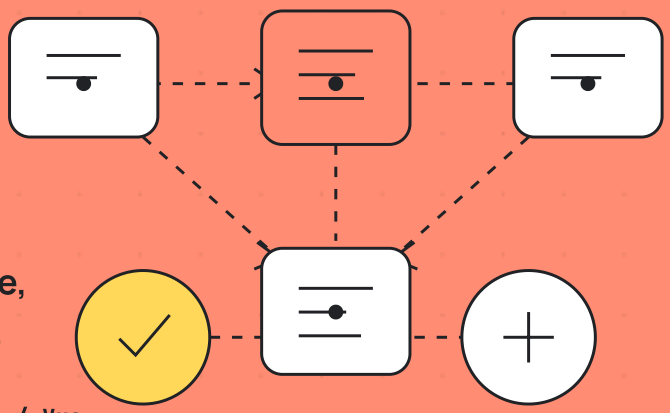
H/ HelpMeSwitch

FREE SAMPLE

INTERVIEW PLAYBOOK / FRONTEND

Frontend Interview Playbook



Prepare for questions about browser behaviour, rendering, performance, state, architecture and production debugging.

for frontend engineers · 2–8 yrs · React / Angular / Vue

THE PREMISE

You rarely lose the round on the first answer. You lose it when the interviewer asks *why*, *what failed*, and *what did you personally do?*

108	400+	12	108	108	12
QUESTIONS	FOLLOW-UPS	MODULES	FRAMEWORKS	REJECTION SIGNALS	DRILLS

FREE SAMPLE · 8 OF 108 QUESTIONS

What's inside this sample

One flagship question from eight different rounds — the complete weak-answer teardown, follow-ups and answer framework for each, exactly as they appear in the full playbook.

Q1.4 Describe a production bug in your frontend that was hard to reproduce. How did you find the root cause?

WHAT IT REALLY TESTS Real debugging under uncertainty - not "I added a log" but a hypothesis-driven hunt with browser tooling.

! WEAK ANSWER

> "There was a bug that only some users hit. I added some console logs and eventually found it and fixed it. It was an edge case with the state not updating correctly."

WHY IT FAILS

- × "Some users" - no attempt to characterise who or what differed.
- × "Added logs and eventually found it" skips the actual reasoning.
- × "State not updating correctly" is a symptom, not a root cause.
- × No tool, no hypothesis, no reproduction strategy.

↳ FOLLOW-UP LADDER

1. TECHNICAL
   What was different about the users who hit it - device, network, browser, timing?
   Narrowing the population to find the variable.
2. FAILURE
   What was your first hypothesis, and how did you disprove it?
   Structured elimination vs flailing.
3. TECHNICAL
   Which tool actually cracked it - Profiler, network tab, source maps, session replay?
   Fluency with real browser tooling.
4. MEASUREMENT
   How did you confirm the fix held for the affected users?
   Verification, not hope.

→ ANSWER FRAMEWORK

1 Symptom and who reported it → 2 What made it intermittent → 3 Population narrowing → 4 Hypothesis →
5 Tool that tested it → 6 Root cause → 7 Fix → 8 Confirmation.

× REJECTION SIGNALS

- × "Console.log everywhere" as the whole method.
- × Never characterised the affected population.
- × Root cause stated as a symptom.
- × No verification that the fix actually resolved it in the wild.

✓ MODEL ANSWER SKELETON

- → We had a bug where a saved form intermittently submitted stale data, but only for a minority of users and never on my machine. I noticed from logs that the affected sessions had two rapid submits milliseconds apart, which pointed at a race, not a data bug.
- → My first hypothesis was a double-click, but the timestamps were too tight and identical across users - it was React 18 automatic batching interacting with an async validation that resolved out of order, so a stale closure captured an old form value.
- → I confirmed it with a session replay tool that showed the exact keystroke-then-submit timing, then reproduced it locally by throttling the network to Slow 3G, which widened the race window.
- → The fix was an AbortController to cancel the in-flight validation on resubmit and reading form values from a ref at submit time rather than from a captured closure.
- → I verified by watching the stale-submit error rate, which went from ~0.4% of submissions to zero over the next release.

Q2.5

An API sends `"false"`, `"0"`, empty strings, and `null` for fields the UI treats as booleans and numbers. A disabled feature becomes enabled for some users. How do you diagnose and harden the boundary?

WHAT IT REALLY TESTS Whether the candidate understands coercion where it matters: untrusted data crossing into business logic with ambiguous missing-value semantics.

! WEAK ANSWER

> "JavaScript coercion is weird, so I would wrap the flag in `Boolean(value)`, use `Number(value)` for quantities, and switch every comparison to `==`."

WHY IT FAILS

- `Boolean("false")` is true, so the proposed fix preserves the incident.
- `Number("")` and `Number(null)` both become zero and erase meaning.
- Strict equality does not repair wrongly typed input.
- No contract, telemetry, or rejection path is defined.

4 FOLLOW-UP LADDER

1. TECHNICAL
   Why are `Boolean('false')` and `Boolean('0')` both true?
   Truthiness of non-empty strings.
2. TRADEOFF
   How do you distinguish absent, null, empty, zero, and invalid?
   Domain semantics before conversion.
3. OWNERSHIP
   Where should validation and normalisation live in a large frontend?
   One typed boundary rather than scattered component fixes.
4. MEASUREMENT
   What would you log without leaking sensitive payload data?
   Schema-failure metrics, field paths, and safe samples.

→ ANSWER FRAMEWORK

1. Capture the raw failing payload
2. Define each field's domain semantics
3. Parse explicitly with an allowlisted schema
4. Reject or default invalid values deliberately
5. Expose typed data to the app
6. Measure contract violations
7. Add boundary fixtures.

× REJECTION SIGNALS

- Uses generic `Boolean` or `Number` conversion without examples.
- Treats missing, null, empty, and zero as interchangeable.
- Fixes truthiness checks component by component.
- Silently defaults malformed server data with no visibility.

✓ MODEL ANSWER SKELETON

- I would first preserve the raw response and identify which value crossed the contract.
- A non-empty string is truthy, so `Boolean('false')` is true; similarly `Number('')` and `Number(null)` both produce zero, which can collapse 'missing' into a real quantity.
- The fix belongs at the API boundary: define a schema that accepts only the actual wire format, parse known boolean strings explicitly, and decide per field whether null means absent, unknown, or invalid.
- Components should receive a typed object and never reinterpret raw transport values.
- On a violation I would fail safely for permission-like flags, emit a redacted schema-error metric with the field path, and add fixtures for false, zero, empty, null, missing, and malformed values.

Q3.5

You have a search box firing fetches as the user types. Responses can arrive out of order and show stale results. How do you fix it?

WHAT IT REALLY TESTS Recognising async race conditions in the UI and knowing AbortController plus request-versioning as the real fixes.

! WEAK ANSWER

> "I'd add a debounce so it doesn't fire on every keystroke. That reduces the number of requests and mostly fixes the flicker of wrong results."

WHY IT FAILS

- Debounce reduces requests but doesn't fix out-of-order arrival.
- Ignores that a slow early request can still land after a fast late one.
- No cancellation of stale requests.
- No way to discard responses that no longer match the query.

4 FOLLOW-UP LADDER

1. TECHNICAL
   Debounce alone - does it actually prevent the stale-result race?
   No; a slow earlier request can still resolve last.
2. TECHNICAL
   How does AbortController help here?
   Cancel the previous fetch before starting the next.
3. TECHNICAL
   If you can't cancel, how do you discard stale responses?
   Track a request id/latest query and ignore mismatches.
4. TRADEOFF
   How do data libraries like React Query handle this for you?
   Keyed queries + cancellation of superseded fetches.

→ ANSWER FRAMEWORK

1. Diagnose it as a race, not a volume problem
2. Debounce to cut noise
3. AbortController to cancel the superseded request
4. Or version requests and drop responses that don't match the latest
5. Handle AbortError
6. Note library-level solutions.

× REJECTION SIGNALS

- Thinks debounce alone solves it.
- No cancellation or versioning.
- Doesn't handle the AbortError that cancellation throws.
- Unaware React Query/SWR solve this by keying.

✓ MODEL ANSWER SKELETON

- The root problem is a race, not request volume: keystroke A's request can be slower than keystroke B's, so an older query's response lands last and overwrites the newer one.
- Debounce helps by cutting the number of in-flight requests, but it doesn't fix the race - a slow earlier fetch can still resolve after a faster later one.
- The real fix is cancellation: I keep an `AbortController` per request, and before starting a new fetch I call `abort()` on the previous one and pass the new controller's `signal` to `fetch`, so the superseded request is cancelled and its promise rejects with an `AbortError` that I catch and ignore.
- As a belt-and-braces fallback where cancellation isn't available, I version requests - store the latest query string or an incrementing id and, when a response arrives, only apply it if it still matches the current query, otherwise discard it.
- In practice I'd lean on React Query or SWR, which key queries by input and automatically cancel or ignore superseded fetches, so I get debounce, cancellation, and caching without hand-rolling it.

Q4.8 A scroll handler makes the page sluggish. Walk me through diagnosing and fixing it.

WHAT IT REALLY TESTS Whether the candidate understands main-thread contention, passive listeners, throttling, and offloading work - a practical browser-performance scenario.

! WEAK ANSWER

> "I'd add a throttle to the scroll handler so it doesn't run so often. That usually fixes scroll performance issues because the function runs fewer times."

WHY IT FAILS

- Throttle helps frequency but ignores what the handler *does*.
- No mention of passive listeners and scroll-blocking.
- Doesn't consider forced reflows inside the handler.
- No profiling step to find the actual cost.

4 FOLLOW-UP LADDER

1. TECHNICAL
   What does a non-passive scroll listener block, and how do you fix it?
   It can block scrolling; {passive:true} lets the browser scroll immediately.
2. MEASUREMENT
   How would you check whether the handler forces layout?
   Performance panel - purple 'forced reflow' warnings.
3. TRADEOFF
   When is IntersectionObserver a better tool than a scroll handler?
   Visibility detection without main-thread scroll work.
4. TECHNICAL
   Where would you offload heavy computation?
   Web Worker / requestIdleCallback.

→ ANSWER FRAMEWORK

1. Profile first to see what the handler costs
2. Make the listener passive so scrolling isn't blocked
3. Throttle/rAF to align with frames
4. Remove forced reflows (batch reads/writes)
5. Replace visibility-detection scroll logic with IntersectionObserver
6. Offload heavy work to a worker or idle time.

× REJECTION SIGNALS

- Throttle as the only tool.
- Never heard of passive listeners.
- Doesn't profile before optimising.
- Unaware of IntersectionObserver.

✓ MODEL ANSWER SKELETON

- I'd profile first in the Performance panel while scrolling, because the fix depends on whether the cost is the handler firing too often, the handler forcing layout, or the listener blocking the scroll itself.
- A common hidden cost is a non-passive scroll listener: because a handler *could* call `preventDefault`, the browser may wait for it before scrolling, so I mark it `{passive: true}` to tell the browser I won't prevent default and let it scroll immediately on the compositor.
- Then I check the flamegraph for purple 'forced reflow' markers - if the handler reads `offsetTop` after writing styles, it thrashes layout every frame, so I batch reads then writes and wrap DOM writes in `requestAnimationFrame` to run at most once per frame.
- If the handler exists just to detect when elements enter the viewport - lazy-loading, infinite scroll, sticky headers - I replace it entirely with `IntersectionObserver`, which does visibility detection off the main thread with no scroll handler at all.
- And if there's genuinely heavy computation, I move it to a Web Worker or defer it with `requestIdleCallback` so it doesn't compete with scrolling.

Q5.9

PRACTICAL DRILL

Here's a component tree where typing in a search box re-renders a 200-item list on every keystroke and it lags. Diagnose and fix it, step by step.

WHAT IT REALLY TESTS Whether the candidate can trace a real re-render cascade and apply the right combination of state colocation, memo, and stable references.

! WEAK ANSWER

> N/A - drill. A weak attempt says "add useMemo everywhere" or "use React.memo" without identifying why the list re-renders or checking that props are stable.

WHY IT FAILS

- Blanket-memoises without diagnosing the cause.
- Doesn't notice the search state lives too high, re-rendering the list.
- Ignores inline handlers/objects defeating memo.
- No profiling to confirm.

↳ FOLLOW-UP LADDER

1. TECHNICAL
   Why is the list re-rendering when only the search input changed?
   Search state in a common parent re-renders all children.
2. TECHNICAL
   How does colocating the input state help?
   Move state down so only the input subtree re-renders.
3. FAILURE
   After memoising the list, what could still defeat React.memo?
   New array/handler identity passed as props each render.
4. MEASUREMENT
   How do you confirm the fix with the Profiler?
   Commit count/duration before and after.

→ ANSWER FRAMEWORK

1. Profile to find what re-renders → 2. Identify search state sitting in a shared parent →
3. Colocate input state or split components so the list isn't in the re-render path → 4. Wrap the list/items in React.memo →
5. Stabilise array and handler props (useMemo/useCallback) → 6. Confirm with Profiler.

× REJECTION SIGNALS

- Adds memo everywhere without diagnosis.
- Doesn't see the state-placement issue.
- Overlooks props that break memo identity.
- No before/after measurement.

✓ MODEL ANSWER SKELETON

- → Drill output: I'd start in the React Profiler, record a keystroke, and confirm the list is committing on every keypress. The root cause is almost always state placement - the search input's state lives in a parent that also renders the list, so every keystroke re-renders the parent and therefore the whole list.
- → The cleanest fix is to colocate: move the search input and its state into a small dedicated component so typing only re-renders that subtree, and let the list read the committed query via context or a lifted callback that only fires on submit or debounce, not per keystroke.
- → If the list genuinely must live under the same parent, I wrap the list and each row in `React.memo` so they skip re-rendering when their props are referentially equal. But memo only works if the props keep identity, so I'd check for the classic defeaters - a filtered array computed inline each render, or an inline `onSelect={() => ...}` handler - and stabilise them with `useMemo` for the array and `useCallback` for the handlers.
- → If the filtering itself is heavy I'd debounce the query so it runs on pause, not per keystroke.
- → Then I re-record in the Profiler and confirm the list's commit count drops from every-keystroke to only when the query actually changes.

Q6.7 How do you implement an optimistic update, and how do you handle it failing?

WHAT IT REALLY TESTS Whether the candidate can update the UI ahead of the server and correctly roll back, not just fire-and-hope.

! WEAK ANSWER

> "An optimistic update is when you update the UI before the server responds so it feels fast. If it fails you show an error message telling the user something went wrong."

WHY IT FAILS

- No rollback - just an error message leaves the UI in a wrong state.
- Doesn't capture the previous state to restore.
- No mention of reconciling with the server's real response.
- Ignores concurrent/competing updates.

4 FOLLOW-UP LADDER

1. FAILURE
   When the mutation fails, how do you get the UI back to correct?
   Snapshot previous state and roll back to it.
2. TECHNICAL
   How do you reconcile with the server's actual response on success?
   Replace optimistic value with server truth / invalidate & refetch.
3. TECHNICAL
   What can go wrong with concurrent optimistic updates?
   Ordering/overwrite; need per-mutation snapshots or ids.
4. TRADEOFF
   When is an optimistic update the wrong choice?
   Money/irreversible actions or high-failure operations.

→ ANSWER FRAMEWORK

1. Snapshot current state → 2. Apply the expected change to the UI immediately → 3. Fire the mutation →
4. On success reconcile with server truth (or invalidate/refetch) → 5. On failure roll back to the snapshot and surface the error →
6. Guard concurrency → 7. Avoid for irreversible/high-risk actions.

× REJECTION SIGNALS

- No rollback mechanism.
- Doesn't snapshot previous state.
- Never reconciles with server response.
- Applies optimism to risky/irreversible actions.

✓ MODEL ANSWER SKELETON

→ An optimistic update applies the change to the UI immediately, before the server confirms, so the interaction feels instant - but the part that actually matters is the failure path. Before I apply the change I snapshot the current state; I apply the expected new value to the cache or store; then I fire the mutation.

→ On success I reconcile with the server's real response - either replace my optimistic value with what the server returned, or invalidate the query so it refetches the authoritative data, because my guess might differ from the server's canonical result.

→ On failure I roll back to the snapshot I took and surface the error, so the UI returns to a correct state rather than sitting on a lie with just a toast.

→ With React Query I'd do exactly this in `onMutate` (cancel in-flight queries, snapshot, optimistically set) and restore in `onError`, then `invalidateQueries` in `onSettled`.

→ I'm careful with concurrency - overlapping optimistic updates need per-mutation snapshots so one rollback doesn't clobber another's change - and I don't use optimism for irreversible or money-moving actions like a payment, where a wrong-then-corrected UI is worse than a brief spinner.

Q9.3 How do you handle a flaky API on the client - timeouts, retries, and failures?

WHAT IT REALLY TESTS Whether the candidate can build resilient fetching with backoff, jitter, idempotency awareness, and timeouts, not just a try/catch.

! WEAK ANSWER

> "I'd add retries so if a request fails it tries again. Maybe retry a few times. I'd also show an error message if it keeps failing. A timeout stops it hanging forever."

WHY IT FAILS

- Retries without backoff can hammer a struggling server.
- No jitter - synchronised retries cause thundering herd.
- Doesn't consider idempotency (retrying a POST).
- `fetch` has no native timeout - omission unaddressed.

4 FOLLOW-UP LADDER

1. TECHNICAL
   Why retry with exponential backoff and jitter instead of immediately?
   Avoid hammering; jitter prevents synchronised retry storms.
2. FAILURE
   Which requests are safe to retry automatically?
   Idempotent GET/PUT/DELETE; not a bare POST without an idempotency key.
3. TECHNICAL
   How do you actually time out a fetch?
   AbortController + setTimeout; fetch has no timeout option.
4. TRADEOFF
   Which status codes should you retry versus not?
   Retry 429/503/network; not 400/401/403/404.

→ ANSWER FRAMEWORK

1. Timeout via AbortController (fetch has none natively) → 2. Retry only idempotent requests, or POSTs with an idempotency key →
3. Exponential backoff + jitter to avoid herds → 4. Retry transient/retriable statuses (429/503/network), respect Retry-After →
5. Don't retry client errors (4xx) → 6. Cap attempts, then surface a clear error and recovery.

× REJECTION SIGNALS

- Retries everything, including POSTs, blindly.
- No backoff or jitter.
- Thinks fetch has a timeout option.
- Retries 400/401 pointlessly.

✓ MODEL ANSWER SKELETON

- First, `fetch` has no native timeout, so I implement one with an `AbortController` and a `setTimeout` that aborts the request, otherwise a hung connection spins forever. For retries I'm careful on two axes.
- Timing: I retry with exponential backoff - 200ms, 400ms, 800ms - plus jitter, because retrying immediately hammers an already-struggling server, and jitter stops many clients retrying in lockstep and creating a thundering herd.
- Safety: I only auto-retry *idempotent* requests - GET, PUT, DELETE - because retrying a bare POST can double-charge or duplicate; if a POST must be retriable I have the server support an idempotency key so replays are deduplicated. I retry only transient failures - network errors, 429, 502/503/504 - and honour a `Retry-After` header when present, but I never retry a 400, 401, 403, or 404 because those won't succeed on repeat and just waste time.
- I cap the attempts, and when it still fails I surface a clear, actionable error with a manual retry option rather than an infinite spinner.
- In practice React Query gives me configurable retries, backoff, and timeouts declaratively, so I'd lean on it rather than hand-rolling this per call.

Q11.9

PRACTICAL DRILL

You're paged: "checkout is showing a blank white screen for ~20% of users, started 10 minutes ago." Talk me through your first 15 minutes.

WHAT IT REALLY TESTS Whether the candidate mitigates first, correlates with a release, and works evidence-first under incident pressure - not debugging in prod live.

! WEAK ANSWER

> "N/A - drill. A weak attempt starts reading through the checkout code to find the bug, tries to write a fix live, and never considers rolling back or checking what changed."

WHY IT FAILS

- × Debugs the code before mitigating a revenue-impacting incident.
- × Doesn't correlate the 10-minute onset with a deploy.
- × No rollback consideration.
- × Doesn't quantify/scope impact or communicate.

4 FOLLOW-UP LADDER

1. OWNERSHIP
   What's your very first action, and why not start reading code?
   Mitigate first - rollback if a deploy correlates; stop the bleeding.
2. TECHNICAL
   What does 'started 10 minutes ago' most likely implicate?
   A deploy, config/flag flip, or a third-party script/API change.
3. TECHNICAL
   Why 20% and not 100% - what does that suggest?
   A cohort: canary, region, browser, feature flag, one CDN edge.
4. MEASUREMENT
   How do you confirm recovery after mitigating?
   Error rate and checkout success for the affected cohort return to baseline.

→ ANSWER FRAMEWORK

1. Assess scope/impact and communicate (comms channel, status) →
2. Correlate onset with deploys/flags/third-parties in the last ~15 min →
3. Mitigate first: roll back or flip the flag, don't debug live →
4. Check error monitoring (source-mapped) for the exploding error + which cohort (the 20%) →
5. Confirm the affected cohort recovers →
6. Root-cause and add a regression guard afterwards.

× REJECTION SIGNALS

- × Reads code / patches live before mitigating.
- × Ignores the deploy correlation.
- × Doesn't scope the 20% cohort.
- × No recovery verification or comms.

✓ MODEL ANSWER SKELETON

- → Drill output: this is a revenue-critical incident, so my mindset is mitigate first, root-cause later - I don't start reading checkout code. Minute zero to two: acknowledge the page, open an incident channel, and confirm scope - is it really ~20% and is checkout success rate actually down - because I need to know the blast radius and keep stakeholders informed.
- → Two to five: correlate the 10-minute onset with change events - our deploy log, feature-flag flips, and third-party script/API status - because a blank white screen appearing suddenly is almost always a JS bundle error from a release or a config change, not spontaneous.
- → The '20%, not 100%' detail is a strong signal it's a *cohort* - a canary/gradual rollout, one region, a specific browser hitting an unsupported API, a feature flag, or one bad CDN edge - so I check error monitoring (with source maps) for the spiking error and segment by release and browser to see who's failing; a white screen usually means an uncaught error before render, which the stack trace will pinpoint. Five to ten: mitigate - if a deploy correlates, roll it back or disable the feature flag, which stops the bleeding for everyone without me having to understand the bug yet.
- → Ten to fifteen: confirm recovery by watching the affected cohort's error rate and checkout success return to baseline, not the global average.
- → Only then do I do the calm root-cause and add a test or error boundary so a render-time error degrades gracefully instead of blanking the page next time.

YOU JUST READ 8. THERE ARE 108.

The full round is waiting.

This sample gave you one question from each of eight rounds. The full playbook carries every question, every follow-up, and the answer framework for all 108 — the ones most likely to appear in your next interview.

01

See the full contents

Every module, every question and the exact follow-ups an interviewer asks after your first answer.

02

Practise the second question

Anyone can rehearse the opener. Strong candidates separate on what failed, what they owned, and what they would change now.

03

Walk in ready

Downloadable PDF, yours to keep. Read it the night before and stop being surprised by the follow-up.

> **A polished answer can sound borrowed. A specific answer sounds true.**

GET THE FULL PLAYBOOK helpmeswitch.in/playbooks/frontend-interview-playbook