30 Câu Hỏi Phỏng Vấn Playwright — Phiên Bản Nói Miệng

Tài liệu này viết để **luyện nói** trước buổi phỏng vấn HR/technical screen, nơi interviewer hỏi miệng và em trả lời miệng — không có laptop, không show code.

Cách dùng:
- Đọc câu trả lời, hiểu cấu trúc lập luận (quan điểm → lý do → ví dụ thực tế → kết luận).
- Đừng học thuộc lòng. Học **nhịp điệu** và **câu chốt** ấn tượng.
- Mỗi câu trả lời ~1-2 phút khi nói. Tập nói trước gương hoặc ghi âm để check.

Nguyên tắc khi trả lời phỏng vấn nói:
1. Mở đầu bằng câu chốt rõ ràng, không vòng vo.
2. Dùng "em từng gặp tình huống...", "team em quy ước...", "ví dụ thực tế là..." — đây là tín hiệu của người làm thật chứ không đọc tài liệu.
3. Có ý kiến cá nhân + biết trade-off — không phải đáp án tuyệt đối.
4. Không kể tên API hoặc method dài dòng. Nói khái niệm, ví dụ tự nhiên.

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

A: Em không nói Playwright tốt hơn hai cái kia — em nói nó phù hợp hơn với bài toán team em. Có ba lý do chính.

Thứ nhất là cross-browser thật sự. Cypress chỉ chạy trên một bản Chromium đã được vá lại, còn Playwright chạy trực tiếp được cả WebKit, Firefox, Chromium. Sản phẩm team em có user dùng Safari trên iPhone, nên đây là yêu cầu cứng không thương lượng được.

Thứ hai là auto-wait và web-first assertion. Tức là khi em assert một element visible, Playwright tự retry trong khung timeout cho đến khi thấy nó. Hồi em làm Selenium, phải tự viết explicit wait khắp nơi, mà vẫn flaky. Chuyển sang Playwright thì flaky giảm khoảng 80% mà không phải làm gì đặc biệt.

Thứ ba là tốc độ và cách ly. Playwright có khái niệm browser context — như profile incognito, tạo và xóa cực rẻ. Suite khoảng 400 case của team em chạy song song xong trong 4 phút trên CI.

💡 **Mẹo nói:** Đừng nói "Cypress dở" hay "Selenium lỗi thời". Interviewer có thể đang dùng những tool đó. Nói theo hướng "phù hợp với bài toán" sẽ an toàn và chuyên nghiệp hơn.

🎯 **Follow-up hay gặp:** *"Khi nào em không chọn Playwright?"* → Khi cần test Safari thật trên iOS device thì Playwright không thay được — vẫn phải BrowserStack hoặc device thật. Hoặc project legacy phải support IE11 thì bắt buộc Selenium.

Câu 2. Khi nào em cần chờ một element xuất hiện, và khi nào auto-wait đã đủ?

A: Mặc định em không tự chờ gì cả. Playwright đã có auto-wait built-in cho mọi action như click, fill, và mọi assertion. Em chỉ chờ thủ công trong ba tình huống.

Một là chờ một element biến mất — ví dụ chờ spinner loading ẩn đi trước khi click tiếp. Cái này auto-wait không lo được vì action không tự biết phải chờ cái khác mất.

Hai là chờ một điều kiện không gắn với element nào trên DOM. Ví dụ em cần chờ một biến global trên window được set xong, hoặc chờ app báo "đã ready" qua một flag. Lúc đó em chờ theo function.

Ba là chờ một network response cụ thể. Ví dụ sau khi submit form, UI chưa thay đổi rõ ràng nhưng em muốn assert API đã trả 200. Em chờ theo URL và status.

Còn nếu em phải dùng "chờ cố định N giây" — ví dụ chờ 3 giây — thì 99% là code đang giấu một race condition. Em sẽ tìm điều kiện thực sự cần chờ, không chờ mò.

💡 **Mẹo nói:** Câu chốt "chờ mò là che bug, không phải fix bug" rất ăn điểm. Interviewer thường gật đầu ngay.

Câu 3. Test của em chạy song song và đôi khi có ca thất bại lẻ tẻ. Em debug thế nào?

A: Flaky test là vấn đề ai cũng gặp. Em có quy trình ba bước chứ không đoán mò.

Bước một là thu thập bằng chứng. Em bật trace recording khi test fail lần đầu — Playwright sẽ tự lưu lại toàn bộ timeline: action, network request, console log, screenshot từng bước. Em mở trace ra xem được chính xác bước nào đứng và DOM lúc đó ra sao.

Bước hai là tái hiện lỗi. Em chạy file đó lặp lại 20 lần với 4 worker. Nếu fail 2-3 trên 20 thì đúng là flaky thật, còn fail 1 trên 100 thì thường do môi trường — không nên cố fix sai chỗ.

Bước ba là phân loại nguyên nhân. Trong kinh nghiệm em, 90% flaky đến từ ba nhóm: một là test phụ thuộc data của test khác — em sửa bằng cách dùng fixture tạo data riêng; hai là chờ UI thay vì chờ data đến từ API — em sửa bằng cách chờ response thật; ba là animation CSS chưa xong nhưng test đã assert — em tắt animation toàn cục khi test.

Cái em tuyệt đối tránh là tăng retry lên 3 lần rồi xong. Đó là che lỗi chứ không phải sửa lỗi. Sau 3 tháng test sẽ thật sự fail và không ai biết nguyên nhân.

💡 **Mẹo nói:** Nhấn mạnh "có quy trình" — đây là tín hiệu của senior. Junior thường nói "em thử retry trước, không được thì add wait" — câu đó interviewer trừ điểm ngay.

Câu 4. Em giải thích Browser, BrowserContext và Page bằng một ví dụ thực tế giúp em.

A: Em hay giải thích bằng tình huống test app e-commerce có buyer và seller chat với nhau.

Browser là process Chromium hoặc WebKit thật sự, em chỉ cần một cái cho cả test suite. Bật tắt nó tốn khoảng một đến hai giây, nên không nên làm nhiều lần.

BrowserContext giống như một profile incognito. Mỗi context có cookie, localStorage, permission riêng — hoàn toàn cách ly. Trong tình huống chat buyer-seller, em tạo hai context: một cho buyer đã login bằng tài khoản buyer, một cho seller. Mỗi context tạo và đóng chỉ tốn vài mili giây.

Page là tab bên trong context. Buyer có thể có nhiều page trong cùng context của mình — tương tự một user mở nhiều tab trên Chrome.

Hệ quả thực chiến là: mỗi test nên có context riêng, đừng tái sử dụng. Vì nếu tái dùng, state rác từ test trước sẽ rớt sang. May là Playwright Test mặc định đã làm vậy rồi qua fixture page, em không phải lo.

💡 **Mẹo nói:** Câu chuyện buyer-seller cụ thể rất gây ấn tượng. Interviewer sẽ hình dung được ngay em đã giải bài toán thật, không phải đọc tài liệu.

Câu 5. Em phân biệt Locator và ElementHandle như thế nào? Tại sao em không dùng ElementHandle?

A: Locator là một mô tả về cách tìm phần tử. Nó chưa thực sự tìm — Playwright sẽ tìm ngay tại thời điểm em gọi action. Còn ElementHandle là con trỏ trực tiếp đến element đã tìm thấy ngay tại một thời điểm cố định.

Hệ quả là trong app SPA hiện đại, React hay Vue re-render liên tục, ElementHandle sẽ bị stale ngay — kiểu như lỗi stale element reference của Selenium ngày xưa. Locator thì không, vì nó re-query mỗi lần em call action, có tự retry trong khung timeout.

Trong ba năm dùng Playwright em chưa gặp tình huống nào bắt buộc phải dùng ElementHandle. Mọi thứ ElementHandle làm được thì Locator cũng làm — lấy text, lấy attribute, lấy bounding box. Nên

team em ban hẳn việc dùng ElementHandle trong lint rule.

💡 **Mẹo nói:** Nếu interviewer hỏi tiếp "vậy còn `page.$()` thì sao?" — đó là API legacy trả về ElementHandle, em cũng tránh. Câu này nói thêm sẽ ghi điểm thêm.

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

A: Selector chiếm khoảng 50% công sức maintain test suite, nên em có thứ tự ưu tiên rõ ràng, team em còn viết vào lint rule luôn.

Ưu tiên đầu tiên là selector theo role và name — ví dụ tìm button có tên "Submit". Đây là cách bền nhất, và bonus là nó ép code đi qua accessibility tree. Nếu test không tìm được thì thường app có vấn đề về a11y, đáng để fix.

Ưu tiên hai là theo label, placeholder, hoặc text hiển thị — dùng cho form và content tĩnh.

Ưu tiên ba là test ID. Khi DOM không có semantic rõ — ví dụ một thẻ div làm icon button — em sẽ yêu cầu dev thêm thuộc tính test ID. Team em quy ước test ID phải có ý nghĩa, không phải số ngẫu nhiên.

CSS selector em chỉ dùng khi ba cái trên không khả thi, và phải là class bền — không phải class hash kiểu Tailwind hay CSS Module sinh ra theo build.

Còn XPath thì gần như không bao giờ. Trong ba năm em chỉ dùng đúng một lần, là khi test một bảng dữ liệu cũ không có test ID và cần selector theo quan hệ — kiểu "cell ở cùng row với text X".

Lúc nào em phá quy tắc? Khi label hoặc role thay đổi theo ngôn ngữ. App tiếng Việt mà em ưu tiên text thì mỗi lần dịch lại là test fail. Lúc đó em chuyển sang dùng test ID — hoặc tốt hơn nữa là dùng key i18n thay vì text hiển thị.

💡 **Mẹo nói:** Câu "bonus là ép code đi qua accessibility tree" rất hay — vừa thể hiện em hiểu sâu, vừa cho thấy em quan tâm a11y.

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

A: Em dùng kỹ thuật lưu trạng thái đăng nhập vào file một lần, rồi tái dùng cho mọi test sau.

Cụ thể là em có một project setup chạy đầu tiên, thực hiện login một lần — và em không click qua UI mà gọi thẳng API login để lấy token, nhanh hơn nhiều. Sau khi login xong, em lưu cookie và localStorage vào một file JSON.

Sau đó mỗi project test sẽ khai báo "phụ thuộc setup" và dùng file JSON đó làm trạng thái khởi tạo. Mỗi test mở context đã có sẵn cookie và token, không phải login lại.

Khi em cần test với nhiều role khác nhau — admin, manager, user thường — em tạo nhiều file JSON khác nhau và mỗi project test trỏ tới một file. Test admin và test user chạy song song không conflict gì cả.

Còn login flow thật thì sao? Em vẫn phải test, không thể skip. Em vô hiệu hóa storage state mặc định cho riêng file đó, để nó vào trang login như user mới.

Kết quả thực tế ở dự án em: trước tối ưu, mỗi test login mất khoảng 3 giây, nhân với 300 test là 15 phút chỉ để login. Sau khi áp dụng kỹ thuật này còn dưới 30 giây cho toàn bộ.

💡 **Mẹo nói:** Số liệu "15 phút giảm xuống dưới 30 giây" là câu ấn tượng. Phải có con số cụ thể chứ đừng nói chung chung "nhanh hơn nhiều".

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

A: Em dùng kỹ thuật chặn request từ client để mock response. Nhưng em chia ra ba tầng tùy độ thật cần.

Tầng một là mock thuần. Khi API hoàn toàn chưa có, em đọc spec OpenAPI hoặc Postman collection của BE, viết handler trả về dữ liệu giả lập.

Tầng hai là modify response thật. Khi API có rồi nhưng em cần test edge case khó tái hiện — ví dụ trường hợp danh sách có 100 item, hoặc trường hợp BE trả lỗi 500. Em không mock toàn bộ, mà fetch response thật rồi chỉ sửa phần cần test. Như vậy phần payload còn lại vẫn realistic.

Tầng ba là mock lỗi mạng. Em làm cho request bị abort hoặc timeout để verify UI có hiển thị error state đúng không.

Quan trọng là em không mock 100% trong regression suite. Em chỉ mock có chọn lọc cho edge case khó tái hiện. Regression chính vẫn phải hit BE thật trên môi trường staging — nếu không thì test pass mà sản phẩm vẫn lỗi vì contract giữa FE và BE đã đổi mà mình không biết.

💡 **Mẹo nói:** Câu chốt "mock quá nhiều thì test thành test với chính mock của mình" — sẽ làm interviewer cười và nhớ em.

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

A: Playwright có sẵn HTTP client trong test runner, nên em dùng nó trong ba tình huống thực tế.

Một là setup dữ liệu trước test. Ví dụ test "user xem đơn hàng đã đặt" — em sẽ gọi API tạo order trước, không click qua UI tạo từng đơn. Một test có thể từ 45 giây xuống còn 8 giây chỉ nhờ thay đổi này.

Hai là cleanup. Test xong em xóa data đã tạo qua DELETE endpoint, để test sau có môi trường sạch.

Ba là hybrid test. Ví dụ tạo order qua UI, rồi verify nó tồn tại qua API bằng admin token. Đây là pattern em dùng nhiều — UI test verify "user thấy gì", còn API verify "BE đã lưu đúng chưa".

Còn lúc nào em không dùng Playwright? Em không dùng cho API contract test thuần. Đó là việc của Postman/Newman, RestAssured, hoặc Pact. Lý do là Playwright thiếu tooling chuyên dụng — schema validation, contract testing, snapshot response. Trộn chung làm suite vừa chậm vừa rối, không tận dụng được tool chuyên dụng.

💡 **Mẹo nói:** Câu "test setup qua API thay vì UI" là câu kinh điển. Nếu chưa nói được ở câu 17 (tối ưu suite), đảm bảo nói được ở đây.

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

A: Em không dùng POM truyền thống nữa — kiểu class với một đống locator và method. Tài liệu chính thức của Playwright thực ra cũng không khuyến nghị nó nữa. Em dùng hai pattern hợp với Playwright hơn.

Một là fixture-based. Mỗi page là một fixture trả về object có các action chính. Test có thể destructure trực tiếp từ fixture, đỡ phải khởi tạo class mỗi lần. Khi em cần login page trong test, em chỉ cần inject loginPage, không phải nhớ tạo new LoginPage gì cả.

Hai là component object. Thay vì một page object khổng lồ cho cả trang, em chia theo component. Header có HeaderComponent, bảng order có OrderTableComponent. Cách này khớp với cách front-end chia component, nên dễ tìm và dễ maintain.

Có một quy tắc cứng em luôn giữ: page object không chứa assertion. Assertion thuộc về test. Page object chỉ làm action và trả về dữ liệu. Lý do là nếu page object có assertion bên trong, thì cùng một method không thể dùng cho cả happy path và error case. Ví dụ method login mà bên trong assert "Welcome" hiển thị, thì test "login sai password" sẽ throw ngay tại method login chứ chưa đến chỗ em muốn assert thông báo lỗi.

💡 **Mẹo nói:** Quy tắc "page object không chứa assertion" rất ăn điểm vì nhiều ứng viên không nghĩ ra. Nếu interviewer hỏi sâu, em có thể kể tình huống cụ thể của login.

Câu 11. Em test upload file như thế nào? Và những trường hợp tricky em từng gặp?

A: Trường hợp thường là input file chuẩn HTML — em set file trực tiếp vào input. Cái này work cả khi input bị ẩn bằng CSS, vì Playwright thao tác qua DOM chứ không qua giao diện thật.

Trường hợp tricky thứ nhất là drag-drop zone không có input thật. Lúc đó em lắng nghe sự kiện chọn file của browser — khi user click vào zone, em can thiệp ở mức browser dialog.

Tricky thứ hai là test với file rất lớn, ví dụ 500MB. Em không bao giờ upload file thật trong E2E test, vì nó chậm và tốn storage CI. Em chia làm hai test: một test với file nhỏ — chỉ vài KB — để verify flow upload đúng; một test với mock response từ BE trả về lỗi "file quá lớn" để verify UI hiển thị thông báo đúng.

Tricky thứ ba là khi em không muốn để file thật trong repo — em tạo file content trực tiếp trong code bằng buffer, không cần file vật lý trên disk. Đây là kỹ thuật em dùng nhiều khi test upload CSV — sinh CSV trong test, upload, verify.

💡 **Mẹo nói:** "Không bao giờ upload file thật trong E2E" là tín hiệu của người đã đụng vấn đề performance trên CI thật.

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

A: Em chia làm ba bước rõ ràng và có một bẫy lớn em sẽ nói cuối cùng.

Bước một là đăng ký listener trước khi click. Đây là điều kiện sống còn — nếu em click nút download xong rồi mới đăng ký listener thì sự kiện đã trôi qua, em không bao giờ bắt được.

Bước hai là lấy đối tượng download nhưng chưa vội save. Em có thể lấy ra tên file đề xuất để assert — ví dụ tên file phải có dạng "orders-năm-tháng-ngày.csv" không.

Bước ba là save vào path tạm và đọc content để verify. Em assert nội dung có header CSV đúng không, số dòng đúng không.

Bẫy lớn em từng gặp là trên CI headless, một số browser preview PDF thay vì download — kiểu Chrome có PDF viewer built-in. Lúc đó listener download không bao giờ fire. Cách giải quyết là yêu

cầu BE trả response với header Content-Disposition là attachment, hoặc nút download có thuộc tính download trong HTML — như vậy browser bắt buộc tải xuống chứ không preview.

💡 **Mẹo nói:** Phần "bẫy PDF preview" là điểm sáng — chỉ người đã thực sự đụng case mới biết.

Câu 13. Test một popup OAuth — ví dụ Sign in with Google mở tab mới — em handle thế nào?

A: Em handle bằng cách đăng ký listener "có page mới được mở" trước khi click vào nút đăng nhập. Khi popup xuất hiện, em nhận được handle của tab mới và thao tác trên đó như một page thông thường. Sau khi OAuth xong, popup tự đóng và em quay lại assert trên page chính.

Nhưng quan điểm thực tế của em là: em hiếm khi test OAuth thật trong E2E. Lý do là Google và GitHub có CAPTCHA và rate-limit chống bot — bot test cũng bị chặn. Test thật sẽ flaky cực kỳ.

Em làm theo hai hướng. Một là test OAuth callback của riêng app em — tức là phần backend nhận token từ Google rồi xử lý. Phần đó em mock IdP, hoặc dùng tài khoản test đã được Google whitelist riêng cho team em. Hai là dùng storage state có sẵn token để bỏ qua hoàn toàn bước đăng nhập, chỉ test các phần sau đăng nhập.

💡 **Mẹo nói:** Nhiều người nghĩ phải test cả OAuth flow thật — câu này em chỉ ra "đừng cố làm cái không kiểm soát được" là tín hiệu của người đã trầy xước thực tế.

Câu 14. Em test iframe của bên thứ ba như Stripe hay reCAPTCHA thế nào?

A: Em phân ra hai loại iframe, xử lý khác nhau.

Iframe của chính app em — ví dụ một widget custom — thì em dùng frame locator để định vị, rồi thao tác bên trong như page thường.

Iframe của third party như reCAPTCHA thì em không cố test. Đây là dịch vụ chống bot, mà test automation chính là bot — em có cố thì cũng bị chặn. Cách thực tế là yêu cầu BE expose một endpoint test-mode bỏ qua captcha cho môi trường staging, hoặc dùng test key reCAPTCHA của Google luôn pass.

Với Stripe payment, em không thanh toán thật. Stripe có sẵn test card số 4242 4242 4242 4242 dành riêng cho test. Em verify form submit đúng và webhook nhận được, không cần giao dịch thật.

Một bẫy thường gặp với iframe là nó load lazy — em đã define frame locator nhưng iframe chưa render. Em không dùng wait cố định, mà chờ bằng assertion. Khi em assert một element bên trong

frame visible, Playwright sẽ tự đợi cả iframe load.

💡 **Mẹo nói:** Câu "không test cái mình không kiểm soát" là nguyên tắc rất senior. Lặp lại nhiều lần ở các câu khác cũng được — đó là cách suy nghĩ.

Câu 15. Visual regression test — em đã dùng chưa? Em xử lý ảnh khác nhau giữa Mac và Linux thế nào?

A: Em có dùng. Playwright có hỗ trợ chụp screenshot và so sánh tự động. Nhưng visual test rất dễ flaky nếu setup sai, nên em follow ba nguyên tắc.

Một là baseline phải sinh trong môi trường giống CI. Mac và Linux render font khác nhau vì cách anti-aliasing khác nhau — em không bao giờ commit baseline từ máy local của mình. Em luôn sinh baseline trong Docker image dùng cho CI, hoặc dùng official Playwright Docker image. Như vậy baseline và actual luôn render cùng môi trường.

Hai là mask phần dynamic. Timestamp, avatar user, banner quảng cáo — những phần này thay đổi mỗi lần test, em che lại bằng feature mask. Em cũng cho phép sai số khoảng 1% pixel để chấp nhận anti-aliasing khác biệt nhỏ.

Ba là tắt animation trước khi chụp. Nếu không, mỗi lần chụp có thể bắt được frame khác của animation.

Quan trọng là em không visual test cho mọi page. Em chỉ test cho component thật sự quan trọng — landing page, design system. Visual test chạy chậm và mỗi lần đổi design phải review lại baseline, nên phải có quy trình review baseline trong pull request.

💡 **Mẹo nói:** "Visual test có chi phí maintain cao" — câu này thể hiện em hiểu trade-off, không phải fan cuồng tool.

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

A: Em có checklist sáu mục, đi từ xác suất hay xảy ra nhất.

Một là viewport. Local em mở trình duyệt size 1920x1080, CI mặc định 1280x720. Element có thể bị responsive layout đẩy đi hoặc ẩn dưới fold. Em set viewport cố định trong config để tránh.

Hai là headless versus headed. Một số behavior khác nhau trong headless — hover, focus, hoặc các tính năng cần GPU. Em test local ở chế độ headless để giống CI.

Ba là tốc độ. CI thường chậm hơn máy local. Em không vội tăng timeout toàn cục, mà mở trace ra xem action nào chậm và tăng có chọn lọc.

Bốn là race condition data. Local em chạy một test, CI chạy bốn worker song song — bốn test cùng tạo user "test arobase example chấm com" thì conflict. Em sửa bằng cách tạo data unique với timestamp hoặc worker index.

Năm là environment variable. Base URL local là localhost, CI là staging có rate-limit và cache khác. Em log base URL ở đầu test để xác nhận đúng môi trường.

Sáu là timezone. CI thường chạy UTC, local Việt Nam là plus 7. Test liên quan đến date time fail là vì đây. Em set timezone trong context cố định là Asia Ho Chi Minh.

💡 **Mẹo nói:** Liệt kê có thứ tự theo xác suất là tín hiệu "có hệ thống". Đừng nói lung tung "có nhiều nguyên nhân lắm".

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

A: Em không bắt đầu bằng "tăng worker lên". Em bắt đầu bằng đo và đi tuần tự sáu bước.

Một là đo. Em bật reporter HTML, xem top 10 test chậm nhất. Quy luật 80/20 luôn đúng — 80% thời gian nằm ở 20% test.

Hai là thay setup qua UI bằng setup qua API. Test cần "user có 5 đơn hàng" — em không click qua UI tạo từng đơn, mà gọi API tạo trực tiếp. Một test có thể từ 45 giây xuống 8 giây chỉ với thay đổi này.

Ba là tái sử dụng đăng nhập — như em đã nói ở câu 7. Trước tối ưu, 300 test login 3 giây mỗi cái là 15 phút chỉ để đăng nhập.

Bốn là parallel ở mức project. Em tách suite thành nhiều project — smoke, regression, billing, admin. CI chạy mỗi project trên một runner riêng song song.

Năm là sharding. Khi hết cách tối ưu code, em chia test ra nhiều máy CI. Mỗi máy chạy một phần, sau cùng merge report lại.

Sáu là chỉ chạy test liên quan. Trên pull request, em không chạy full suite. Em xem file nào đổi và chỉ chạy test liên quan. Full suite chỉ chạy nightly.

Kết quả thực tế ở một dự án của em: 38 phút xuống còn 6 phút sau khi áp dụng tuần tự sáu bước trên.

💡 **Mẹo nói:** Con số "38 phút xuống 6 phút" là câu kết ấn tượng — nhớ nói rõ.

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

A: Em dùng GitHub Actions là chính. Setup điển hình của em có vài nguyên tắc.

Một là dùng Docker image chính thức của Playwright — tránh chuyện thiếu thư viện hệ thống cho browser. Đây là lỗi rất hay gặp khi setup từ Ubuntu trống.

Hai là cache node modules nhưng không cache browser. Browser của Playwright phải khớp đúng phiên bản với package, nếu cache không khéo sẽ tạo flaky rất khó debug.

Ba là sharding bốn way — bốn job song song, sau cùng merge report lại.

Bốn là upload artifact luôn, kể cả khi test pass. Vì test "pass" có thể vừa qua retry, em cần trace để biết. Nếu chỉ upload khi fail thì khi debug intermittent sẽ không có gì.

Năm là publish HTML report lên GitHub Pages hoặc S3. Em không bắt dev tải zip về máy rồi unzip rồi mở file index — đó là trải nghiệm tệ.

Còn khi test fail trên main branch, em có ba quy tắc. Một là Slack notification có link trực tiếp đến report, không phải "fail, vào CI xem". Hai là trace zip đính kèm để người fix mở ngay được, không phải reproduce local. Ba là quy tắc cứng: test flaky thì tạo ticket fix, không bao giờ thêm retry rồi quên. Test deterministic fail thì block deploy.

💡 **Mẹo nói:** Câu "không bao giờ thêm retry rồi quên" là cam kết chất lượng — interviewer rất thích nghe.

Câu 19. Em assert một API call mà UI trigger ngầm — không có DOM thay đổi rõ ràng — em làm thế nào?

A: Em đăng ký lắng nghe request hoặc response trước khi thực hiện hành động trigger. Sau khi click xong, em chờ event đó về và assert payload hoặc status.

Ví dụ thực tế là test analytics. Khi user click "Add to cart", em muốn verify một event tracking được gửi đi với đúng product ID và đúng action. Không có DOM thay đổi gì — chỉ có request đi ngầm. Em lắng nghe request đến endpoint analytics, click button, rồi assert payload có field "product added" với đúng SKU.

Có một kỹ thuật nâng cao em dùng cho test phủ định — test "không được gọi API X khi user làm Y". Ví dụ user click checkbox "remember me" thì không được gọi save preference ngay, mà phải đợi user click Save. Em collect tất cả request trong test, rồi assert ở cuối là không có request nào tới endpoint đó. Đây là pattern hay bị bỏ sót — chỉ test happy path thì không bắt được lỗi này.

💡 **Mẹo nói:** Test phủ định "không được gọi" là chi tiết rất sâu, ít người nghĩ tới. Nói được câu này thì em rất nổi bật.

Câu 20. Test fail trên CI nhưng pass khi em debug local — em điều tra thế nào?

A: Trước hết em phân biệt rõ một điều: chế độ debug làm hai thứ — chạy headed và timeout vô hạn. Bản thân nó đã thay đổi điều kiện chạy, nên "debug pass, CI fail" là chuyện rất bình thường, không phải bug Playwright.

Em điều tra theo hướng loại bỏ biến số, từng bước một.

Một là tách yếu tố headed. Em chạy local headed nhưng không debug — vẫn pass không? Nếu vẫn pass thì headed không phải nguyên nhân.

Hai là tách yếu tố tốc độ. Em chạy local headless với tốc độ thật — bằng CI. Nếu fail thì là race condition tốc độ. Nếu vẫn pass thì là môi trường.

Ba là tách yếu tố môi trường. Em SSH vào CI runner — GitHub Actions có tool tmate cho phép vào runner đang chạy. Hoặc em chạy chính Docker image của CI ngay tại máy local. Nếu reproduce được thì là môi trường, không phải code.

Bốn là bật trace toàn bộ cho test đó trên CI, xem action nào đứng lâu nhất, console có lỗi gì, network có request nào timeout.

Năm là biện pháp cuối — bật debug log của Playwright thật chi tiết. Xem command nào fail ở tầng protocol.

Trong kinh nghiệm em, top ba nguyên nhân hay gặp: một là test phụ thuộc thứ tự — local chạy một test pass, CI chạy nguyên file thì state từ test trước rò qua; hai là data race giữa các worker; ba là animation trên CI vẫn còn ở thời điểm assert vì CI render chậm hơn, local em "may mắn" thấy state ổn.

💡 **Mẹo nói:** "Loại bỏ biến số từng bước" là tư duy khoa học. Câu chuyện cụ thể về tmate cũng là tín hiệu của người làm thật.

Câu 21. Em quản lý test data thế nào? Không lẽ hard-code "test arobase example chấm com" ở mọi nơi?

A: Test data tệ là nguyên nhân flaky thứ hai sau race condition, nên em có chiến lược ba tầng.

Tầng một là data đơn giản như email, tên, địa chỉ — em sinh random bằng thư viện faker. Mỗi test có data unique nên 4 worker chạy song song không conflict.

Tầng hai là builder pattern cho object phức tạp. Ví dụ order có 15 field — em không muốn mỗi test phải khai báo cả 15 field khi chỉ quan tâm 1 field. Em viết builder có default values cho mọi field, test chỉ override cái nó quan tâm. Khi cần test "thanh toán COD", em chỉ builder dot with payment COD dot build — không cần biết các field còn lại có gì.

Tầng ba là test data API. Khi data phức tạp cần persist vào DB, em không tạo trực tiếp qua SQL vì như vậy bỏ qua business logic. Em yêu cầu BE expose endpoint chỉ enable ở môi trường test — ví dụ "seed user with role admin" — endpoint này hash password đúng, set role đúng, sync cache nếu cần.

Có một quy tắc cứng em luôn áp dụng: mỗi test tự tạo data của mình, không phụ thuộc data có sẵn. Test "user A có đơn hàng" thì tự tạo user A và tạo đơn — không assume DB đã có sẵn. Lý do là DB trên CI có thể bị reset bất kỳ lúc nào, hoặc test khác xóa data mà em không biết.

Và cleanup phải qua fixture, không qua afterAll. Fixture cleanup chạy cả khi test fail; afterAll thì không — nên dùng afterAll sẽ để rác lại khi có test fail.

💡 **Mẹo nói:** "Test tự lo data của mình" là nguyên tắc rất ăn điểm. Junior thường share data global rồi rối loạn.

Câu 22. App em test dùng Web Components hoặc Shadow DOM — em xử lý thế nào?

A: Tin tốt là Playwright tự động đi xuyên Shadow DOM với hầu hết locator. Em dùng getByRole, getByText, getByLabel như bình thường — Playwright tự đi xuyên shadow boundary, không cần kỹ thuật đặc biệt.

Đây là điểm khác lớn so với Selenium ngày xưa — Selenium phải inject JavaScript thủ công để xuyên shadow DOM, rất khổ.

Có vài điều cần biết. CSS selector cũng đi xuyên được. XPath thì không — đây là một lý do nữa em không dùng XPath.

Trường hợp duy nhất Playwright không xuyên được là closed shadow root — tức dev khai báo shadow root là closed. Cái này hiếm gặp trong app thật, nhưng nếu gặp thì không có cách nào — phải nhờ dev đổi sang open.

Em từng test app Salesforce Lightning có rất nhiều Shadow DOM lồng nhau. Bài học là component lazy-render — locator tìm thấy element trước khi nó interactive. Em không check visible rồi click, mà dùng action trực tiếp như click hoặc assertion như toBeEnabled — Playwright sẽ tự retry cho đến khi element ready.

💡 **Mẹo nói:** Nhắc Salesforce Lightning hoặc Stencil là tín hiệu em đã đụng case thật, không chỉ đọc tài liệu.

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

A: Em dùng device preset của Playwright. Họ có sẵn cấu hình cho Pixel, iPhone, iPad — đầy đủ viewport, user agent, touch enabled, pixel ratio. Em chỉ cần import và áp vào project là xong.

Khi test mobile, em phải đổi cách thao tác. Click thành tap, vì mobile dùng touch event. Swipe gesture thì em dispatch touch event với tọa độ start và end. Menu mobile thường là drawer ẩn — em phải mở drawer trước rồi mới tap vào item bên trong.

Em không chạy tất cả test trên cả desktop và mobile. Quy tắc của em là: smoke test chạy cả hai để đảm bảo cả hai đường base hoạt động; test responsive-specific như hamburger menu hay bottom sheet thì chỉ mobile project; test logic nghiệp vụ thì chỉ desktop để tiết kiệm thời gian, vì logic không khác giữa hai viewport.

Có một bẫy lớn em phải nói rõ: Playwright "mobile" không phải iOS Safari thật. Nó vẫn là Chromium hoặc WebKit chạy headless với viewport mobile. Bug iOS-specific như quirk của Safari, hành vi keyboard pop up trên iOS — những cái này phải test trên device thật qua BrowserStack hoặc Sauce Labs. Đừng nhầm tưởng test mobile project xong là đã phủ iOS — đó là sai lầm em thấy nhiều team mắc.

💡 **Mẹo nói:** Câu "Playwright mobile không phải iOS thật" rất ăn điểm — nhiều người không biết.

Câu 24. Em test feature liên quan đến thời gian — countdown, expiry, "đặt lịch 30 phút sau" — em làm thế nào?

A: Em dùng kỹ thuật mock đồng hồ của browser. Playwright có API cho phép em "đóng băng" hoặc "tua nhanh" thời gian.

Ví dụ test countdown 30 giây thì hiển thị Expired: em set thời gian giả là 10 giờ sáng hôm nay, vào trang checkout, rồi tua nhanh 30 giây — không phải chờ thật 30 giây. Test chạy gần như instant.

Em dùng kỹ thuật này cho nhiều use case: test thông báo nhắc nhở sau 24 giờ, test session timeout sau 15 phút không active, test "đặt lịch tương lai" cần freeze ngày hôm nay là một ngày cụ thể.

Có một điều quan trọng phải nhớ: phải install clock trước khi navigate vào page. Nếu page đã chạy setInterval rồi mới install clock, interval đó không bị mock — vẫn chạy thật.

Một trường hợp không dùng được là khi thời gian được tính ở BE — ví dụ token expiry check ở server side. Lúc đó em không control được clock của BE từ test. Cách giải quyết là mock luôn response của BE trả về thời gian em muốn, hoặc nhờ BE expose endpoint test mode cho phép em advance clock ở server.

💡 **Mẹo nói:** Trước đây nhiều người dùng waitForTimeout cho test countdown — câu này em chứng minh em đã biết cách làm đúng.

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

A: Drag drop có nhiều cách implement khác nhau — HTML5 native, lib React DnD, lib React Beautiful DnD, hoặc custom với mouse event. Mỗi loại cần kỹ thuật khác, nên em thử theo thứ tự độ tin cậy.

Cách một là dùng API dragTo của Playwright — đơn giản nhất, work với HTML5 native drag drop.

Cách hai là manual mouse event khi cách một không trigger được. Em hover vào source, mouse down, di chuột qua các điểm trung gian — không teleport thẳng tới target — rồi mouse up tại target. Phải di qua điểm trung gian vì nhiều lib drag drop chỉ detect khi có mouse move sự kiện liên tục, không phải teleport.

Cách ba là dispatch event trực tiếp khi cả hai cách trên đều không work. Nhiều lib dùng synthetic event của React — chỉ nhận event dispatch thủ công, không nhận event thật từ browser. Em dispatch dragstart trên source, drop trên target, dragend trên source.

Bẫy em đã gặp: số lượng bước khi mouse move quan trọng. Nếu di chuột không có steps thì chuột teleport — lib không detect. Phải có ít nhất 5 bước. Element phải visible trong viewport — drag từ item ngoài viewport thường fail. Phải scroll vào view trước.

Một quan điểm thực tế: em không test pixel-perfect drag behavior nếu UI drag drop chỉ là cách trigger action ở BE. Ví dụ kéo reorder list rồi gọi API reorder — em test API thẳng, UI test chỉ verify "kéo xong list update đúng". Đỡ tốn công và đỡ flaky hơn nhiều.

💡 **Mẹo nói:** Có ba cách backup và biết khi nào dùng cái nào — đây là người đã trầy nhiều với drag drop.

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

A: Câu này quan trọng vì test "click Submit, hiện success" không đủ — phải verify data thật sự lưu vào DB. Em có ba cách, ưu tiên theo thứ tự.

Cách một và là cách em ưu tiên là qua API admin endpoint. BE expose endpoint chỉ admin truy cập, trả về state của bất kỳ resource nào — ví dụ get order by ID có cả internal field. Sau khi tạo order qua UI, em gọi API này verify total đúng, status đúng, số item đúng.

Cách hai là connect DB trực tiếp trong test. Khi không có API admin, em dùng client như pg cho Postgres hoặc mongodb cho MongoDB, query thẳng vào DB.

Cách ba là worker fixture cho DB connection — connect một lần dùng cho cả worker, không connect lại mỗi test.

Quan điểm thực tế của em: ưu tiên API admin endpoint. Lý do là DB schema có thể đổi liên tục, còn API thì có contract — đổi sẽ break nhiều consumer khác chứ không phải mỗi test. Test mà query DB trực tiếp thì mỗi lần đổi schema lại break, maintain mệt.

Và em không query DB cho mọi assertion. Nếu UI hiển thị đúng thì 90% là DB đúng. Em chỉ query DB khi UI không hiển thị được — ví dụ verify audit log, background job đã chạy chưa, hoặc side effect mà user không thấy.

Cẩn thận một điều với connection pool trên CI: 4 worker mỗi worker mở vài connection là dễ exhaust pool của DB. Em set max connection thấp hoặc dùng connection pooler.

💡 **Mẹo nói:** "Ưu tiên API admin vì có contract" là tư duy maintain dài hạn, rất senior.

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

A: Em theo testing pyramid — không phải mọi thứ đều E2E.

Đáy pyramid là unit test cho logic thuần — utils, helpers, business rules. Ví dụ hàm tính discount, hàm validate email. Nhanh nhất, viết nhiều nhất.

Giữa là component test cho UI component có state phức tạp, test isolated. Ví dụ DatePicker với prop disable cuối tuần, MultiSelect có search, autocomplete với keyboard navigation. Em mount component với props cụ thể, test mọi state mà không cần boot cả app.

Đỉnh là E2E test cho user flow xuyên nhiều page, integration thật. Ví dụ checkout với coupon, thanh toán, nhận email confirmation. Chậm nhất, viết ít nhất, nhưng phủ critical flow ra tiền.

Ưu điểm component test mà em thấy rõ là nó nhanh hơn E2E nhiều lần — không cần boot app, không cần DB. Test mọi state dễ dàng vì pass props trực tiếp, không phải setup data qua API. Và isolated — không bị ảnh hưởng bởi bug component khác.

Em không viết component test cho component thuần trình bày như button hay badge — không có logic gì để test. Và không viết cho integration giữa nhiều component — đó là việc của E2E.

Thực tế ở team em phân bố khoảng 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 — đó là cách cân

bằng cost và confidence.

💡 **Mẹo nói:** Testing pyramid là kiến thức nền — biết áp dụng đúng tỷ lệ là tín hiệu senior.

Câu 28. Em test app realtime như chat hoặc notification — WebSocket — như thế nào?

A: Realtime test khó vì không có request-response rõ ràng. WebSocket là kết nối dài, không có cấu trúc "click rồi đợi response" như REST. Em dùng ba kỹ thuật tùy bài toán.

Cách một và là cách em ưu tiên là assert qua UI. Em không quan tâm WebSocket bên dưới, chỉ quan tâm kết quả cuối. User A gửi tin nhắn, user B phải thấy tin nhắn đó. Em dùng assertion có auto-retry — Playwright sẽ chờ cho đến khi UI bên B hiển thị tin từ A. Đơn giản và bền vững.

Cách hai là khi cần verify protocol — ví dụ payload WebSocket có đúng format không. Em lắng nghe sự kiện framereceived và framesent trên WebSocket connection, collect các frame vào mảng, rồi assert mảng có frame em mong đợi.

Cách ba là mock WebSocket server khi BE chưa có. Em chạy mock server bằng lib ws của Node trong test, point app vào mock đó. Cách này ít dùng, chỉ khi parallel development.

Bẫy em đã gặp với realtime: connection lifecycle bất đồng bộ. Em vào page, gửi message ngay — connection chưa ready, message rớt im lặng. Em phải chờ một signal "đã connected" — ví dụ UI hiển thị "online", hoặc trạng thái indicator chuyển xanh.

Reconnection cũng phải test. App tốt sẽ tự reconnect khi mất mạng. Em test bằng cách set context offline rồi online lại, verify UI vẫn nhận được message sau khi online lại.

Còn Server-Sent Event thì Playwright không có API riêng — em assert qua UI, hoặc intercept response stream nếu cần verify nội dung.

💡 **Mẹo nói:** "Assert kết quả cuối, không quan tâm cách nó tới" — câu chốt rất hay cho realtime test.

Câu 29. Em handle browser permission như notification, location, camera khi test?

A: Permission prompt là blocker — không grant thì test không tiếp tục được. Em grant permission ở mức context, trước khi page navigate.

Khi tạo context, em khai báo luôn các permission cần — geolocation, notification, clipboard, camera, microphone. Em cũng set sẵn vị trí GPS giả, locale, timezone — như user đang ở Sài Gòn hay Hà Nội tùy test case.

Ví dụ test thực tế: feature "tìm cửa hàng gần tôi" — em grant geolocation và set tọa độ Sài Gòn, verify danh sách cửa hàng đúng vùng. Feature "copy link" — em grant clipboard read và write, click button copy, đọc clipboard ra verify đúng URL được copy vào.

Camera và microphone phức tạp hơn một chút. Em grant permission, nhưng còn cần Chromium chạy với flag fake media stream — để app nhận được video giả thay vì cố mở camera thật.

Em cũng test trường hợp ngược lại: user từ chối permission. Em set permission rỗng, verify app hiển thị fallback UI đúng. Ví dụ feature "Use my location" mà user deny thì phải có fallback "Enter city manually".

Một bẫy nhỏ: cách grant này chỉ work cho permission API của browser. Native dialog như file picker hay print dialog thì không control được bằng permission — phải dùng event listener của Playwright cho từng loại.

💡 **Mẹo nói:** Test "user từ chối permission" là edge case rất hay quên. Nói được câu này thể hiện em nghĩ về negative test case.

Câu 30. Quy trình review test code của team em ra sao? Em reject pull request test khi nào?

A: Em coi test code là production code — cùng standard, cùng review chặt. Em có checklist rõ ràng, reject ngay nếu vi phạm vài điểm.

Một là dùng chờ cố định mà không có comment giải thích — 99% là race condition giấu đi.

Hai là có assertion trong page object. Em đã nói ở câu 10 — vi phạm separation of concerns.

Ba là test phụ thuộc thứ tự — lạm dụng serial, share variable global giữa các test. Test phải chạy độc lập theo bất kỳ thứ tự nào.

Bốn là hard-code data sẽ conflict trên CI parallel. Email cố định, ID cố định — chỉ pass trên local một worker, lên CI bốn worker là vỡ.

Năm là selector dùng XPath hoặc CSS class hash sinh tự động. Class hash kiểu "css gạch một hai ba bốn" của CSS in JS — sẽ break ngay khi build lại.

Sáu là test không có assertion thực sự. Test chỉ click qua các bước nhưng không expect gì — đó là smoke test che giấu chứ không phải verification.

Em sẽ comment yêu cầu sửa nếu thấy tên test không rõ ý đồ, test quá dài thường là test hai thứ trong một test, hoặc duplicate setup giữa nhiều test có thể extract thành fixture.

Anti-pattern em hay gặp nhất ở junior là "em sửa test cho nó pass" — thấy test fail liền sửa expected value thành actual value, không hiểu tại sao fail. Khi review em luôn hỏi: "test fail vì code sai, hay test

sai?" — nếu không trả lời được thì reject.

Mindset em truyền cho team là: test code dirty rồi thì 6 tháng sau không ai dám đụng, suite chết. Đầu tư công review từ đầu rẻ hơn nhiều so với rewrite cả suite sau này.

💡 **Mẹo nói:** Câu chốt "test fail vì code sai, hay test sai?" là câu trademark của senior — interviewer sẽ nhớ em qua câu này.

Lời khuyên chung khi đi phỏng vấn Playwright

Trước phỏng vấn:
- Đọc qua mô tả công việc kỹ — họ dùng tech stack gì, sản phẩm gì. Câu trả lời của em nên có ví dụ liên quan đến domain họ làm.
- Chuẩn bị 2-3 câu chuyện thật về dự án em đã làm — flaky test em đã fix, suite em đã tối ưu, bug em đã catch nhờ test. Câu chuyện cụ thể luôn ăn điểm hơn lý thuyết.
- Tập nói trước gương hoặc ghi âm. Nhịp điệu và tự tin là yếu tố quan trọng không kém nội dung.

Trong phỏng vấn:
- **Mở đầu bằng câu chốt, không vòng vo.** "Em không nói X tốt hơn Y, em nói X phù hợp hơn với bài toán này" — câu mở đầu kiểu này rất chuyên nghiệp.
- **Luôn nói "tại sao".** Mỗi technique đều giải thích lý do em chọn nó thay vì cách khác. "Em ưu tiên role selector vì bền hơn và bonus là ép code đi qua accessibility" — câu này ăn điểm hơn nhiều so với "em dùng role selector".
- **Có trade-off, đừng tuyệt đối.** "Cách này tốt cho A nhưng đánh đổi B" thể hiện em hiểu sâu, không phải fan cuồng tool.
- **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 nhiều so với "em chưa bao giờ mắc lỗi".
- **Không biết thì nói không biết.** "Em chưa làm case này, nhưng nếu phải làm em sẽ tiếp cận theo hướng..." — trung thực và có hướng giải quyết. Đoán mò bị bắt là mất điểm nặng.

Hỏi ngược lại interviewer:
- *"Team đang gặp pain point gì lớn nhất với test automation hiện tại?"* — thể hiện em quan tâm dự án thật, vừa biết team họ đang ở đâu trong hành trình.
- *"Tỷ lệ flaky test trên CI hiện tại của team khoảng bao nhiêu?"* — câu hỏi này filter team có thật sự đầu tư cho chất lượng test không. Nếu họ không biết hoặc nói "rất nhiều", đó là red flag.
- *"Test coverage của team đang ở mức nào và mục tiêu trong 6 tháng tới?"* — câu này thể hiện em nghĩ về kết quả dài hạn.

Khi live coding (nếu có vòng này):
- Nói lớn suy nghĩ của em khi code. Interviewer quan tâm cách em tư duy hơn là kết quả cuối.
- Bắt đầu bằng câu hỏi clarify yêu cầu. Đừng nhảy vào code ngay.

- Viết skeleton trước, fill detail sau. Đừng cố perfect từ dòng đầu tiên.
- Khi stuck, nói "em đang nghĩ về..." thay vì im lặng. Im lặng dài làm interviewer lo lắng.