx playwright show-trace` để xem action, network, console.
2. **Lặp lại lỗi:** `npx playwright test file.spec.ts --repeat-each=20 --workers=4`. Fail 2-3/20 lần là flaky thật.
3. **Phân loại nguyên nhân:** 90% flaky đến từ một trong ba thứ:
- Test phụ thuộc data của test khác → dùng fixture tạo data riêng.
- Chờ UI thay vì chờ data → dùng `waitForResponse`.
- Animation/CSS transition → tắt bằng `page.addStyleTag({ content: '* { transition: none !important; animation: none !important; }' })`.

Em tuyệt đối tránh "giải pháp" thêm `retries: 3` rồi xong. Nó che lỗi chứ không sửa lỗi.

Câu 4. Giải thích Browser, BrowserContext, Page bằng ví dụ thực tế.

Vì sao họ hỏi: Hiểu mô hình này quyết định em viết test có scale được không.

Trả lời: Em hay giải thích bằng tình huống test e-commerce có Buyer và Seller chat với nhau:
- **Browser:** Process Chromium/WebKit thật. Một browser dùng chung cho cả suite — bật/tắt tốn 1-2 giây.
- **BrowserContext:** Như profile incognito. Cookies, localStorage, permissions riêng. Em tạo `contextBuyer` và `contextSeller` — mỗi cái tự đăng nhập user khác nhau, cách ly hoàn toàn. Bật tốn vài ms.
- **Page:** Một tab bên trong context. Buyer có thể có nhiều page trong cùng context.

```
const contextBuyer = await browser.newContext({ storageState: 'buyer.json' });
const contextSeller = await browser.newContext({ storageState: 'seller.json' });
const pageBuyer = await contextBuyer.newPage();
const pageSeller = await contextSeller.newPage();
// pageBuyer gửi tin nhắn → pageSeller assert nhận được
```

Hệ quả: mỗi test nên có context riêng (mặc định Playwright Test đã làm vậy qua fixture `page`). Đừng tái dùng context — sẽ kế thừa state rác.

Câu 5. Khác biệt giữa Locator và ElementHandle — tại sao em không dùng ElementHandle?

Vì sao họ hỏi: Người từ Selenium hay phản xạ "lấy element rồi click" — đó là cách viết flaky nhất trong Playwright.

Trả lời: Locator là "mô tả cách tìm phần tử" — Playwright sẽ tìm tại thời điểm em call action. ElementHandle là "con trỏ trực tiếp đến phần tử đã tìm thấy ngay lúc đó".

Hệ quả: trong SPA hiện đại, React/Vue re-render liên tục → ElementHandle stale ngay. Locator thì không — re-query mỗi lần action, tự retry trong timeout.

Trong 3 năm em chưa gặp tình huống nào BẮT BUỘC dùng ElementHandle. `page.$` và `page.$$` là legacy API team em ban trong ESLint config.

"Nhưng tôi cần element để truyền vào function khác?" → Truyền Locator. Nó cũng có `textContent()`, `boundingBox()`, `evaluate()` — và an toàn hơn.

Câu 6. Selector strategy của em là gì? Khi nào em phá quy tắc đó?

Vì sao họ hỏi: Selector chiếm 50% công sức maintain test suite.

Trả lời: Thứ tự ưu tiên, viết vào ESLint custom rule:
1. `getByRole` **+ name:** Bền nhất, ép code đi qua accessibility. Nếu test không tìm được thì thường là vấn đề a11y đáng fix.
2. `getByLabel` / `getByPlaceholder` / `getByText`**:** Cho form và content tĩnh.
3. `getByTestId`**:** Khi DOM không có semantic rõ (ví dụ `<div>` làm icon button). Team quy ước `data-testid="kebab-case-meaningful"`.
4. **CSS selector:** Chỉ khi 3 cái trên không khả thi — phải là class bền (không phải class hash

Tailwind/CSS Module).
5. **XPath:** Gần như không bao giờ.

Khi nào phá quy tắc: Khi label/role thay đổi theo locale. App tiếng Việt, em ưu tiên `testid` để không phải maintain text dịch — hoặc dùng key i18n thay vì text hiển thị.

Câu 7. Em login thế nào trong 300 test case mà không tốn 300 lần đăng nhập?

Trả lời: Em dùng `storageState` theo mô hình project dependency:
1. **Project "setup":** Chạy `global.setup.ts` một lần, login qua UI hoặc tốt hơn là gọi `POST /auth/login` lấy token, sau đó `page.context().storageState({ path: '.auth/user.json' })`.
2. **Project test thật:** Khai báo `dependencies: ['setup']` và `use: { storageState: '.auth/user.json' }`. Mỗi test mở context đã có sẵn cookie/token.

Mở rộng thực tế:
- **Multi-role:** Tạo nhiều file `admin.json`, `manager.json`, `user.json` — mỗi project Playwright dùng một `storageState`. Test admin và user chạy song song không conflict.
- **Test login flow thật:** Login flow vẫn cần test. Dùng `test.use({ storageState: { cookies: [], origins: [] } })` cho file đó để vô hiệu hóa storageState toàn cục.

Câu 8. Em làm gì khi backend chưa sẵn sàng nhưng UI cần test ngay?

Trả lời: Em dùng `page.route()` theo 3 tầng độ thật:

Tầng 1 — Mock thuần: Đọc OpenAPI spec, viết route handler trả về fixture JSON.

```
await page.route('**/api/orders', async (route) => {
  await route.fulfill({ status: 200, json: ordersFixture });
});
```

Tầng 2 — Modify response thật: API có rồi nhưng cần test edge case (100 items, lỗi 500). Không mock toàn bộ, fetch response thật rồi sửa.

```
await page.route('**/api/orders', async (route) => {
  const response = await route.fetch();
  const data = await response.json();
  data.items = generateBigList(100); // chỉ sửa phần cần test
  await route.fulfill({ response, json: data });
});
```

Tầng 3 — Mock lỗi mạng: `await route.abort('failed')` để test UI có hiển thị error state đúng không.

Quan trọng: Em không mock 100% trong regression suite. Chỉ mock có chọn lọc cho edge case khó tái hiện. Regression vẫn phải hit BE thật trên staging.

Câu 9. API test bằng Playwright — bao giờ hợp lý, bao giờ không?

Trả lời: Em dùng `request` fixture trong 3 tình huống:
- **Setup data:** Trước test "user xem đơn hàng", gọi `POST /api/orders` tạo data — nhanh hơn click qua UI.
- **Cleanup:** Test xong xóa data đã tạo qua DELETE endpoint.
- **Hybrid test:** Tạo order qua UI, verify qua `GET /api/orders/{id}` bằng admin token.

Em KHÔNG dùng Playwright cho API contract test thuần — đó là việc của Postman/Newman, RestAssured, hay Pact. Playwright thiếu tooling cho schema validation, contract testing. Trộn chung làm suite chậm và rối.

Câu 10. Page Object Model — em vẫn dùng không, hay có cách nào tốt hơn?

Vì sao họ hỏi: POM là tín ngưỡng cũ. Docs Playwright thậm chí không khuyến nghị nó nữa.

Trả lời: Em không dùng POM truyền thống (class với field locators + một đống method). Em dùng 2 pattern hợp với Playwright hơn:

1. Fixture-based: Mỗi page là một fixture trả về object có các action chính.

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

test('login', async ({ loginPage }) => {
  await loginPage.loginAs('admin');
});
```

2. Component object: Thay vì một page object khổng lồ, chia theo component (`HeaderComponent`, `OrderTableComponent`). Khớp với cách FE chia component.

Quy tắc cứng: Page object KHÔNG chứa assertion. Assertion thuộc về test. Page object chỉ làm action và trả về data — như vậy dùng được cho cả test "happy path" và "error case".

Câu 11. Upload file 500MB hoặc từ drag-drop zone, em xử lý sao?

Trả lời:

Trường hợp thường: `setInputFiles` trên `<input type="file">` — kể cả khi input bị ẩn (`display: none`).

Drag-drop zone không có input thật: Dùng `filechooser` event.

```ts
const fileChooserPromise = page.waitForEvent('filechooser');
await page.getByText('Drop files here').click();
const fileChooser = await fileChooserPromise;
await fileChooser.setFiles('fixture/photo.jpg');
```

File rất lớn (500MB): KHÔNG upload file thật trong E2E — chậm và tốn storage CI. Cách thực tế: (1) test logic UI với file nhỏ (1KB); (2) test validate "file quá lớn" bằng cách mock response từ BE.

Upload buffer (không cần file thật):

```ts
await page.locator('input[type="file"]').setInputFiles({
  name: 'data.csv',
  mimeType: 'text/csv',
  buffer: Buffer.from('a,b,c\n1,2,3'),
});
```

Câu 12. Test download một file rồi verify nội dung — full chain ra sao?

Trả lời: Chia 3 bước rõ ràng:

1. **Đăng ký listener TRƯỚC khi click** (nếu đăng ký sau click sẽ miss event).
2. **Lấy Download object, chưa vội save.**
3. **Save vào path tạm rồi assert content.**

```
const downloadPromise = page.waitForEvent('download');
await page.getByRole('button', { name: 'Export CSV' }).click();
const download = await downloadPromise;

expect(download.suggestedFilename()).toMatch(/^orders-\d{4}-\d{2}-\d{2}\.csv$/);

const path = await download.path();
const content = fs.readFileSync(path!, 'utf-8');
expect(content).toContain('Order ID,Customer,Total');
expect(content.split('\n').length).toBeGreaterThan(10);
```

Bẫy thường gặp: Trong CI headless, một số browser preview PDF thay vì download. Đảm bảo button có thuộc tính `download` hoặc BE trả `Content-Disposition: attachment`.

Câu 13. Popup OAuth (Google login) mở ra tab mới — em handle thế nào?

Trả lời: Dùng `context.waitForEvent('page')` thay vì `page.waitForEvent('popup')` khi tab mới không phải direct popup:

```
const newPagePromise = context.waitForEvent('page');
await page.getByRole('button', { name: 'Sign in with Google' }).click();
const popup = await newPagePromise;
await popup.waitForLoadState();

await popup.locator('input[type="email"]').fill('test@example.com');
await popup.getByRole('button', { name: 'Next' }).click();
// ...

// Popup tự đóng, quay về page chính
await expect(page.getByText('Welcome')).toBeVisible();
```

Quan điểm thực tế: Em hiếm khi test OAuth thật trong E2E. Google/GitHub có CAPTCHA và rate-limit. Em test OAuth flow của app em (callback handling, token storage) bằng cách mock IdP — dùng `storageState` có sẵn token, hoặc dùng test account staging đã được BE whitelist.

Câu 14. iFrame third-party (Stripe, reCAPTCHA, YouTube) — em test thế nào?

Trả lời: Phân biệt 2 loại:

iFrame của chính app: Dùng `page.frameLocator()` bình thường.

```
const stripeFrame = page.frameLocator('iframe[name="__privateStripeFrame"]');
await stripeFrame.getByPlaceholder('Card number').fill('4242 4242 4242 4242');
```

iFrame third-party (reCAPTCHA): KHÔNG cố test. Đây là dịch vụ chống bot — bot test cũng bị chặn. Cách thực tế: (1) BE expose endpoint test-mode bỏ qua captcha cho staging; (2) hoặc dùng test key reCAPTCHA của Google luôn pass.

Payment thật: Stripe có test card `4242...` — verify form submit và webhook xử lý, không cần thanh toán thật.

Bẫy iframe: `frameLocator` hoạt động lazy — tìm frame khi call action, không phải khi gọi `frameLocator`. Nếu iframe load chậm, chờ bằng assertion thay vì `waitForTimeout`.

Câu 15. Visual regression test — em xử lý ảnh khác nhau giữa Mac/Linux thế nào?

Trả lời: Em dùng `toHaveScreenshot` và follow 3 nguyên tắc:

1. Baseline phải sinh trong môi trường giống CI: Mac và Linux render font khác nhau (anti-aliasing). Em không commit baseline từ máy local. Em chạy `--update-snapshots` trong Docker image của CI, hoặc dùng `mcr.microsoft.com/playwright`.

2. Mask phần dynamic:

```
await expect(page).toHaveScreenshot('dashboard.png', {
  mask: [page.locator('.timestamp'), page.locator('.user-avatar')],
  maxDiffPixelRatio: 0.01, // cho phép 1% pixel khác (anti-aliasing)
});
```

3. Tắt animation: `animations: 'disabled'` trong config.

Em chỉ visual test cho component quan trọng (landing page, design system) — không phải mọi page. Visual test chậm và baseline đổi design lại phải review — phải có quy trình review baseline trong PR.

Câu 16. Test pass local nhưng fail trên CI — em check theo thứ tự nào?

Trả lời: Theo xác suất hay xảy ra:

1. **Viewport khác nhau:** Local 1920x1080, CI default 1280x720. Element bị responsive đẩy đi. Fix: `use: { viewport: { width: 1440, height: 900 } }`.
2. **Headless vs headed:** Một số element behavior khác trong headless (hover, focus). Test local headless để giống CI.
3. **Tốc độ CI chậm hơn:** Em không vội tăng timeout toàn cục — xem trace để biết action nào chậm và tăng có chọn lọc.
4. **Race condition data:** Local 1 test, CI 4 worker cùng tạo `test@example.com` → conflict. Dùng `faker` hoặc `test.info().workerIndex` để tạo unique.
5. **Environment variable:** `BASE_URL` local là `localhost:3000`, CI là staging có rate-limit/cache khác.
6. **Timezone:** CI thường UTC, local +07:00. Set `timezoneId: 'Asia/Ho_Chi_Minh'` trong context.

Câu 17. Test suite chạy 25 phút trên CI. Em tối ưu xuống thế nào?

Trả lời: Em không bắt đầu bằng "tăng worker lên". Em bắt đầu bằng đo:

1. **Đo:** Reporter HTML, xem top 10 test chậm. Thường 80% thời gian nằm ở 20% test.
2. **Replace UI setup bằng API setup:** Test cần "user có 5 order" — đừng click qua UI. Gọi `POST /api/orders` 5 lần qua `request` fixture. Một test từ 45s xuống 8s là chuyện thường.
3. **Reuse authentication:** `storageState` (Câu 7). Trước tối ưu, mỗi test login 3s × 300 test = 15 phút chỉ để login.
4. **Parallel ở mức project:** Tách thành nhiều project (smoke, regression, billing, admin). CI chạy mỗi project song song trên runner riêng.
5. **Sharding:** Khi hết cách tối ưu code: `--shard=1/4`, `--shard=2/4`. Merge HTML report bằng blob reporter.

6. **Chỉ chạy test liên quan:** Trên PR dùng `git diff` xác định path đổi, chỉ chạy test liên quan. Full suite chạy nightly.

Kết quả thực tế ở một dự án: 38 phút → 6 phút sau khi áp dụng tuần tự các bước trên.

Câu 18. Em integrate Playwright vào CI/CD thế nào? Xử lý report fail ra sao?

Trả lời: Setup GitHub Actions điển hình:
- Dùng official Docker image `mcr.microsoft.com/playwright:v1.x-jammy` — tránh thiếu OS dep cho browser.
- Cache `node_modules` nhưng KHÔNG cache browser của Playwright — version mismatch gây flaky khó debug.
- Sharding 4-way: 4 job song song, merge report cuối cùng.
- Upload artifact `playwright-report/` và `test-results/` — luôn (kể cả pass, để debug intermittent).
- Publish HTML report lên GitHub Pages hoặc S3 — không bắt dev tải zip về.

Khi fail trên main branch:
1. **Slack notification** với link trực tiếp đến HTML report.
2. **Trace zip đính kèm** — người fix mở trace ngay được, không phải reproduce local.
3. **Quy tắc:** Test flaky trên main → tạo ticket, KHÔNG retry rồi quên. Test deterministic fail → block deploy.

Câu 19. Em assert API call mà UI trigger ngầm — không có DOM thay đổi rõ?

Trả lời: Đăng ký `waitForRequest` / `waitForResponse` TRƯỚC khi click:

```
// Verify analytics event được gửi đúng payload
const trackingRequest = page.waitForRequest(req =>
  req.url().includes('/api/track') && req.method() === 'POST'
);

await page.getByRole('button', { name: 'Add to cart' }).click();

const request = await trackingRequest;
const payload = request.postDataJSON();
expect(payload.event).toBe('product_added');
expect(payload.productId).toBe('SKU-123');
```

Technique nâng cao: Collect TẤT CẢ request trong test, assert ở cuối — hữu ích cho test "không được gọi API X":

```
const requests: string[] = [];
page.on('request', req => requests.push(req.url()));

await page.getByRole('checkbox', { name: 'Stay logged in' }).check();
// Action này không được gọi /api/save-preference

expect(requests.filter(u => u.includes('/api/save-preference'))).toHaveLength(0);
```

Câu 20. Test fail trên CI nhưng pass khi `--debug` local. Em điều tra thế nào?

Trả lời: Em phân biệt rõ: `--debug` làm 2 thứ — (1) chạy headed, (2) timeout vô hạn. Bản thân nó đã thay đổi điều kiện chạy. Việc "debug pass, CI fail" rất bình thường, không phải bug Playwright.

Điều tra theo hướng "loại bỏ biến số":

1. **Tách yếu tố headed:** Chạy local headed không `--debug` — vẫn pass? Vậy headed không phải nguyên nhân.
2. **Tách yếu tố tốc độ:** Chạy local headless với `slowMo: 0` — bằng tốc độ CI. Fail → race condition tốc độ.
3. **Tách yếu tố môi trường:** SSH vào CI runner (dùng `tmate` trong GitHub Actions). Hoặc chạy CI Docker image local: `docker run -it mcr.microsoft.com/playwright:v1.x`.
4. **Bật trace toàn bộ:** `trace: 'on'` cho test đó trên CI. Xem action nào "đứng" lâu, console có lỗi gì, network có request nào timeout.
5. **Last resort:** `DEBUG=pw:api,pw:protocol` log Playwright. Xem CDP command nào fail.

Trong kinh nghiệm em, top 3 nguyên nhân: (a) test phụ thuộc thứ tự — local chạy 1 test pass, CI chạy nguyên file thì state từ test trước rò qua; (b) data race với worker khác; (c) animation trên CI

vẫn còn ở thời điểm assert vì CI render chậm hơn, local "may mắn" thấy state ổn.

Phụ lục A — Page Object KHÔNG chứa assertion (chi tiết Câu 10)

❌ Cách SAI — Page object có assertion bên trong

```
export class LoginPage {
  constructor(private page: Page) {}

  async login(username: string, password: string) {
    await this.page.getByLabel('Username').fill(username);
    await this.page.getByLabel('Password').fill(password);
    await this.page.getByRole('button', { name: 'Sign in' }).click();

    // ❌ Assertion lẫn vào đây
    await expect(this.page.getByText('Welcome')).toBeVisible();
    await expect(this.page).toHaveURL('/dashboard');
  }
}
```

Vấn đề: Test "sai password" sẽ throw ngay tại `login()` vì assertion `Welcome` không pass. Test "redirect về trang trước" cũng fail vì URL không phải `/dashboard`. Mỗi flow mới phải tạo thêm method `loginExpectingError()`, `loginAndRedirectTo()` — page object phình ra theo số test case.

✅ Cách ĐÚNG — Page object chỉ làm action

```
export class LoginPage {
  readonly errorMessage = this.page.getByRole('alert');
  readonly usernameField = this.page.getByLabel('Username');
  readonly passwordField = this.page.getByLabel('Password');
  readonly submitButton = this.page.getByRole('button', { name: 'Sign in' });

  constructor(private page: Page) {}

  async goto() {
    await this.page.goto('/login');
  }

  // Chỉ làm action — không assert
  async submitCredentials(username: string, password: string) {
    await this.usernameField.fill(username);
    await this.passwordField.fill(password);
    await this.submitButton.click();
  }
}
```

Test linh hoạt với mọi scenario:

```
test('login thành công → vào dashboard', async ({ page }) => {
  const loginPage = new LoginPage(page);
  await loginPage.goto();
  await loginPage.submitCredentials('admin', 'correct-password');
  await expect(page).toHaveURL('/dashboard');
});

test('sai password → hiển thị error', async ({ page }) => {
  const loginPage = new LoginPage(page);
  await loginPage.goto();
  await loginPage.submitCredentials('admin', 'wrong-password');
  await expect(loginPage.errorMessage).toHaveText('Invalid credentials');
});

test('redirect về trang trước khi login', async ({ page }) => {
  await page.goto('/orders/123');
  const loginPage = new LoginPage(page);
  await loginPage.submitCredentials('admin', 'correct-password');
  await expect(page).toHaveURL('/orders/123');
});
```

Page Object	Test
Action: click, fill, navigate	**Assertion:** `expect(...).toBe(...)`
Query: trả về text, count, boolean	**Logic:** quyết định cái gì là "đúng"
Expose locators để test reuse	**Tự pick locator** từ page object

Naming tip: Đặt `submitCredentials()` thay vì `login()`. Tên trung tính, không ngụ ý "thành công" — dùng được cho mọi outcome.

Phụ lục B — Fixture: setup/teardown cho test cases

1. Cấu trúc fixture chuẩn

```
// fixtures.ts
import { test as base } from '@playwright/test';

export const test = base.extend<{
  authenticatedPage: Page;
}>({
  authenticatedPage: async ({ page }, use) => {
    // ===== SETUP (chạy TRƯỚC mỗi test) =====
    await page.goto('/login');
    await page.getByLabel('Username').fill('admin');
    await page.getByLabel('Password').fill('password');
    await page.getByRole('button', { name: 'Sign in' }).click();

    // ===== TRAO QUYỀN CHO TEST =====
    await use(page);

    // ===== TEARDOWN (chạy SAU mỗi test) =====
    await page.evaluate(() => localStorage.clear());
  },
});

export { expect } from '@playwright/test';
```

2. Fixture trả về data, có cleanup

```ts
export const test = base.extend<{ testUser: User; testOrder: Order }>({
  testUser: async ({ request }, use) => {
    // Tạo user qua API (nhanh hơn UI)
    const response = await request.post('/api/test/users', {
      data: { email: `test-${Date.now()}@example.com`, role: 'admin' },
    });
    const user = await response.json();

    await use(user);

    // Cleanup
    await request.delete(`/api/test/users/${user.id}`);
  },

  // Fixture phụ thuộc fixture khác
  testOrder: async ({ request, testUser }, use) => {
    const response = await request.post('/api/orders', {
      headers: { Authorization: `Bearer ${testUser.token}` },
      data: { items: [{ sku: 'SKU-1', qty: 1 }] },
    });
    const order = await response.json();
    await use(order);
    await request.delete(`/api/orders/${order.id}`);
  },
});
```

3. Fixture scope — `test` vs `worker`

Scope	Chạy khi nào	Dùng cho
`'test'` (mặc định)	Mỗi test một lần	User test, data test, page state
`'worker'`	Một lần / worker process	API token chung, DB connection, expensive setup

```ts
export const test = base.extend<{}, { apiToken: string }>({
  apiToken: [
    async ({}, use) => {
      const { token } = await fetchToken();
      await use(token);  // Chia sẻ giữa các test trong worker
    },
    { scope: 'worker' },
  ],
});
```

⚠️ Worker-scope fixture phải **read-only** hoặc **idempotent** — nếu test thay đổi state, test sau sẽ thấy state đã đổi.

4. `auto` fixture — thay thế `beforeEach` toàn cục

```ts
export const test = base.extend<{ testTimer: void }>({
  testTimer: [
    async ({}, use, testInfo) => {
      const start = Date.now();
      await use();
      console.log(`${testInfo.title}: ${Date.now() - start}ms`);
    },
    { auto: true },  // Tự chạy cho mọi test, không cần inject
  ],
});
```

Use case: Log start/end, attach screenshot khi fail, reset DB trước mỗi test.

5. Nhiều page trong cùng test — multi-user testing

`authenticatedPage: Page` chỉ là type annotation. Em hoàn toàn có thể có nhiều fixture cùng kiểu `Page`:

```ts
export const test = base.extend<{
  buyerPage: Page;
  sellerPage: Page;
}>({
  buyerPage: async ({ browser }, use) => {
    const context = await browser.newContext({ storageState: 'auth/buyer.json' });
    const page = await context.newPage();
    await use(page);
    await context.close();
  },

  sellerPage: async ({ browser }, use) => {
    const context = await browser.newContext({ storageState: 'auth/seller.json' });
    const page = await context.newPage();
    await use(page);
    await context.close();
  },
});
```

```
test('buyer gửi tin → seller nhận realtime', async ({ buyerPage, sellerPage }) => {
  await buyerPage.goto('/orders/123/chat');
  await sellerPage.goto('/orders/123/chat');

  await buyerPage.getByPlaceholder('Type a message').fill('Khi nào ship hàng?');
  await buyerPage.getByRole('button', { name: 'Send' }).click();

  await expect(sellerPage.getByText('Khi nào ship hàng?')).toBeVisible();
});
```

Quy tắc về context vs page:

Tình huống	Cách làm
Hai **user khác nhau**	Mỗi user một **context** riêng (cookie cách ly)
Cùng một **user, 2 tab**	Hai **page** trong **cùng context**
Test cần >10 user	Cân nhắc k6/Artillery thay vì Playwright

❌ **Bẫy:** Hai page chung một context = cùng cookies/session → login user thứ 2 sẽ đè user thứ 1.

6. Khi nào dùng gì

Tình huống	Nên dùng
Setup giống nhau cho 1-2 test gần nhau	`test.beforeEach` trong `test.describe`
Setup dùng lại ở nhiều file, có cleanup	**Custom fixture (test-scope)**
Setup tốn chi phí, không đổi giữa test	**Custom fixture (worker-scope)**
Chạy code cho MỌI test (logging, reset DB)	**Auto fixture**
Login cho gần như mọi test	`storageState` **ở config level** (rẻ hơn fixture)

Câu 21. Test data management — em không hard-code `test@example.com` khắp nơi, em làm thế nào?

Vì sao họ hỏi: Test data tệ là nguồn flaky test số 2 sau race condition. Junior thường tạo data inline trong từng test, suite phình to và conflict liên tục.

Trả lời: Em dùng 3 tầng theo độ phức tạp:

1. Faker cho data đơn giản: Email, tên, địa chỉ — dùng `@faker-js/faker`. Mỗi test có data unique, không conflict song song.

```ts
import { faker } from '@faker-js/faker';

const email = faker.internet.email();
const phone = faker.phone.number('+84 9# ### ####');
```

2. Builder pattern cho object phức tạp: Order có nhiều field, mỗi test chỉ cần override 1-2 field. Builder cho phép default values + override linh hoạt.

```ts
class OrderBuilder {
  private data: OrderInput = {
    customer: faker.person.fullName(),
    items: [{ sku: 'SKU-1', qty: 1, price: 100 }],
    shippingAddress: faker.location.streetAddress(),
    paymentMethod: 'card',
  };

  withItems(items: OrderItem[]) { this.data.items = items; return this; }
  withPayment(method: PaymentMethod) { this.data.paymentMethod = method; return this
  build() { return { ...this.data }; }
}

// Test chỉ override cái nó quan tâm
const order = new OrderBuilder()
  .withPayment('cod')
  .build();
```

3. Test data API cho data cần persist: BE expose endpoint riêng `/api/test/seed-user` chỉ enable ở môi trường test, tạo data nhanh và đúng cách (đã hash password, set role đúng, sync cache).

Quy tắc cứng:

- **Mỗi test tự tạo data, không phụ thuộc data có sẵn.** Test "user A có order" → tự tạo user A + order, không assume DB đã có.
- **Worker-scoped unique:** Email/username chứa `test.info().workerIndex` hoặc timestamp + random để 4 worker song song không conflict.
- **Cleanup qua fixture, không qua `afterAll`.** Fixture cleanup chạy cả khi test fail; `afterAll` thì không.

Câu 22. App dùng Shadow DOM (Web Components, Salesforce Lightning) — em test thế nào?

Vì sao họ hỏi: Selenium từng phải dùng JavaScript inject để xuyên Shadow DOM. Câu này check em có biết Playwright xử lý tự động không.

Trả lời: Playwright **tự động pierce shadow DOM** với hầu hết locator. Em dùng `getByRole`, `getByText`, `getByLabel` như bình thường — Playwright sẽ tự đi xuyên shadow boundary.

```
// HTML:
// <my-button>
//   #shadow-root
//     <button>Submit</button>
// </my-button>

// ✅ Hoạt động bình thường — không cần code đặc biệt
await page.getByRole('button', { name: 'Submit' }).click();
```

Điều cần biết:
- **CSS selector cũng pierce shadow DOM:** `page.locator('my-button button')` work — Playwright tự đi xuyên.
- **XPath KHÔNG pierce shadow DOM.** Đây là lý do nữa em không dùng XPath.
- **Component test attribute:** Nếu app dùng Stencil/Lit và set `:host` attribute, em vẫn select được qua `getByTestId`.

Bẫy thường gặp với Salesforce Lightning: Component lazy-render — locator có thể tìm thấy element trước khi nó interactive. Em luôn dùng action (như `click`) hoặc assertion (`toBeEnabled()`) thay vì check `isVisible()` rồi click — Playwright sẽ tự retry.

Khi nào pierce KHÔNG work: `closed` shadow root (`attachShadow({ mode: 'closed' })`). Hiếm gặp trong web app thật, nhưng nếu gặp thì không có cách nào — phải hỏi dev đổi sang `open`.

Câu 23. Em test responsive design / mobile viewport như thế nào?

Vì sao họ hỏi: Trong sản phẩm thật, 60-70% traffic từ mobile. Test chỉ desktop là bỏ sót lớn.

Trả lời: Em dùng `devices` preset của Playwright + project riêng cho mobile:

```ts
// playwright.config.ts
import { devices } from '@playwright/test';

export default defineConfig({
  projects: [
    { name: 'desktop-chrome', use: { ...devices['Desktop Chrome'] } },
    { name: 'mobile-chrome', use: { ...devices['Pixel 7'] } },
    { name: 'mobile-safari', use: { ...devices['iPhone 14'] } },
  ],
});
```

`devices` preset set sẵn: viewport, user-agent, `isMobile: true`, `hasTouch: true`, device scale factor.

Khác biệt thực tế khi test mobile:

```ts
// Touch event thay vì click
await page.locator('.card').tap();

// Swipe gesture
await page.locator('.carousel').dispatchEvent('touchstart', { touches: [...] });

// Mobile menu thường là drawer ẩn — phải mở trước khi tương tác
await page.getByRole('button', { name: 'Menu' }).tap();
await page.getByRole('link', { name: 'Profile' }).tap();
```

Em KHÔNG chạy tất cả test trên cả desktop và mobile. Quy tắc:
- **Smoke test:** Chạy cả desktop + mobile để đảm bảo cả 2 đường base hoạt động.
- **Test responsive-specific:** Hamburger menu, drawer, bottom sheet — chỉ mobile project.
- **Test logic nghiệp vụ:** Chỉ desktop để tiết kiệm thời gian. Logic không khác giữa viewport.

Bẫy: Playwright "mobile" không phải iOS Safari thật — nó vẫn là Chromium/WebKit chạy headless. Bug iOS-specific (Safari quirk, viewport keyboard) phải test trên BrowserStack/Sauce Labs với device thật.

Câu 24. Test feature liên quan đến thời gian — countdown, expiry, "đặt lịch 30 phút sau"?

Vì sao họ hỏi: Đây là test khó. Cách amateur là `waitForTimeout(60000)` — sai và chậm. Câu này lọc senior.

Trả lời: Em dùng `page.clock` (Playwright 1.45+) để mock `Date`, `setTimeout`, `setInterval`:

```
test('countdown hết hạn sau 30s thì hiển thị "Expired"', async ({ page }) => {
  // Cài time giả TRƯỚC khi vào trang
  await page.clock.install({ time: new Date('2026-01-01T10:00:00Z') });

  await page.goto('/checkout');
  await expect(page.locator('.countdown')).toHaveText('00:30');

  // "Tua nhanh" 30 giây — không phải chờ thật
  await page.clock.fastForward('00:30');

  await expect(page.locator('.countdown')).toHaveText('Expired');
});
```

Use case thường gặp:

```
// 1. Test "thông báo nhắc nhở sau 24h"
await page.clock.install({ time: '2026-01-01T10:00:00Z' });
await page.goto('/dashboard');
await page.clock.fastForward('24:00:00'); // tua 24 giờ
await expect(page.getByText('Reminder')).toBeVisible();

// 2. Test "session timeout sau 15 phút không active"
await page.clock.install();
await page.goto('/');
// User không thao tác
await page.clock.fastForward('15:01');
await expect(page).toHaveURL(/\/login/); // bị đẩy về login

// 3. Test "đặt lịch tương lai"
await page.clock.setFixedTime(new Date('2026-06-15T10:00:00')); // freeze time
// Hiển thị calendar — "hôm nay" là 15/06/2026
```

Quan trọng: `page.clock.install()` phải gọi TRƯỚC `page.goto()`. Nếu app đã chạy `setInterval` rồi mới install clock, interval đó không bị mock.

Trường hợp không dùng được `page.clock`: Khi thời gian được tính ở BE (ví dụ token expiry check ở server). Lúc đó em phải:
- Mock BE response trả về thời gian em muốn.
- Hoặc BE expose endpoint `/api/test/advance-clock?seconds=900` chỉ enable ở env test.

Câu 25. Drag and drop — em làm thế nào và những bẫy gì?

Vì sao họ hỏi: Drag-drop có nhiều cách implement (HTML5 native, react-dnd, react-beautiful-dnd, custom với mouse event). Mỗi loại cần kỹ thuật khác.

Trả lời: Em thử theo thứ tự độ tin cậy:

Cách 1 — `dragTo()`: Cách đơn giản nhất, dùng cho HTML5 native drag-and-drop.

```
await page.locator('#item-1').dragTo(page.locator('#dropzone'));
```

Cách 2 — Manual mouse event: Khi `dragTo()` không trigger được handler (thường vì lib custom):

```
const source = page.locator('#item-1');
const target = page.locator('#dropzone');

await source.hover();
await page.mouse.down();
// Một số lib cần move qua điểm trung gian để trigger detection
await page.mouse.move(100, 100, { steps: 5 });
await target.hover();
await page.mouse.up();
```

Cách 3 — Dispatch event trực tiếp: Khi lib dùng synthetic events (react-dnd), cả hai cách trên đều không work:

```
await source.dispatchEvent('dragstart');
await target.dispatchEvent('drop');
await source.dispatchEvent('dragend');
```

Bẫy thực tế:
- **`steps` quan trọng:** `mouse.move(x, y)` không có `steps` → chuột "teleport", lib drag-drop không detect. Luôn dùng `steps: 5` trở lên.
- **Element phải visible trong viewport:** Drag từ item ngoài viewport thường fail. Scroll vào view trước.
- **Touch device:** Mobile drag-drop dùng `touchstart` / `touchmove` / `touchend`, không phải mouse event. Phải test trên project có `hasTouch: true`.

Khi nào em không test drag-drop qua UI: Khi UI drag-drop chỉ là cách trigger action ở BE (reorder list → gọi API `PATCH /items/reorder`). Em test API thẳng, UI test chỉ verify "kéo xong list update đúng" — không cần test pixel-perfect drag behavior.

Câu 26. Em verify state ở database sau action UI như thế nào?

Vì sao họ hỏi: Test "click Submit, form hiển thị success" không đủ — phải verify data thực sự lưu vào DB. Câu này lọc người đã làm test end-to-end thật.

Trả lời: Em có 3 cách, ưu tiên theo thứ tự:

1. API admin endpoint (recommended): BE expose endpoint chỉ admin truy cập, return state cần verify.

```
test('tạo order qua UI → DB lưu đúng total', async ({ page, adminApi }) => {
  await page.goto('/checkout');
  await page.getByRole('button', { name: 'Place order' }).click();

  const orderId = await page.locator('[data-order-id]').getAttribute('data-order-id'

  // Verify qua API admin
  const response = await adminApi.get(`/api/admin/orders/${orderId}`);
  const order = await response.json();

  expect(order.total).toBe(250000);
  expect(order.status).toBe('pending_payment');
  expect(order.items).toHaveLength(3);
});
```

2. Connect DB trực tiếp trong test: Khi không có API admin, dùng client như `pg`, `mysql2`, `mongodb`.

```
import { Pool } from 'pg';

const db = new Pool({ connectionString: process.env.TEST_DB_URL });

test('verify trực tiếp DB', async ({ page }) => {
  await page.goto('/...');
  await page.getByRole('button', { name: 'Save' }).click();

  const { rows } = await db.query(
    'SELECT * FROM orders WHERE customer_email = $1',
    [testEmail]
  );
  expect(rows).toHaveLength(1);
  expect(rows[0].total).toBe(250000);
});
```

3. Worker fixture cho DB connection: Connect 1 lần, dùng cho cả worker.

```ts
export const test = base.extend<{}, { db: Pool }>({
  db: [
    async ({}, use) => {
      const pool = new Pool({ connectionString: process.env.TEST_DB_URL });
      await use(pool);
      await pool.end();
    },
    { scope: 'worker' },
  ],
});
```

Quan điểm thực tế:

- **Ưu tiên API admin endpoint.** DB schema có thể đổi, API thì có contract.
- **Đừng query DB cho mọi assertion.** Nếu UI hiển thị đúng → 90% là DB đúng. Chỉ query DB khi UI không hiển thị được (background job, audit log, side effect).
- **Cẩn thận với connection pool trên CI.** 4 worker × N connection = dễ exhaust pool. Set max connection thấp hoặc dùng connection pooler (PgBouncer).

Câu 27. Khi nào em viết Component Test thay vì E2E test?

Vì sao họ hỏi: Playwright có `@playwright/experimental-ct-*` để test React/Vue component isolated. Nhiều người không biết hoặc không biết khi nào dùng.

Trả lời: Em theo testing pyramid — không phải mọi thứ đều E2E:

Loại	Khi nào dùng	Ví dụ
Unit test	Logic thuần (utils, helpers, business rules)	`calculateDiscount()`
Component test	UI component có state phức tạp, isolated	`<DatePicker>`, `<MultiSelect>`
E2E test	User flow xuyên nhiều page, integration thật	"Checkout với coupon → thanh toán → email confirmation"

Khi nào em viết Component test:

```tsx
// component.spec.ts (CT)
import { test, expect } from '@playwright/experimental-ct-react';
import { DatePicker } from './DatePicker';

test('DatePicker disable ngày cuối tuần khi prop weekendsDisabled=true', async ({ mo
  let selectedDate: Date | null = null;
  const component = await mount(
    <DatePicker
      weekendsDisabled
      onChange={(d) => { selectedDate = d; }}
    />
  );

  await component.getByRole('button', { name: 'Saturday' }).click();
  expect(selectedDate).toBeNull(); // không bị set vì weekend disabled

  await component.getByRole('button', { name: 'Monday' }).click();
  expect(selectedDate).not.toBeNull();
});
```

Ưu điểm Component test:

- **Nhanh hơn E2E nhiều lần** — không cần boot app, không cần DB.
- **Test mọi state dễ dàng** — pass props trực tiếp, không phải setup data qua API.
- **Isolated** — không bị ảnh hưởng bởi bug component khác.

Em KHÔNG viết Component test cho:

- Component thuần trình bày (button, badge) — không có logic gì để test.
- Integration giữa nhiều component — đó là việc của E2E.

Thực tế ở team em: ~70% unit test, ~20% component test cho component phức tạp, ~10% E2E cho critical flow. E2E ít nhưng phủ hết "happy path" của các flow ra tiền.

Câu 28. App realtime (WebSocket, SSE, chat) — em test thế nào?

Vì sao họ hỏi: Realtime test khó vì không có request/response rõ ràng. `waitForResponse` không work vì WebSocket là kết nối dài.

Trả lời: Em dùng 3 kỹ thuật tùy bài toán:

1. Assert qua UI (đơn giản nhất, recommended):

```
test('user A gửi tin → user B nhận', async ({ pageA, pageB }) => {
  await pageA.goto('/chat/room-1');
  await pageB.goto('/chat/room-1');

  await pageA.getByPlaceholder('Message').fill('Hello');
  await pageA.getByRole('button', { name: 'Send' }).click();

  // toBeVisible() có auto-retry — chờ WebSocket push tới
  await expect(pageB.getByText('Hello')).toBeVisible({ timeout: 5000 });
});
```

WebSocket "ẩn" — em quan tâm kết quả cuối (user B thấy tin nhắn), không quan tâm cách nó tới.

2. Listen WebSocket frame khi cần verify protocol:

```
const frames: any[] = [];
page.on('websocket', ws => {
  ws.on('framereceived', frame => frames.push(JSON.parse(frame.payload)));
  ws.on('framesent', frame => console.log('Sent:', frame.payload));
});

await page.goto('/chat');
await page.getByRole('button', { name: 'Send' }).click();

// Đợi frame chứa tin nhắn về
await expect.poll(() =>
  frames.find(f => f.type === 'message' && f.text === 'Hello')
).toBeTruthy();
```

3. Mock WebSocket server (advanced): Khi BE chưa có WebSocket, em chạy mock server với `ws` lib trong Node và point app vào đó.

Bẫy thực tế:

- **Connection lifecycle:** WebSocket connect bất đồng bộ. Đôi khi test `goto` xong, send message ngay → connection chưa ready, message rớt. Em chờ một signal "đã connected" (UI hiển thị "online", hoặc poll endpoint).
- **Reconnection:** App tốt sẽ tự reconnect khi mất mạng. Em test bằng `context.setOffline(true)` rồi `false`, verify UI vẫn nhận được message sau khi online lại.
- **SSE (Server-Sent Events):** Playwright không có API riêng — em assert qua UI, hoặc dùng `page.on('response')` để intercept response stream.

Câu 29. Em handle browser permission (notification, location, camera, clipboard)?

Vì sao họ hỏi: Permission prompt là blocker — không grant thì test không tiếp tục được. Junior thường stuck ở đây.

Trả lời: Em grant permission ở mức context, TRƯỚC khi page navigate:

```
// Trong fixture hoặc test
const context = await browser.newContext({
  permissions: ['geolocation', 'notifications', 'clipboard-read', 'clipboard-write'],
  geolocation: { latitude: 10.7769, longitude: 106.7009 }, // Saigon
  locale: 'vi-VN',
  timezoneId: 'Asia/Ho_Chi_Minh',
});
```

Các permission thường dùng:

```
// 1. Test "tìm cửa hàng gần tôi" — mock GPS
await context.grantPermissions(['geolocation']);
await context.setGeolocation({ latitude: 21.0285, longitude: 105.8542 }); // Hà Nội

// 2. Test web notification
await context.grantPermissions(['notifications']);

// 3. Test "copy link" button
await context.grantPermissions(['clipboard-read', 'clipboard-write']);
await page.getByRole('button', { name: 'Copy link' }).click();
const clipboardText = await page.evaluate(() => navigator.clipboard.readText());
expect(clipboardText).toBe('https://example.com/share/abc');

// 4. Camera/microphone — phức tạp hơn, dùng fake media
const context = await browser.newContext({
  permissions: ['camera', 'microphone'],
});
// Chromium chạy với flag --use-fake-device-for-media-stream
```

Test "user từ chối permission": Set permissions rỗng và verify app hiển thị fallback UI đúng.

```
const context = await browser.newContext({ permissions: [] });
await page.goto('/');
await page.getByRole('button', { name: 'Use my location' }).click();
await expect(page.getByText('Permission denied. Enter city manually:')).toBeVisible(
```

Bẫy: `grantPermissions` chỉ work cho permission API của browser. Native dialog (file picker, print dialog) không control được bằng permission — phải dùng `waitForEvent('filechooser')` hoặc

emulate print.

Câu 30. Quy trình code review test code của team em thế nào? Em reject PR test khi nào?

Vì sao họ hỏi: Câu cuối, đánh giá maturity của em với automation. Senior nhìn test code khác junior — không chỉ "test pass là OK".

Trả lời: Em có checklist khi review PR test, reject nếu vi phạm:

1. Reject ngay nếu:
- `waitForTimeout` **mà không có comment giải thích.** 99% là race condition giấu đi.
- **Assertion trong page object.** Vi phạm separation of concerns (xem Phụ lục A).
- **Test phụ thuộc test khác** (`test.serial` lạm dụng, share variable global). Test phải chạy độc lập, theo bất kỳ thứ tự nào.
- **Hard-code data sẽ conflict trên CI parallel.** Email cố định, ID cố định.
- **Selector dùng XPath hoặc CSS class hash.** `.css-1a2b3c4` của CSS-in-JS — sẽ break ở build sau.
- **Test không có assertion thực sự.** "Test chỉ click qua các bước, không expect gì" — đó là smoke test che giấu, không phải verification.

2. Comment yêu cầu sửa nếu:
- **Tên test không rõ ý đồ.** `test('test 1')`, `test('it works')` → đổi sang `test('user thấy lỗi khi nhập email sai format')`.
- **Test quá dài (>50 dòng).** Thường là test 2 thứ trong 1 test — tách ra.
- **Duplicate setup giữa các test.** Đề xuất extract thành fixture.
- **Locator dài và phức tạp.** `page.locator('div > div.foo > span:nth-child(3)')` → yêu cầu thêm `data-testid` hoặc dùng `getByRole`.
- **Comment giải thích "code làm gì".** Code tự nói được — comment chỉ giải thích "tại sao".

3. Approve có note nếu:
- **Test pass nhưng có thể tối ưu** (gọi UI thay vì API setup). Approve để không block, tạo ticket follow-up.
- **Coverage chưa đủ edge case.** Yêu cầu thêm test case trong PR sau.

Anti-pattern em hay gặp nhất: *"Em sửa test cho nó pass"* — Junior thấy test fail, sửa expected value thành actual value để pass, không hiểu vì sao fail. Khi review em luôn hỏi: *"Test fail vì code sai, hay test sai?"* Nếu không trả lời được → reject.

Mindset em truyền cho team: Test code là production code. Cùng standard, cùng review chặt. Test code dirty rồi → 6 tháng sau không ai dám đụng → suite chết.

- **Đừng đọc thuộc lòng API.** Interviewer không quan tâm em nhớ đúng `setInputFiles` hay `loadFiles`. Họ quan tâm em biết bài toán nào dùng cái nào.
- **Luôn nói "tại sao".** "Em dùng `getByRole` vì bền hơn và bonus là check accessibility" ăn điểm hơn nhiều so với "em dùng `getByRole`".
- **Nói về thất bại thật.** "Em từng tăng retry để giấu flaky test, sau đó học được phải fix root cause" — câu này nói thật, người ta tin em hơn.
- **Hỏi ngược lại interviewer:** *"Team đang gặp pain point gì lớn nhất với test automation hiện tại?"* — vừa thể hiện em quan tâm dự án thật, vừa biết team họ đang ở đâu.