30 Câu Hỏi Phỏng Vấn Selenium — Phiên Bản Nói Miệng

> Tài liệu này dành cho vòng phỏng vấn nói: HR screen, technical interview qua call, hoặc trao đổi miệng không có laptop. Câu trả lời viết theo nhịp hội thoại — không có code block dài, không có tên method khô khan. Mỗi câu kèm một 💡 *Mẹo nói* để biết nhấn ý nào, và một câu chốt trademark để gây ấn tượng. Đọc to 2-3 lần là thuộc nhịp.

Phần 1 — Câu hỏi căn bản đóng gói trong tình huống (10 câu)

Câu 1. Nếu PM hỏi vì sao team chọn Selenium thay vì Cypress hoặc Playwright, em giải thích thế nào trong 2 phút?

Đây không phải câu giới thiệu Selenium kiểu Wikipedia — đây là câu test em có biết **trình bày trade-off** với người không kỹ thuật hay không.

Em sẽ bắt đầu bằng câu hỏi ngược: PM quan tâm điều gì nhất — chi phí, tốc độ release, hay rủi ro? Vì câu trả lời phụ thuộc vào ưu tiên của họ.

Nếu PM lo về **rủi ro dài hạn**, em nói: Selenium tồn tại hơn mười lăm năm, được dùng ở mọi enterprise lớn, có Java/C-sharp ecosystem rất mạnh. Khi engineer rời team, người mới tuyển vào biết Selenium ngay — không cần training lại. Cypress hay Playwright còn trẻ, cộng đồng nhỏ hơn ở stack Java.

Nếu PM lo về **tốc độ phát triển**, em thừa nhận thẳng: Cypress và Playwright nhanh hơn cho team JavaScript thuần. Nhưng app của team mình là Java backend, web Angular — Selenium tích hợp tự nhiên với pipeline Maven và CI hiện tại, không cần dựng stack mới.

Nếu PM lo về **cross-browser**, em nhấn vào điểm này: Selenium Grid là chuẩn công nghiệp cho test trên nhiều browser thật. Cypress không hỗ trợ Safari đầy đủ. Đây là deal-breaker nếu app có khách dùng Safari nhiều.

Em sẽ chốt bằng câu: "Tool tốt không tồn tại, chỉ có tool phù hợp với context của team mình."

> 💡 **Mẹo nói:** Đừng nói "Selenium tốt hơn" — đó là câu của người fanboy. Hãy nói trade-off và đặt câu hỏi ngược cho PM. Câu chốt: *"Tool theo nhu cầu, không theo trend."*

Câu 2. Em đang test một dropdown có 50 option và muốn verify option "Hà Nội" có tồn tại. Em viết logic ra sao?

Câu này check em có hiểu sự khác nhau giữa **find một** và **find nhiều** ở mức ứng dụng, không phải định nghĩa.

Em sẽ KHÔNG dùng find một rồi catch exception — đây là anti-pattern phổ biến. Lý do: find một sẽ ném `NoSuchElementException` nếu không thấy, và exception là cách báo lỗi đắt — không phải cách kiểm tra điều kiện.

Cách đúng là **dùng find nhiều**. Find nhiều trả về danh sách — nếu không có gì khớp, danh sách rỗng, không ném exception. Em sẽ find tất cả option trong dropdown, sau đó lọc xem có option nào có text bằng "Hà Nội" không.

Em hay viết theo công thức ba bước: mở dropdown, find list tất cả option, dùng stream filter để kiểm tra có match không. Nếu danh sách kết quả lớn hơn 0, có nghĩa là tồn tại.

Một mẹo nhỏ: với dropdown native HTML thẻ `select`, em dùng class `Select` của Selenium — class này có method tiện như `getOptions` để lấy hết option luôn. Còn dropdown custom dạng div lồng nhau, em phải find element thông thường.

Bẫy hay gặp: dropdown lười tải — chỉ render option khi user mở dropdown ra. Em phải click mở trước, đợi danh sách render, mới find được. Quên bước này thì find ra danh sách rỗng dù option có tồn tại trong code.

> 💡 **Mẹo nói:** Nhấn vào "không dùng exception để kiểm tra tồn tại". Đây là dấu hiệu em biết viết clean code. Câu chốt: *"Dùng find một để chờ, dùng find nhiều để kiểm tra."*

Câu 3. Dev push code mới, locator của em vỡ hàng loạt. Em làm gì trước — yêu cầu dev thêm test ID, hay tự fix bằng XPath?

Đây là câu test em có nghĩ về **bảo trì dài hạn** hay chỉ lo chữa cháy ngắn hạn.

Câu trả lời thẳng của em là: **trao đổi với dev trước, fix tạm bằng XPath sau, không bao giờ chỉ fix mà không nói gì**.

Lý do là: nếu em chỉ tự fix mỗi lần code dev đổi, em đang gánh chi phí của họ. Tuần sau họ đổi lại, em lại fix tiếp. Vòng lặp này không có lối ra, và team sẽ thấy automation là gánh nặng chứ không phải tài sản.

Cách em làm là chia thời gian: dành **30 phút đầu** để fix tạm những test critical đang chặn CI — dùng XPath hoặc CSS selector phức tạp, có comment "TODO: cần data-testid". Sau đó **dành thời gian còn lại** để ping dev qua Slack hoặc tạo ticket: "Page X có 5 element chưa có data-testid, đây là danh sách, làm ơn thêm vào".

Em thường kèm câu giải thích nhẹ nhàng: "data-testid là attribute riêng cho test, dev không cần dùng cho production code, css cũng không cần dùng. Đây là contract giữa dev và QA — đổi code không ảnh hưởng test, đổi test không ảnh hưởng code".

Đa số dev sẽ đồng ý sau khi hiểu, vì nó cũng có lợi cho họ — không bị QA ping lỗi mỗi sprint.

Nếu dev từ chối thẳng, em có hai lựa chọn: nâng lên với team lead, hoặc chấp nhận và viết test với XPath ổn định nhất có thể — ưu tiên match theo attribute ngữ nghĩa như `name`, `aria-label`, không match theo class style.

> 💡 **Mẹo nói:** Câu này check tư duy collaboration của em. Đừng nói "em tự fix thôi" — đó là dấu hiệu engineer thiếu trưởng thành. Câu chốt: *"Test ổn định là trách nhiệm chung của QA và dev, không phải chỉ QA."*

Câu 4. Team em có test flaky 15% vì wait. Em đọc code thì thấy có cả implicit wait và explicit wait trộn lẫn. Em sửa thế nào và giải thích cho team ra sao?

Đây là combo cực kỳ phổ biến và là một trong những nguyên nhân flaky test khó debug nhất. Em đã gặp ít nhất ba dự án có vấn đề này.

Trước hết em giải thích vì sao combo này gây flaky. Implicit wait là cài đặt toàn cục — "mọi find element chờ tối đa X giây". Explicit wait là chờ có điều kiện cụ thể tại một dòng. Khi trộn hai cái, **timeout cộng dồn** không lường được.

Ví dụ em set implicit 10 giây và explicit wait 10 giây cho điều kiện element clickable. Nếu element không bao giờ clickable, em **kỳ vọng** test fail sau 10 giây. Nhưng thực tế nó có thể chờ 20 giây hoặc lâu hơn vì explicit wait đang poll, mỗi lần poll lại trigger implicit wait. Một test fail tốn 20-30 giây thay vì 10 giây, suite 100 test mất thêm cả tiếng.

Cách fix là **chọn một cách và bỏ cách kia**. Em luôn chọn explicit wait. Implicit wait set về 0 toàn cục.

Khi giải thích cho team, em không nói "code các bạn sai". Em nói "em đã đọc tài liệu Selenium chính thức và họ khuyến cáo không trộn — đây là link tài liệu". Em đưa số liệu cụ thể: "trước khi fix, suite flaky 15%, sau khi fix còn 2%". Số liệu thắng tranh luận, không phải ý kiến cá nhân.

Em cũng đề xuất thêm một quy ước trong team: "mọi wait phải dùng explicit, không ai được set implicit wait ở đâu cả". Quy ước này đưa vào code review checklist.

> 💡 **Mẹo nói:** Câu này check cả kiến thức kỹ thuật và kỹ năng thuyết phục. Số liệu trước, ý kiến sau. Câu chốt: *"Implicit wait là chờ mù, explicit wait là chờ có lý do — đừng trộn hai."*

Câu 5. Em được giao một test cũ dùng đầy Thread.sleep. Em refactor ra sao mà không phá test?

Câu này check em có biết **refactor an toàn** hay chỉ biết viết code mới.

Em sẽ KHÔNG xóa hết sleep và chạy thử — đó là cách của người mới. Test có thể đang pass nhờ sleep, xóa đi mất pass mà không biết lý do.

Quy trình em làm theo bốn bước.

Bước một, **đọc context của mỗi sleep**. Sleep đang chờ gì? Chờ element xuất hiện? Chờ animation? Chờ API response? Comment ngay vào code "sleep này đang chờ ABC". Đa số sleep không có comment — em phải đoán từ bối cảnh.

Bước hai, **thay từng sleep một**, không thay tất cả cùng lúc. Mỗi lần thay một sleep bằng explicit wait có điều kiện cụ thể, chạy test 5-10 lần. Nếu pass ổn định, commit. Nếu fail, hoặc điều kiện wait sai, hoặc có sleep thật sự cần thiết em chưa hiểu — quay lại bước một.

Bước ba, **giữ một số ít sleep có lý do**. Có những trường hợp hiếm sleep thật sự cần — chờ CSS animation không có event kết thúc, chờ race condition cố ý. Em giữ những sleep này nhưng thêm comment giải thích rõ ràng: "sleep 300ms vì animation slide-in không có hook, không có cách nào khác".

Bước bốn, **đo kết quả**. Sau refactor, suite chạy nhanh hơn bao nhiêu, tỉ lệ pass ổn định bao nhiêu. Báo cáo cho team thấy giá trị của việc refactor.

Em từng refactor một suite có hơn 200 sleep, mất 2 tuần. Suite giảm từ 38 phút xuống 14 phút, fail rate giảm từ 8% xuống dưới 1%. Đây là loại bằng chứng mạnh nhất để biện hộ cho việc dành thời gian refactor.

> 💡 **Mẹo nói:** Đừng nói "em xóa hết sleep" — quá đơn giản và nguy hiểm. Câu chốt: *"Sleep là che bug, không phải fix bug — nhưng xóa sleep cũng không phải xóa tất cả một lúc."*

Câu 6. Em mở một test có sẵn của đồng nghiệp, find element và lưu vào biến, vài dòng sau click thì lỗi stale element. Em sửa thế nào?

Câu này check em hiểu **DOM lifecycle** ở mức nào.

Trước hết em giải thích nguyên nhân. Stale element xảy ra khi reference em đang giữ không còn trỏ đến element thật trong DOM nữa. Có hai nguyên nhân chính: trang reload hoặc navigate, hoặc framework như React/Vue re-render component đó.

Code lỗi điển hình là: find element và lưu vào biến, sau đó làm hành động khác (như scroll, hoặc click element khác trigger re-render), rồi mới quay lại dùng biến cũ. Khi đó reference đã chết.

Có ba cách sửa, từ tốt nhất đến tệ nhất.

Cách tốt nhất: **không lưu reference lâu**. Find ngay trước khi dùng. Code dài hơn một chút nhưng an toàn. Hầu hết case em chọn cách này.

Cách thứ hai: **dùng retry helper**. Viết hàm wrapper bắt exception stale, find lại và thử lại tối đa 2-3 lần. Cách này dùng khi không tránh được phải lưu reference vì lý do performance.

Cách thứ ba: **dùng explicit wait với condition tự định nghĩa**. Wait sẽ tự retry mỗi vài trăm milli giây, nuốt stale exception trong khi chờ. Đây là cách Selenium đã làm sẵn trong fluent wait.

Cách em hay khuyên đồng nghiệp khi review code: nếu một biến `WebElement` xuất hiện trong code, kéo dài hơn 5-10 dòng và có hành động trigger DOM change giữa chừng, là dấu hiệu cần refactor. Đa số stale

element là do giữ biến quá lâu, không phải do Selenium yếu.

> 💡 **Mẹo nói:** Nhấn vào "không lưu reference lâu" như quy tắc chính. Câu chốt: *"Stale element không phải bug Selenium — đó là bug pattern của em khi giữ tham chiếu lâu hơn cần thiết."*

Câu 7. Em viết test cho một page có form đăng ký 15 trường. Em tổ chức code như thế nào để không bị thành "đống lộn xộn"?

Đây là câu Page Object Model nhưng đóng gói trong tình huống thật — không phải hỏi định nghĩa POM.

Em sẽ tách thành ba lớp.

Lớp một, **page class chính** — `RegistrationPage`. Class này có locator cho 15 element và method đại diện cho hành động người dùng làm trên trang. Quan trọng: không nên có 15 method setter kiểu `setFirstName`, `setLastName`. Thay vào đó có một method `fillForm` nhận object data, fill toàn bộ form trong một lần gọi.

Lớp hai, **data builder class** — `RegistrationDataBuilder`. Đây là pattern xây dựng data động: "tôi muốn data hợp lệ" hoặc "tôi muốn data thiếu email" — builder trả về object data tương ứng. Test không hard-code 15 trường vào, chỉ gọi builder.

Lớp ba, **test case** — chỉ chứa logic test, không có locator, không có sendKeys. Test đọc gần như tiếng Anh: tạo data hợp lệ, mở trang đăng ký, fill form với data đó, click submit, verify chuyển sang trang welcome.

Lợi ích của cách này: khi UI đổi — ví dụ thêm trường thứ 16, hoặc đổi locator của 5 trường — em chỉ sửa trong `RegistrationPage`. 50 test case đang dùng nó không phải đụng tay vào.

Khi đối tác hỏi data mới — ví dụ "test với user đã verified" — em thêm method vào builder. Test case dùng builder mới chỉ đổi một dòng.

Đây là cách POM thật sự sống được trong dự án dài hạn. POM không phải chỉ là tách locator ra một class — đó là cấp độ một. Cấp độ trưởng thành là tách cả data và hành vi.

> 💡 **Mẹo nói:** Đừng đọc định nghĩa POM. Hãy mô tả ba lớp em chia và lý do. Câu chốt: *"Không có POM thì test pass là may, không phải skill. Có POM mà không có data builder thì pass cũng chỉ một mức skill thấp hơn."*

Câu 8. Em test một trang SPA của React, sau khi click nút thì element em đang giữ "biến mất". Selenium báo lỗi gì và em xử lý ra sao?

Câu này test em có hiểu **SPA và Selenium tương tác ra sao**.

Selenium sẽ báo một trong hai lỗi: `StaleElementReferenceException` nếu em đang giữ reference cũ, hoặc `NoSuchElementException` nếu em đang find lại nhưng tìm bằng locator cũ.

Lý do bản chất: React re-render component sau khi state thay đổi. Element cũ bị unmount khỏi DOM, element mới được mount với cùng vai trò nhưng có thể khác attribute, khác vị trí trong cây DOM.

Có ba kỹ thuật em hay dùng để handle SPA.

Kỹ thuật một, **chờ state mới ổn định, không chờ element cũ biến mất**. Sau khi click trigger re-render, em chờ một element đại diện cho state mới — ví dụ một message "Saved successfully" hoặc một trang khác. Khi element đó xuất hiện, em biết React đã render xong.

Kỹ thuật hai, **dùng data-testid ổn định qua các state**. Yêu cầu dev đặt data-testid không đổi cho cùng một logical element, dù state đổi như nào. Ví dụ nút "Submit" và sau khi click trở thành nút "Submitting..." — vẫn cùng data-testid "submit-btn".

Kỹ thuật ba, **không tin vào timing của re-render**. Đừng sleep cố định. Đừng giả định "React render trong 100ms". Render time phụ thuộc vào props phức tạp, state, network. Luôn dùng explicit wait có điều kiện.

Có một bẫy đặc biệt với React: animation transition. Khi component vừa mount, nó có thể trong giai đoạn fade-in — không click được. Em chờ điều kiện `elementToBeClickable`, không chỉ `visibility`.

> 💡 **Mẹo nói:** SPA là chỗ phân biệt người làm test web cổ điển và người làm test web hiện đại. Câu chốt: *"Trong SPA, element không bao giờ là vĩnh viễn — đừng tin nó tồn tại quá vài giây."*

Câu 9. Em phỏng vấn vị trí junior cho team, ứng viên nói "em dùng cả ID, name, class, XPath, CSS selector". Em hỏi follow-up gì?

Câu này test em có biết phân biệt **người biết liệt kê** và **người biết lựa chọn**.

Em sẽ hỏi ba câu follow-up theo thứ tự độ khó.

Câu một: "Nếu em có ID, em có dùng XPath không? Vì sao?". Câu này check ứng viên có biết ưu tiên locator hay không. Câu trả lời tốt: ID là lựa chọn đầu, vì nó duy nhất, ổn định, và Selenium tối ưu cho ID. XPath chỉ dùng khi không có lựa chọn nào khác.

Câu hai: "Em từng gặp class name dạng `btn-primary-large-active-with-icon-v2` chưa? Em có dùng làm locator không?". Câu này check ứng viên có biết phân biệt **class ngữ nghĩa** và **class style**. Class kiểu trên là style — đổi mỗi lần designer chỉnh CSS, không nên dùng. Class ngữ nghĩa như `error-message` thì OK hơn.

Câu ba: "Em viết XPath để tìm thẻ cha của một element có chữ ABC chưa?". Câu này test ứng viên có biết XPath đi ngược cây DOM — điểm mạnh duy nhất XPath hơn CSS. Nếu không biết, có nghĩa là ứng viên chưa thực sự gặp tình huống cần XPath, chỉ dùng XPath theo quán tính.

Sau ba câu này, em có hình dung khá rõ ứng viên là người hiểu nguyên tắc hay chỉ học công cụ.

Một câu hỏi bonus em hay hỏi senior: "Lần cuối em yêu cầu dev thêm data-testid là khi nào?". Câu trả lời "chưa bao giờ" là red flag — engineer test trưởng thành phải hiểu data-testid là pattern chuẩn ngành.

> 💡 **Mẹo nói:** Câu này test khả năng **phỏng vấn người khác** — interviewer hay hỏi senior. Câu chốt: *"Người biết liệt kê locator chưa chắc biết chọn locator."*

Câu 10. Em chuẩn bị bàn giao test suite cho team mới. Em viết tài liệu gì, và quan trọng nhất là viết gì để 6 tháng sau team mới không "vứt đi viết lại"?

Câu này check em có nghĩ về **sustainability** của automation hay không.

Em từng nhận bàn giao một suite và sau ba tháng team em phải viết lại từ đầu — vì người cũ không viết tài liệu, locator không có comment, không có ai biết "test này test cái gì". Em không muốn ai khác trải qua điều đó.

Tài liệu em viết có bốn phần.

Phần một, **README technical**: cài đặt môi trường, chạy test ra sao, cấu trúc thư mục, các biến môi trường cần set. Phần này có sẵn ở 90% dự án, không có gì đặc biệt.

Phần hai, **test inventory**: bảng liệt kê tất cả test case, mỗi case có mô tả ngắn gọn (1-2 câu) về kịch bản, feature nó cover, mức ưu tiên. Khi team mới cần fix nhanh, họ biết test nào quan trọng để giữ và test nào có thể tạm bỏ.

Phần ba, **kiến trúc và quyết định**: vì sao em chọn POM thay vì Page Factory, vì sao team dùng JUnit 5 thay vì TestNG, vì sao test data sinh qua API thay vì hard-code. Phần này quan trọng nhất — nó giúp team mới hiểu **lý do**, không chỉ **kết quả**. Nếu họ thấy lý do còn đúng, họ giữ. Nếu không còn đúng, họ refactor có ý thức, không phải vứt đi.

Phần bốn, **các quirk đã biết**: những test thỉnh thoảng flaky vì lý do nào, những workaround đang dùng tạm thời, những TODO chưa giải quyết. Nói trung thực — đừng giấu.

Em đề xuất thêm một buổi **knowledge transfer trực tiếp**: walkthrough code trên màn hình, trả lời câu hỏi. Tài liệu không thay thế được conversation.

> 💡 **Mẹo nói:** Câu này hay được hỏi cho vị trí senior. Nhấn vào "lý do của các quyết định", không chỉ "cách dùng". Câu chốt: *"Tài liệu tốt giải thích vì sao, không chỉ làm thế nào — vì 'cách' sẽ lỗi thời, 'vì sao' vẫn còn nguyên giá trị."*

Phần 2 — Câu hỏi tình huống và thực chiến (10 câu)

Câu 11. Em đã từng gặp test flaky. Em đã debug và fix nó như thế nào?

Câu này là câu phân biệt rõ nhất giữa người làm 3 tháng và người làm 3 năm. Em sẽ kể một case thật.

Có lần em có một test login chạy trên CI fail khoảng 20% — chạy local thì pass mọi lần. Bước đầu em không vội fix, mà thu thập bằng chứng. Em bật **screenshot khi fail**, bật **video record**, bật **log của console browser**. Sau hai chục lần chạy trên CI, em có đủ dữ liệu.

Em phát hiện ra: trên CI server, khi load trang dashboard sau login, có một call API mất khoảng 1,8 giây thay vì 200 mili giây ở local. Trong khoảng đó, nút "Tạo mới" tồn tại trong DOM nhưng bị overlay che. Test em đang chờ element xuất hiện rồi click — và click trúng overlay.

Fix là đổi điều kiện chờ từ "**xuất hiện**" sang "**clickable**" — clickable kiểm tra cả visible và enabled. Sau fix, fail rate giảm về 0 sau 200 lần chạy.

Bài học em rút ra là: **flaky test luôn có nguyên nhân thật**. Đừng dùng retry để che, đừng tăng timeout cho qua chuyện. Em phải hiểu vì sao nó flaky, vì nó đang nói cho em biết một sự thật về app.

> 💡 **Mẹo nói:** Kể case thật theo công thức: triệu chứng → bằng chứng → nguyên nhân gốc → fix → kết quả đo được. Câu chốt: *"Flaky test không phải để retry, mà để học."*

Câu 12. Test của em chạy 45 phút và team than chậm. Em tối ưu từ đâu?

Em sẽ tiếp cận theo ba lớp.

Lớp một là đo trước khi tối ưu. Em cần biết test nào tốn lâu nhất. Em xuất report time của từng test ra file, sắp xếp giảm dần, và sẽ thấy quy luật 80-20: 20% test tốn 80% thời gian. Đây là chỗ em tập trung.

Lớp hai là chạy song song. Đây là đòn bẩy lớn nhất. Nếu suite hiện chạy tuần tự trên 1 luồng, em chia thành 5 luồng song song — thời gian giảm gần năm lần. Selenium hỗ trợ qua Grid hoặc qua tích hợp với framework như TestNG hay JUnit 5. Lưu ý: test phải **độc lập** — không chia sẻ state, không phụ thuộc thứ tự.

Lớp ba là cắt việc không cần thiết. Em rà soát: test nào lặp lại setup giống nhau — có thể gom vào fixture chia sẻ. Test nào đăng nhập qua UI lặp đi lặp lại — chuyển sang đăng nhập qua API rồi lưu cookie. Test nào navigate qua nhiều trang trung gian không cần thiết — đi thẳng đến trang cần test bằng URL.

Sau ba lớp đó, em từng tối ưu một suite 45 phút xuống còn 8 phút mà không giảm coverage.

> 💡 **Mẹo nói:** Đừng nói chung chung "chạy song song". Hãy có con số: "5 luồng giảm 5 lần", "45 phút xuống 8 phút". Câu chốt: *"Tối ưu test là tối ưu cycle feedback cho cả team, không phải khoe kỹ thuật."*

Câu 13. Em xử lý alert, popup và confirm dialog của trình duyệt như thế nào?

Đầu tiên cần phân biệt hai loại popup khác nhau:

Alert của trình duyệt — kiểu hộp thoại `alert`, `confirm`, `prompt` mà JavaScript bật lên. Loại này **không nằm trong DOM** — em không thể find bằng locator. Selenium có cơ chế riêng để xử lý: chuyển driver sang context của alert, đọc text, chấp nhận hoặc từ chối, gửi text nếu là prompt.

Modal hoặc popup do app vẽ — đây thật ra là HTML bình thường nằm trong DOM. Em xử lý như mọi element khác — find rồi tương tác.

Bẫy thường gặp là nhầm hai loại. Em thấy bạn mới hay cố find một alert của trình duyệt bằng XPath, không hiểu vì sao luôn fail. Cách phân biệt: nếu dialog đó kéo ra ngoài cửa sổ trình duyệt được, hoặc style không khớp với app, gần như chắc là alert hệ thống.

Một mẹo nhỏ: với alert hệ thống bất ngờ xuất hiện và làm test fail, em hay thêm một hook chạy trước mỗi test, kiểm tra xem có alert nào đang treo không và tự đóng đi.

> 💡 **Mẹo nói:** Câu này hỏi để check em có hiểu khái niệm DOM hay không. Nhấn vào điểm "alert hệ thống không nằm trong DOM". Câu chốt: *"Alert hệ thống không phải HTML, đừng cố tìm bằng locator."*

Câu 14. Có một test đôi khi fail vì element bị element khác che lấp. Em xử lý ra sao?

Đây là một biến thể của câu flaky test, nhưng cụ thể hơn.

Triệu chứng kinh điển: test cố click một nút, nhận lỗi đại loại "element click intercepted" hoặc "another element would receive the click". Nghĩa là click đến đúng tọa độ rồi nhưng một element khác đang đè lên trên.

Có ba nguyên nhân hay gặp. Một là **modal hoặc overlay** đang hiển thị mà em không để ý — ví dụ một banner cookie consent. Hai là **animation đang chạy** — element đích đang trượt vào và chưa ổn định vị trí. Ba là **sticky header hoặc footer** — em scroll xuống nhưng element bị header đè.

Cách xử lý theo thứ tự ưu tiên: thứ nhất, **chờ overlay biến mất** rồi mới click — đây là cách đúng nhất. Thứ hai, **scroll element vào giữa màn hình** trước khi click — tránh bị header đè. Thứ ba, nếu không còn cách nào, **click qua JavaScript** — đây là vũ khí cuối cùng vì nó bypass cả layer hiển thị, có thể che bug thật.

Em không bao giờ dùng JavaScript click làm giải pháp đầu tiên — chỉ dùng khi đã chắc UI thật sự không cho click theo cách thông thường.

> 💡 **Mẹo nói:** Nhấn vào "JS click là vũ khí cuối, không phải lựa chọn đầu". Đây là red flag với interviewer nếu em coi nó là solution mặc định. Câu chốt: *"Nếu user không click được, test cũng*

> *không nên click được."*

Câu 15. Khi nào em chọn Selenium thay vì Cypress hoặc Playwright?

Câu này hay được dùng để check em có hiểu trade-off hay không — đừng trả lời theo kiểu fanboy.

Em chọn **Selenium** khi: thứ nhất, app cần test trên **nhiều trình duyệt** — đặc biệt là Safari và các trình duyệt cũ — Cypress không hỗ trợ Safari đầy đủ. Thứ hai, dự án đã có **infrastructure Selenium Grid sẵn**, đầu tư đó đáng giữ. Thứ ba, team viết test bằng **Java hoặc C-sharp** — Cypress chỉ JavaScript, Playwright hỗ trợ nhưng cộng đồng Java/C-sharp nhỏ hơn nhiều. Thứ tư, cần test **app có nhiều tab, nhiều cửa sổ, nhiều iframe phức tạp** — Selenium xử lý tốt.

Em chọn **Cypress** khi: app là single-page application JavaScript thuần, team là dev front-end muốn tự viết test, ưu tiên trải nghiệm developer trên trải nghiệm cross-browser.

Em chọn **Playwright** khi: bắt đầu project mới năm 2024 trở đi, cần auto-wait thông minh và parallel mạnh ngay từ thiết kế, có thể chấp nhận học stack mới.

Không có lựa chọn "tốt nhất tuyệt đối" — chỉ có "phù hợp nhất với context của em".

> 💡 **Mẹo nói:** Đây là câu test "trí khôn ngữ cảnh". Tránh nói "Selenium tốt nhất" hay "Playwright tốt hơn". Hãy nói trade-off. Câu chốt: *"Tool theo nhu cầu, không theo trend."*

Câu 16. Em test responsive design — chạy trên desktop, tablet, mobile — em làm thế nào với Selenium?

Có ba cách, từ đơn giản đến phức tạp.

Cách một là resize cửa sổ trình duyệt xuống kích thước cần test. Selenium cho phép set kích thước window. Cách này nhanh, đơn giản, nhưng chỉ test được layout — không bắt được các đặc thù của thiết bị thật như touch event, user agent.

Cách hai là dùng device emulation của Chrome. Chrome DevTools có sẵn các profile cho iPhone, iPad, Android. Selenium hỗ trợ truyền các options này khi khởi tạo Chrome. Cách này tốt hơn cách một vì có cả user agent và viewport đúng.

Cách ba là test trên thiết bị thật qua các dịch vụ như BrowserStack, Sauce Labs hoặc Selenium Grid với thiết bị mobile. Đây là cách chính xác nhất nhưng tốn tiền và chậm.

Quan điểm em là: dùng cách một và hai cho **đa số test trong CI**, và để dành cách ba cho **bộ smoke test critical** chạy ít hơn — ví dụ trước mỗi release.

> 💡 **Mẹo nói:** Đừng nói "thì set viewport thôi". Hãy phân ba cấp độ và mục đích từng cấp. Câu chốt: *"Test desktop để bắt regression, test thiết bị thật để bắt thực tế."*

Câu 17. Test cần upload và download file. Selenium xử lý hai việc này ra sao?

Hai việc này hoàn toàn khác nhau về độ khó.

Upload file thì dễ. Nếu trang web có thẻ `input` kiểu `file`, em chỉ cần truyền đường dẫn tuyệt đối của file vào bằng method `sendKeys`. Selenium sẽ "gõ" đường dẫn đó vào — trình duyệt hiểu và treat như user đã chọn file qua dialog. Bẫy hay gặp là một số UI hiện đại che cái input thật bằng button đẹp — em đừng click vào button đó, hãy find thẳng cái input bị ẩn.

Download file thì khó hơn nhiều vì download dialog của trình duyệt **không nằm trong DOM** — Selenium không tương tác được. Cách xử lý là **cấu hình trình duyệt** trước khi khởi tạo: tắt cảnh báo download, đặt thư mục download cố định. Sau khi trigger download trong test, em kiểm tra **file có xuất hiện trong thư mục đó** không, kích thước có đúng không, nội dung có đúng không nếu cần.

Một biến thể nâng cao: thay vì download qua trình duyệt, lấy URL của file rồi tải bằng HTTP client trong code test — nhanh hơn và không phụ thuộc cấu hình trình duyệt. Lựa chọn này tốt khi em chỉ cần verify nội dung chứ không quan tâm UI download.

> 💡 **Mẹo nói:** Phân biệt rõ "upload trong DOM, download ngoài DOM". Đây là điểm phân biệt người đã từng đụng vào với người chưa. Câu chốt: *"Upload là sendKeys, download là kiểm tra file system."*

Câu 18. Test em chạy được trên máy local nhưng fail trên CI. Em debug từ đâu?

Đây là tình huống quá quen thuộc và có một checklist em luôn chạy qua.

Đầu tiên là kích thước màn hình. CI thường chạy headless với kích thước mặc định nhỏ — ví dụ 1024x768 — trong khi local em mở full HD. Element nằm dưới fold sẽ không click được nếu không scroll. Em set explicit window size ngay khi khởi tạo driver.

Thứ hai là tốc độ chạy. CI thường chậm hơn local 2-3 lần vì shared resource. Wait đang ngắn trên local có thể không đủ trên CI. Em xem lại các timeout và explicit wait.

Thứ ba là biến môi trường. URL, credential, feature flag — có thể local trỏ về dev, CI trỏ về staging với data khác. Em check kỹ.

Thứ tư là chế độ headless. Một số element render khác giữa headless và headed mode trong Chrome — đặc biệt là hover state. Em chạy local ở headless mode để reproduce.

Thứ năm là artifact. Em bật screenshot, video, HAR file trên CI. Xem screenshot tại lúc fail thường giải đáp 70% case.

Quy tắc cuối: **đừng debug bằng cách thay đổi test rồi push lên CI để xem**. Vòng feedback ấy quá chậm. Hãy reproduce local trước.

> 💡 **Mẹo nói:** Có một checklist rõ ràng. Câu chốt: *"Trước khi sửa code, hãy nhìn screenshot. Screenshot không nói dối."*

Câu 19. Em quản lý test data như thế nào? Hard-code, file, hay database?

Câu này check em có nghĩ về dài hạn không.

Hard-code trong test là cách của người mới. Có thể chấp nhận với một-hai test nhỏ, nhanh. Nhưng khi suite lớn lên, một thay đổi nhỏ về data đòi hỏi sửa hàng chục file.

Externalize ra file — JSON, YAML, CSV — là bước nâng cấp đầu tiên. Test đọc data từ file, em chỉ cần sửa file. Đây là điểm đa số dự án dừng lại và đủ tốt.

Sinh data động qua API là bước nâng cấp tiếp theo. Trước mỗi test, em gọi API của app để tạo user mới, tạo order mới — chỉ tồn tại cho test này. Lợi ích lớn: test **độc lập hoàn toàn**, có thể chạy song song mà không xung đột data. Sau test, có thể cleanup cũng qua API.

Test data builder pattern là tinh thần phía sau cách thứ ba: viết code helper kiểu "tôi muốn một user, đã verified, không có order nào" — code sẽ trả về user ấy. Test nhìn vào builder thay vì lo data đến từ đâu.

Em không khuyến khích đọc data từ database trực tiếp, vì gắn test vào schema database — schema đổi là test vỡ. API là contract ổn định hơn.

> 💡 **Mẹo nói:** Trình bày theo bậc thang trưởng thành: hard-code → file → API → builder pattern. Câu chốt: *"Test data tốt là test data sinh ra cho riêng test đó."*

Câu 20. Em report kết quả test cho team không chuyên kỹ thuật như thế nào?

Đây là câu hay được hỏi cho vị trí senior — automation không phải chỉ là code, mà là giao tiếp với stakeholder.

Em không gửi log thô. Log dành cho dev, không dành cho PM hay business.

Em dùng **báo cáo HTML** từ các thư viện như Allure hoặc Extent Report. Báo cáo này có: tổng quan pass/fail dạng biểu đồ, mỗi test case có step rõ ràng, screenshot tại lúc fail, video nếu cần, và group theo feature.

Quy trình em hay làm: sau mỗi lần CI chạy đêm, em gửi **một tin ngắn vào kênh Slack chung** của team — kiểu "tối qua chạy 234 test, 230 pass, 4 fail. Báo cáo chi tiết ở link này". Người quan tâm click vào xem, không quan tâm vẫn nắm tình hình.

Trước mỗi release, em làm **slide tóm tắt** — không phải báo cáo kỹ thuật mà là câu trả lời cho câu hỏi của PM: "có release được không?". Em chốt: feature A xanh, feature B có 2 vấn đề và mức độ ảnh hưởng, kiến nghị go hay no-go.

Bài học em rút ra là: **người không kỹ thuật cần kết luận, không cần dữ liệu**. Em đưa kết luận trước, dữ liệu để hỗ trợ kết luận đó.

> 💡 **Mẹo nói:** Nhấn vào communication, không phải tool. Câu chốt: *"PM cần biết release được hay không, không cần biết stack trace."*

Phần 3 — Câu hỏi nâng cao (10 câu)

Câu 21. Selenium 4 có gì mới đáng chú ý so với Selenium 3?

Đây là câu kiểm tra em có cập nhật kiến thức hay không.

Điểm lớn nhất là **chuyển hoàn toàn sang W3C WebDriver protocol**. Selenium 3 vẫn dùng JSON Wire Protocol cũ. W3C là chuẩn web chính thức, ổn định hơn, đặc biệt với các trình duyệt mới.

Thứ hai là **relative locators** — locator kiểu "tìm element nằm bên phải element này", "nằm phía dưới element kia". Tính năng này hữu ích khi test layout, ví dụ "kiểm tra nút Save có nằm cạnh nút Cancel không". Em ít dùng cho test thông thường vì kém ổn định khi layout thay đổi, nhưng cho visual test thì hay.

Thứ ba là **Chrome DevTools Protocol integration**. Selenium 4 cho phép em can thiệp vào trình duyệt sâu hơn: chặn request mạng, giả lập điều kiện mạng chậm, đọc console log, đo performance. Đây là bước tiến lớn vì trước đây phải dùng tool ngoài.

Thứ tư là **Selenium Grid được viết lại hoàn toàn**, hỗ trợ Docker tốt hơn, scale ngang dễ hơn, có giao diện quản lý tốt hơn.

Thứ năm là **deprecate desired capabilities** thay bằng options class — code rõ ràng hơn.

> 💡 **Mẹo nói:** Đừng liệt kê tất cả tính năng — chỉ ba điểm em thực sự dùng. Câu chốt: *"Selenium 4 là Selenium 3 trưởng thành, không phải sản phẩm khác."*

Câu 22. Em viết test cho iframe — Selenium xử lý ra sao và bẫy thường gặp?

Iframe là frame nhúng — một trang HTML lồng trong trang khác. Đây là context tách biệt — Selenium mặc định nhìn vào DOM của trang chính, không thấy gì bên trong iframe.

Để tương tác với element trong iframe, em phải **chuyển context** vào iframe đó trước, làm việc xong rồi **chuyển ra ngoài**. Selenium có method để switch frame theo ID, tên, index, hoặc element của frame.

Có ba bẫy hay gặp.

Một là **quên switch back** ra ngoài sau khi xong việc với iframe. Test tiếp theo cố tìm element trong main frame và fail với lỗi không thấy — em ngơ ngác không hiểu vì sao.

Hai là **nested iframe** — iframe trong iframe. Em phải switch nhiều cấp. Sai một bước là lạc đường.

Ba là **iframe load chậm hơn trang chính**. Em vừa thấy iframe element xuất hiện trong DOM nhưng nội dung bên trong chưa load. Phải chờ explicit cả iframe sẵn sàng để switch vào, rồi sau khi switch vào phải chờ element bên trong.

Mẹo của em: dùng try-finally để đảm bảo luôn switch back về default content kể cả khi có exception. Đây là pattern phải có nếu test có iframe.

> 💡 **Mẹo nói:** Nhấn vào ba bẫy cụ thể. Câu chốt: *"Iframe là một thế giới riêng, đi vào thì nhớ đường ra."*

Câu 23. Em viết test có nhiều tab hoặc cửa sổ — workflow ra sao?

Mỗi tab và cửa sổ trong Selenium đều có một **window handle** — như mã định danh duy nhất.

Khi mới khởi tạo, em có một handle. Khi test mở tab mới hoặc cửa sổ mới — qua click vào link mở tab, hoặc qua JavaScript — Selenium **không tự chuyển sang tab mới**. Em phải chủ động liệt kê tất cả handle hiện có và switch sang handle mới.

Pattern em hay dùng: **lưu handle gốc** trước khi mở tab mới, thực hiện hành động mở tab, sau đó lấy danh sách tất cả handle, lọc ra handle không trùng handle gốc — đó là tab mới. Switch vào, làm việc, đóng nếu cần, rồi switch về handle gốc.

Bẫy hay gặp: nếu app mở popup không lường trước — ví dụ một dialog ad — em sẽ thấy có thêm handle lạ và logic switch bị nhầm. Em hay viết helper hàm "tìm handle theo URL hoặc theo title" thay vì dựa vào thứ tự handle trong list.

Một mẹo nhỏ: từ Selenium 4, có method tiện lợi để mở tab mới hoặc cửa sổ mới trực tiếp từ driver, không cần qua JavaScript. Code sạch hơn nhiều.

> 💡 **Mẹo nói:** Nhấn vào "Selenium không tự switch — em phải chủ động". Câu chốt: *"Mỗi tab là một handle, mỗi handle là một thế giới."*

Câu 24. Em handle cookie và session — đăng nhập một lần dùng cho nhiều test?

Đây là một trong những kỹ thuật tối ưu suite mạnh nhất.

Nếu mỗi test em đăng nhập qua UI — gõ email, gõ password, chờ chuyển trang — em đang lãng phí 5-10 giây mỗi test cho việc lặp lại. Suite một trăm test là mười phút thuần lãng phí.

Cách tối ưu là **đăng nhập một lần, lưu cookie và session, dùng cho mọi test**.

Quy trình: trước khi suite chạy, có một bước **setup global**. Bước này mở trình duyệt, đăng nhập, lấy ra toàn bộ cookie của domain, lưu vào file. Sau đó mỗi test khởi động chỉ cần **load cookie từ file vào driver**, không cần đăng nhập lại. Đẩy thẳng đến trang cần test.

Cách cao cấp hơn: **bypass UI hoàn toàn**. Đăng nhập qua API endpoint, lấy về token hoặc session, set vào cookie hoặc localStorage của trình duyệt. Cách này nhanh hơn nữa vì không cần render trang login.

Lưu ý quan trọng: cookie có **scope domain** và **time-to-live**. Em cần lưu cookie cho đúng domain test, và nếu lưu file thì cần biết khi nào expire để refresh.

Em đã từng dùng kỹ thuật này giảm thời gian suite của một dự án từ 25 phút xuống 9 phút mà không bỏ test nào.

> 💡 **Mẹo nói:** Nhấn vào "đăng nhập qua UI một lần, dùng qua API mãi". Có con số càng tốt. Câu chốt: *"Đăng nhập là việc làm một lần, không phải nghi thức mở đầu mỗi test."*

Câu 25. Em làm cross-browser testing như thế nào? Có gì khác giữa Chrome, Firefox, Safari, Edge?

Cross-browser là một trong những giá trị cốt lõi của Selenium — không phải tool nào cũng làm được tốt.

Về **kỹ thuật setup**, em cấu hình từng driver tương ứng: ChromeDriver cho Chrome, GeckoDriver cho Firefox, SafariDriver cho Safari trên Mac, EdgeDriver cho Edge. Từ Selenium 4, có **Selenium Manager** tự động tải đúng phiên bản driver — em không cần quản lý thủ công nữa.

Về **chạy song song nhiều trình duyệt**, em dùng Selenium Grid hoặc dịch vụ cloud như BrowserStack. Cùng một test case, parametrize theo browser, chạy đồng thời trên 4 browser khác nhau.

Về **khác biệt thực tế giữa các trình duyệt**:

Chrome ổn định nhất, debug dễ nhất, có DevTools Protocol mạnh. Đây là browser em mặc định.

Firefox đôi khi có hành vi khác Chrome ở các sự kiện bàn phím và file upload. Test pass trên Chrome có thể fail trên Firefox vì lý do nhỏ.

Safari là khó nhất. SafariDriver yêu cầu kích hoạt manual trong cài đặt. Safari handle iframe và window khác Chrome ở vài chỗ. Test phải tolerant với khác biệt timing.

Edge gần như Chrome vì cùng nền Chromium — gần như không có vấn đề khác biệt.

Quan điểm em: **không phải mọi test phải chạy mọi browser**. Smoke test critical chạy đa browser. Test chi tiết chạy Chrome là đủ.

> 💡 **Mẹo nói:** Đừng nói "y hệt nhau". Hãy nói khác biệt thực tế của từng browser. Câu chốt: *"Cross-browser không phải copy-paste config, mà là chấp nhận từng browser có cá tính riêng."*

Câu 26. Em tích hợp Selenium vào CI/CD pipeline — workflow tổng quát ra sao?

Em sẽ kể workflow chuẩn của team em.

Bước một là chuẩn bị môi trường. CI runner cần có browser cài sẵn — nếu chạy headless thì Chrome headless là đủ. Em thường dùng Docker image có sẵn Selenium và browser — chỉ một dòng config là xong. Có image chuẩn của Selenium tên là `selenium/standalone-chrome` em hay dùng.

Bước hai là khởi tạo Grid nếu cần parallel. Với dự án nhỏ, không cần Grid, chạy local trong runner. Với dự án lớn, em dựng Grid với hub và nhiều node, chạy song song giảm thời gian.

Bước ba là chạy test theo trigger. Em chia ba mức: **smoke test** chạy mỗi pull request — nhanh, dưới 5 phút, chặn merge nếu fail. **Regression test** chạy mỗi đêm trên branch chính — lâu hơn, có thể 30-45 phút. **Full suite** chạy trước release — bao gồm cả cross-browser.

Bước bốn là xử lý kết quả. Suite kết thúc, em xuất báo cáo HTML, upload lên CI artifact, gửi summary vào Slack. Fail thì có link trực tiếp xem screenshot và log.

Bước năm là retry chiến lược. Em set retry tối đa **một lần** cho test fail — chỉ một, không hơn. Nhiều hơn là che flaky, không phải chấp nhận flaky.

Stack em quen: GitHub Actions hoặc Jenkins cho orchestration, Docker cho môi trường, Allure cho báo cáo, Slack cho notification.

> 💡 **Mẹo nói:** Trình bày theo bước có thứ tự. Câu chốt: *"CI/CD chạy test không phải để có cờ xanh, mà để có thông tin sớm nhất khi có cờ đỏ."*

Câu 27. Code review test automation — em chú ý gì khi review PR của đồng nghiệp?

Em có một checklist trong đầu khi review.

Một là tên test có mô tả ý định không. Tên kiểu "test 01" hoặc "login test" là red flag. Tên tốt nói rõ kịch bản: "user đăng nhập với email không tồn tại thấy thông báo lỗi".

Hai là test có độc lập không. Em đọc xem test có phụ thuộc test khác chạy trước nó không. Phụ thuộc là cấm — phải refactor.

Ba là có Thread.sleep hay wait cứng không. Một dòng sleep là một câu hỏi cần giải trình. Đa số sleep nên thay bằng explicit wait có điều kiện cụ thể.

Bốn là locator có ổn không. Em check XPath dài, class name dạng style — đề nghị thay bằng data-testid hoặc CSS selector ngắn hơn.

Năm là có hard-code data không. URL, email, mật khẩu trong code là red flag. Phải externalize.

Sáu là cleanup ra sao. Test tạo data thì có xóa sau khi xong không. Quên cleanup là nguồn của flaky test sau này.

Bảy là assertion có rõ không. Một test một intent — đừng assert mười thứ lung tung trong một test.

Tám là đọc xem test có nhanh không. Test ba mươi giây trở lên là cần đánh giá lại — có thể tối ưu bằng API setup hoặc cookie injection.

Quy tắc cuối: **em review test như review code production**. Test cũng là code, cũng có bug, cũng cần bảo trì.

> 💡 **Mẹo nói:** Có một checklist rõ ràng. Câu chốt: *"Test là code, code review test phải khắt khe như review code chính."*

Câu 28. Em đo coverage của test automation như thế nào? Coverage cao có nghĩa là chất lượng cao không?

Đây là câu test em có hiểu khái niệm coverage không.

Trước hết, coverage trong automation web có hai cấp:

Code coverage là phần trăm dòng code được test đi qua. Đo qua tool như JaCoCo hay Istanbul. Đa số dùng cho unit test.

Feature coverage là phần trăm tính năng của app có test automation cover. Đo bằng cách map test case về test plan hoặc feature list.

Cho automation UI Selenium, em quan tâm feature coverage hơn code coverage. Code coverage của UI test thấp vì test UI đi qua nhiều layer.

Trả lời câu thứ hai: **coverage cao không đảm bảo chất lượng cao**. Em có thể có 80% coverage mà toàn test chạm vào code, không có assertion gì có ý nghĩa — vẫn pass dù app bị bug.

Em đo chất lượng test bằng vài chỉ số khác. **Bug escape rate** — số bug lọt ra production mà test không bắt được. **Mean time to detect** — bao lâu kể từ lúc bug được commit đến lúc test bắt. **Flakiness rate** — tỉ lệ test fail do nguyên nhân không liên quan đến code.

Coverage là điều kiện cần. Assertion ý nghĩa và bug escape rate thấp mới là điều kiện đủ.

> 💡 **Mẹo nói:** Không né câu hỏi. Nói rõ coverage là gì rồi reject premise "cao là tốt". Câu chốt: *"Coverage cao mà bug vẫn lọt là coverage giả."*

Câu 29. Em test các component khó như drag-and-drop, hover menu, file dialog hệ thống ra sao?

Mỗi loại có một cách riêng.

Drag-and-drop: Selenium có Actions class — chuỗi các hành động phức tạp gồm click chuột phải, hover, drag. Em dùng `dragAndDrop` cho thao tác đơn giản. Bẫy: nhiều thư viện drag-and-drop hiện đại dùng

HTML5 API, không phải mouse event truyền thống — Actions class không trigger đúng. Lúc đó em dùng JavaScript executor để dispatch sự kiện trực tiếp, hoặc dùng helper script từ cộng đồng.

Hover menu: dùng Actions class với `moveToElement`. Bẫy: hover state đôi khi đòi hỏi thời gian để menu hiện ra — phải chờ menu visible sau khi hover.

File dialog hệ thống: như đã nói ở câu upload — nếu là input file thì sendKeys đường dẫn vào, đừng click vào nút trigger dialog. Nếu app dùng custom uploader bypass input thường — dùng JavaScript hoặc reverse engineering xem nó dùng API nào.

Captcha: thẳng thắn nói luôn — Selenium không phải để bypass captcha. Đây là thiết kế: captcha tồn tại để chặn bot. Cách đúng là yêu cầu dev cung cấp **test environment có captcha tắt**, hoặc cung cấp **bypass token cho QA**. Em không khuyến khích các trick "giải captcha tự động" vì vi phạm điều khoản và không bền.

Notification của trình duyệt: pre-grant permission khi khởi tạo browser options.

> 💡 **Mẹo nói:** Câu này hỏi đa thành phần — đừng kể lan man. Chia theo từng loại component và giải pháp gọn. Câu chốt: *"Khó không phải vì Selenium yếu, mà vì component đó đứng ngoài DOM."*

Câu 30. Em định hướng phát triển bản thân trong test automation tiếp theo là gì?

Đây là câu mở để nói về em — đừng đọc CV.

Em chia ra ba mảng đang đầu tư.

Một là kỹ thuật. Em đang đào sâu **API testing** — vì nhiều test UI có thể dịch chuyển xuống tầng API, nhanh hơn và ổn định hơn. Em cũng theo dõi **Playwright** — không phải để bỏ Selenium, mà để có cái nhìn so sánh và mượn ý tưởng. Trong tương lai gần em muốn học sâu **performance testing** với JMeter hoặc k6 — vì khi automation chín, câu hỏi tiếp theo là "app có nhanh không" chứ không chỉ "có đúng không".

Hai là chiến lược test. Em đang đọc về **risk-based testing** — không phải test mọi thứ, mà test đúng chỗ rủi ro nhất. Đây là tư duy của người làm test architect, không chỉ người viết code test.

Ba là kỹ năng mềm. Em đang luyện **report cho stakeholder không kỹ thuật** — biến số liệu thành câu chuyện. Em cũng đang tập **mentor junior** — vì giúp người khác hiểu chính là cách tốt nhất để củng cố hiểu biết của mình.

Mục tiêu năm tới là chuyển từ vai trò "người viết test" sang "người định hình chiến lược test cho cả team".

> 💡 **Mẹo nói:** Trả lời cụ thể, có hướng, có lộ trình. Đừng nói "em muốn học thêm" chung chung. Câu chốt: *"Em không muốn dừng ở viết test, em muốn quyết định team test gì và không test gì."*

Lời khuyên chung khi phỏng vấn nói

Trước buổi phỏng vấn:

- Đọc to bộ 30 câu này 2-3 lần. Không phải để thuộc lòng, mà để quen nhịp nói. Khi quen nhịp, em sẽ tự ứng biến được khi câu hỏi đổi cách hỏi.
- Chuẩn bị sẵn **3 case study thật** từ kinh nghiệm — một về flaky test, một về tối ưu performance, một về xử lý feature khó. Mọi câu hỏi tình huống đều có thể quy về một trong ba case này.
- Chuẩn bị **câu hỏi ngược cho interviewer**. Một câu hỏi tốt cho thấy em quan tâm thật sự: "Suite hiện tại của team chạy bao lâu? Tỉ lệ flaky là bao nhiêu?" — 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 trả lời:

- Mở đầu bằng **ý chính** trong một câu, sau đó mới triển khai. Đừng dẫn dắt vòng vo.
- Dùng **ví dụ thật** thay vì lý thuyết. "Em từng gặp..." luôn ấn tượng hơn "theo nguyên tắc..."
- Có **con số cụ thể** ở chỗ nào được. "Giảm từ 25 phút xuống 9 phút" mạnh hơn "giảm đáng kể".
- Khi không biết, **thừa nhận thẳng**. "Em chưa làm cụ thể cái này nhưng em sẽ tiếp cận theo cách..." — câu này cho thấy em biết cách học, quan trọng hơn biết tất cả.

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.

Câu chốt cho cả buổi:

> *"Em làm automation không phải để khoe kỹ thuật. Em làm để team release nhanh hơn và yên tâm hơn."*

Câu này nói cuối buổi, khi interviewer hỏi "em còn gì muốn chia sẻ không". In dấu mạnh.