

Mobile Developer Interview Questions and Answers

Mobile Developer | Intern to senior level

Mobile developer interview questions and answers for React Native developers: React components, hooks, state, re-renders, TypeScript, Git and behavioral questions.

59 questions Read online: <https://www.freedevtemplates.com/interview-questions/mobile>

TypeScript

1. What is TypeScript and why do we use it?

TypeScript Easy Intern, Junior

TypeScript is a typed superset of JavaScript that compiles to plain JavaScript. It catches type errors in the editor and at build time, and it documents the shape of data for the next person who reads the code.

Every valid JavaScript file is already TypeScript, which is why teams can adopt it gradually. The compiler (tsc) erases types and emits JS that runs anywhere JavaScript runs. Types exist only at compile time: they are not available for runtime checks unless you add a library or generate them.

The usual wins are safer refactors, better editor autocomplete, and APIs that are harder to call wrongly. The cost is a build step and time spent modelling awkward JavaScript patterns. Interviews often follow this with "when would you still use JavaScript?" - small scripts, prototyping, or a codebase that is not yet worth converting.

2. What is the difference between interface and type in TypeScript?

TypeScript Easy Intern, Junior

Both describe object shapes. interface can be merged and is preferred for public object contracts; type can express unions, intersections and primitives, and cannot be reopened later.

For a simple object, they look the same and teams pick one style. interface supports declaration merging: two interface User blocks become one, which is how many .d.ts files extend library types. type aliases are fixed once written.

Use type when you need a union (string | number), a mapped type, or a tuple. Use interface when you are describing an object that others might extend. You can implement either with a class. Do not waste interview time claiming one is "faster"; they compile to the same nothing.

Key points

- interface: extendable, mergeable, object-shaped
- type: unions, intersections, primitives, tuples
- Pick one style for objects and stay consistent

```
interface User { id: string; name: string }
interface User { email?: string } // merged

type Id = string | number; // union: type only
type Admin = User & { role: "admin" }; // intersection
```

3. What is the difference between any, unknown and never in TypeScript?

TypeScript Medium Junior, Mid-level

any turns off checking. **unknown** is a type-safe any: you must narrow it before use. **never** means a value cannot exist, used for exhaustive checks and functions that always throw.

A value typed `any` can be passed anywhere and you can read any property on it, so a type compiles. `unknown` is what you give to untrusted input (`JSON.parse`, a network payload) until you prove its shape with `typeof`, `in`, or a type guard.

`never` is the empty set. A function with return type `never` does not return (throw or infinite loop). In a `switch`, assigning the leftover value to `const _exhaustive: never = value` makes the compiler fail when a new union member is added and not handled.

```
function handle(x: unknown) {
  if (typeof x === "string") return x.toUpperCase();
  throw new Error("expected string");
}

function fail(message: string): never {
  throw new Error(message);
}
```

4. What are generics in TypeScript?

TypeScript Medium Junior, Mid-level

Generics are type parameters: they let a function, class or type work with many types while keeping the relationship between inputs and outputs, instead of falling back to `any`.

`identity<T>(value: T): T` returns the same type it received. Without `T` you would write `any` and lose that link.

Constraints (`T extends { id: string }`) describe the minimum shape. Several parameters (`K`, `V`) model maps and API wrappers.

Common interview follow-ups: default parameters (`T = string`), `keyof T` for safe property access, and why `Array<T>` and `T[]` are the same. Prefer inference (`wrap(user)`) over forcing `wrap<User>(user)` at every call site.

```
function first<T>(items: T[]): T | undefined {
  return items[0];
}

first(["a", "b"]); // string | undefined
first<number>([1, 2, 3]); // number | undefined
```

5. What are union and intersection types in TypeScript?

TypeScript Medium Junior, Mid-level

A union (`A | B`) is a value that is one of the members. An intersection (`A & B`) is a value that must satisfy both types at once.

Unions model "this or that": a result that is `{ ok: true, data: T } | { ok: false, error: string }`. Narrow with a discriminant field (`ok`) before you read `data`. Intersections model "this and that": `User & { role: "admin" }` has every `User` field plus `role`.

A union of objects is not an object with optional fields. `A | B` only lets you access properties common to both until you narrow. That is the usual bug when someone writes `user.role` on a `User | Admin` without checking.

6. What are utility types in TypeScript like `Partial`, `Pick` and `Omit`?

TypeScript Medium Junior, Mid-level

Utility types are built-in helpers that transform an existing type: `Partial` makes every field optional,

Required the opposite, Pick keeps a set of keys, Omit removes them, and Record builds a key/value map.

`Partial<User>` is useful for a PATCH body. `Pick<User, "id" | "name">` is a list-row type. `Omit<User, "password">` is what you send to the client. `Record<Id, User>` is a dictionary. `ReturnType<typeof fn>` and `Parameters<typeof fn>` pull types from functions so you do not repeat them.

You can write the same things with mapped types; the utilities are the readable version. Interviews sometimes ask you to implement `Partial` yourself: `{ [K in keyof T]?: T[K] }`.

```
type User = { id: string; name: string; password: string };
type PublicUser = Omit<User, "password">;
type UserPatch = Partial<Pick<User, "name" | "password">>;
```

7. What is type narrowing in TypeScript?

TypeScript Medium Mid-level, Senior

Narrowing is when TypeScript shrinks a wide type (such as a union) to a more specific type after a runtime check like `typeof`, `instanceof`, `in`, or a custom type guard.

Control-flow analysis follows your `if` and `switch`. `typeof x === "string"` narrows `string | number` to `string` in that branch. `x instanceof Date` works for classes. "width" in value distinguishes object shapes. A user-defined guard is a function whose return type is value is `Fish`.

Equality checks against a discriminant (`event.type === "click"`) are the cleanest pattern for API payloads.

Assertions (`value as User`) skip narrowing and are a common source of production bugs: the compiler believes you, the runtime does not.

```
function isUser(value: unknown): value is { id: string } {
  return typeof value === "object" && value !== null && "id" in value;
}

function label(value: string | number) {
  return typeof value === "string" ? value.toUpperCase() : value.toFixed(2);
}
```

8. What is the difference between enum and union types in TypeScript?

TypeScript Medium Mid-level, Senior

A string union such as `"admin" | "user"` exists only in the type system and erases to plain strings. A TypeScript enum compiles to a JavaScript object (unless it is a `const enum`), which is extra runtime code and a weaker default.

Most codebases prefer union types plus `as const` objects when they also need a runtime list (`const Roles = ["admin", "user"] as const`). Numeric enums are especially surprising because they reverse-map (`Color[0] === "Red"`). `const enum` inlines values but is banned in some projects because it breaks isolated modules and bundlers.

If a library already exposes an enum, use it. When you design a new API, a union is simpler to tree-shake, serialise and explain.

9. How does TypeScript type inference work?

TypeScript Medium Mid-level, Senior

TypeScript fills in types you omit: literals get their literal or widened type, function return types are inferred from return statements, and generic arguments are inferred from the values you pass.

`let count = 0` becomes `number`. `const role = "admin"` stays the literal `"admin"` because `const` cannot be

reassigned. Contextual typing works the other way: a callback argument is inferred from the function you pass it to (`arr.map(item => ...)`).

Write types at module boundaries (function arguments, exported APIs, public class fields) and let inference handle locals. Over-annotating locals makes refactors noisy. When inference widens too far (`[]` is `any[]` without a hint), give the generic or a typed empty value (`[]` as `User[]` or `User[] = []`).

10. What are mapped types and conditional types in TypeScript?

TypeScript Hard Senior

Mapped types build a new object type by looping over keys (`{ [K in keyof T]: ... }`). Conditional types pick result with `T extends U ? X : Y`, which is how many utility types are implemented.

`Partial<T>` is a mapped type that adds `?` to every key. You can also remap keys with `as` (`[K in keyof T as ${K & string}Id]`). Conditional types power `Exclude`, `Extract` and `NonNullable`. They become interesting with `infer`, which pulls a piece out of a type (`T extends Promise<infer U> ? U : T`).

Senior interviews look for judgement: these features belong in shared utility types, not in every component file. If a conditional type is hard to read, a few well-named helpers beat one clever one-liner. Recursion has a depth limit; it is the wrong tool for modelling arbitrarily nested JSON unless you must.

```
type ReadonlyFields<T> = { readonly [K in keyof T]: T[K] };
type Awaited<T> = T extends Promise<infer U> ? Awaited<U> : T;
```

React

11. What is the virtual DOM in React and how does it work?

React Easy Intern, Junior

The virtual DOM is a lightweight JavaScript copy of the UI that React compares with the previous version after each update, so it only changes the parts of the real DOM that actually differ.

When state or props change, React re-runs your component to produce a new tree of React elements (the virtual DOM). It then compares that tree with the previous one in a process called reconciliation, works out the minimal set of changes, and applies only those to the real DOM.

Touching the real DOM is comparatively expensive, and batching updates lets React do it once per update instead of once per change. The virtual DOM is not automatically faster than hand-written DOM code, but it gives you a declarative model: you describe what the UI should look like for the current state, and React figures out how to get there.

Key points

- Virtual DOM = tree of plain objects describing the UI
- Reconciliation compares old and new trees (diffing)
- Only the differences are applied to the real DOM

12. What is the difference between props and state in React?

React Easy Intern, Junior

Props are read-only inputs passed from a parent component, while state is data a component owns and can change over time, which triggers a re-render when updated.

Props make a component reusable: the parent decides the values, and the child must not modify them. State is private memory for a component, created with `useState` or `useReducer`. Updating state tells React to re-render that component with the new value.

A good rule is to keep state as low as possible in the tree, and lift it up to the closest common parent only when

two components need to share it. Data that can be calculated from props or state should be computed during render and not stored in extra state.

In the interview

Props are inputs passed from a parent and are read-only inside the child. State is data a component owns and changes over time with a setter, which triggers a re-render. If two components need the same data, I lift the state to their closest common parent and pass it down as props.

Key points

- Props flow down, are read-only and come from the parent
- State is owned by the component and changes cause re-renders
- Derived values should be calculated, not stored in state

```
function Counter({ step }) { // step is a prop (read-only)
  const [count, setCount] = useState(0); // count is state (owned here)
  return <button onClick={() => setCount(count + step)}>{count}</button>;
}

<Counter step={5} />
```

13. What are React hooks and what are the rules of hooks?

React Easy Junior, Mid-level

Hooks are functions such as `useState` and `useEffect` that let function components use state, lifecycle behaviour and other React features; they must be called at the top level of a component or custom hook, never inside loops, conditions or nested functions.

Before hooks, state and lifecycle methods required class components. Hooks let you do the same in function components and, importantly, let you extract and reuse stateful logic as custom hooks without wrapper components or render props.

React tracks hooks by the order in which they are called during each render. If a hook is skipped in one render because of an if statement, every hook after it would read the wrong state. That is why the two rules exist: only call hooks at the top level, and only call them from React function components or other custom hooks (whose names start with use). The `eslint-plugin-react-hooks` package enforces both.

Key points

- Call hooks at the top level only
- Call hooks only from components or custom hooks
- React relies on call order to match state to hooks

```
function useWindowWidth() { // custom hook
  const [width, setWidth] = useState(window.innerWidth);
  useEffect(() => {
    const onResize = () => setWidth(window.innerWidth);
    window.addEventListener("resize", onResize);
    return () => window.removeEventListener("resize", onResize);
  }, []);
  return width;
}
```

14. Why do we need keys in React lists?

React Easy Junior, Mid-level

Keys give each list item a stable identity so React can match items between renders and correctly add, remove or reorder them without recreating or mixing up their state.

When React diffs two lists it needs to know which new item corresponds to which old one. Without keys it falls back to matching by position, so inserting an item at the top makes every item look changed, and local state

such as an input's text can end up attached to the wrong row.

A key must be unique among siblings and stable across renders. Use an ID from your data. Using the array index is acceptable only for static lists that are never reordered, filtered or inserted into. Never use `Math.random()`, since a new key on every render makes React remount every item.

```
{todos.map(todo => (  
  <TodoItem key={todo.id} todo={todo} /> // stable ID, not the index  
))}
```

15. How does `useEffect` work in React and what is the dependency array?

React Medium Junior, Mid-level

`useEffect` runs a function after the component renders; the dependency array tells React when to re-run it: with no array it runs after every render, with an empty array only after the first, and with values only when those values change.

Effects are for synchronising your component with something outside React, such as subscriptions, timers, network requests or the document title. The function you return from an effect is the cleanup, and React runs it before the next run of the effect and when the component unmounts.

Every value from the component that the effect uses should be listed in the dependency array, otherwise the effect works with stale values. In development with Strict Mode, React deliberately mounts, unmounts and re-mounts components once to help you find missing cleanup. Do not use effects for things you can calculate during render or handle in an event handler.

In the interview

`useEffect` runs after React commits the render, so it is the place to synchronise with things outside React such as subscriptions, timers or network requests. The dependency array controls when it re-runs, and the function it returns is the cleanup that runs before the next effect and on unmount.

Key points

- No array: every render. `[]` : once. `[a, b]`: when a or b change
- Return a function to clean up
- Do not use effects for derived data or user-triggered actions

```
useEffect(() => {  
  const controller = new AbortController();  
  fetch("/api/users/" + userId, { signal: controller.signal })  
    .then(r => r.json())  
    .then(setUser)  
    .catch(err => { if (err.name !== "AbortError") setError(err); });  
  
  return () => controller.abort(); // cleanup when userId changes or unmounts  
}, [userId]);
```

16. What is the difference between controlled and uncontrolled components in React?

React Medium Junior, Mid-level

A controlled component keeps the input's value in React state and updates it through `onChange`, while an uncontrolled component lets the DOM hold the value and you read it with a `ref` when needed.

Controlled inputs make React the single source of truth: the value comes from state and every keystroke calls a handler that updates state. This makes validation, formatting and conditional disabling easy. The cost is a re-render on each keystroke.

Uncontrolled inputs use `defaultValue` and `ref`, so the browser manages the value and you read it on submit. They are simpler for basic forms and file inputs (which are always uncontrolled). Form libraries often use uncontrolled inputs under the hood to avoid re-rendering the whole form while typing.

```
// controlled
const [email, setEmail] = useState("");
<input value={email} onChange={e => setEmail(e.target.value)} />

// uncontrolled
const ref = useRef(null);
<input defaultValue="" ref={ref} />
const onSubmit = () => console.log(ref.current.value);
```

17. What is prop drilling and how do you avoid it in React?

React Medium Junior, Mid-level

Prop drilling is passing props through several layers of components that do not use them just to reach a deeply nested child; you avoid it with composition, the Context API or a state management library.

First try composition: pass the child component itself, or pass children, so intermediate components never need to know about the data. This is often enough and keeps components easy to test.

If many distant components need the same value, such as the current user or theme, use the Context API. Context re-renders every consumer when its value changes, so keep contexts small and focused, and split frequently changing values from stable ones. For complex client state, a store such as Redux Toolkit or Zustand lets components subscribe to just the slice they need.

```
const ThemeContext = createContext("light");

function App() {
  return (
    <ThemeContext.Provider value="dark">
      <Layout /> { /* no theme prop needed */}
    </ThemeContext.Provider>
  );
}

function Button() {
  const theme = useContext(ThemeContext);
  return <button className={theme}>Save</button>;
}
```

18. What is the difference between useMemo and useCallback in React?

React Medium Mid-level, Senior

useMemo caches the result of a calculation, while useCallback caches a function definition; both only recompute when their dependencies change.

useMemo(() => compute(a, b), [a, b]) returns the cached value. useCallback(fn, deps) is shorthand for useMemo(() => fn, deps): it returns the same function instance between renders.

Neither makes your app faster by default; each adds its own cost. Use useMemo for genuinely expensive calculations, and use useCallback mainly when the function is passed to a child wrapped in React.memo or is a dependency of another hook. A stable function reference is useless if the child is not memoized, since the child re-renders with its parent anyway.

Key points

- useMemo returns a value, useCallback returns a function
- Both depend on a dependency array
- Measure before optimising; memoisation has a cost

```
const visible = useMemo(
  () => todos.filter(t => t.status === filter),
  [todos, filter]
);

const handleSelect = useCallback((id) => setSelected(id), []);

const Row = React.memo(function Row({ item, onSelect }) { /* ... */ });
<Row item={item} onSelect={handleSelect} />
```

19. What is the difference between useRef and useState in React?

React Medium Mid-level, Senior

Updating state triggers a re-render, while changing a ref's .current value does not; use state for data shown on screen and refs for DOM access or values that must persist without causing renders.

useRef returns a mutable object whose .current property lives for the whole life of the component. Typical uses are holding a DOM node (to focus an input or measure an element), storing a timer ID, or remembering the previous value of a prop.

Because React does not know when you change a ref, the UI will not update. If a value affects what the user sees, it belongs in state. Avoid reading or writing ref.current during rendering, except for lazy initialisation.

```
function SearchBox() {
  const inputRef = useRef(null);
  const [query, setQuery] = useState(""); // changes the UI

  return (
    <>
      <input ref={inputRef} value={query} onChange={e => setQuery(e.target.value)} />
      <button onClick={() => inputRef.current.focus()}>Focus</button>
    </>
  );
}
```

20. Why does a React component re-render and how do you prevent unnecessary re-renders?

React Hard Senior

A component re-renders when its state changes, its parent re-renders, or a context it consumes changes; you prevent wasted renders by moving state down, using React.memo, and keeping props referentially stable.

By default, when a component renders, all of its children render too, even if their props did not change. React then compares the output and only updates the DOM where something is different, so a re-render is not necessarily a DOM update, but the work of calling your component still costs time.

Start by fixing the structure: keep state close to where it is used, and split a component so a frequently changing part (such as an input) does not drag a heavy sibling along. Passing children as props also helps, because the same element object does not re-render. If that is not enough, wrap pure components in React.memo, stabilise object and function props with useMemo and useCallback, and split large contexts. Use the React DevTools Profiler to confirm what actually rendered and why before optimising.

Key points

- Triggers: own state change, parent render, context change
- Move state down and compose with children first
- React.memo only helps if props are stable

21. How do you optimize performance in a React application?

React Hard Senior

Measure first with the Profiler and Lighthouse, then reduce unnecessary renders, split code with lazy loading, virtualise long lists, and optimise images and bundle size.

Rendering: avoid wasted re-renders by colocating state, using React.memo, useMemo and useCallback where profiling shows a benefit, and avoiding creating new objects in props for memoised children. For slow updates triggered by typing or filtering, useTransition and useDeferredValue keep the interface responsive.

Loading: split the bundle by route with React.lazy and Suspense (or Next.js dynamic imports), remove unused dependencies, and analyse the bundle with a tool such as webpack-bundle-analyzer. Serve properly sized, modern-format images and lazy-load below-the-fold content.

Data and lists: virtualise long lists with a library such as react-window so only visible rows are in the DOM, cache server data with React Query or SWR, and debounce expensive handlers. Track real-user metrics such as Largest Contentful Paint and Interaction to Next Paint, not just lab scores.

Key points

- Profile before optimising
- Reduce renders, reduce bundle size, reduce DOM size
- Use real-user metrics to verify improvements

```
const Reports = React.lazy(() => import("./Reports"));

<Suspense fallback={<Spinner />}>
  <Reports />
</Suspense>
```

Git

22. What is Git and how is it different from GitHub?

Git Easy Intern, Junior

Git is a distributed version control system that tracks changes to files on your machine; GitHub is a hosting service for Git repositories that adds collaboration features such as pull requests, code review and CI.

Git records snapshots of a project as commits. Every clone holds the full history, so you can commit, branch, view logs and compare versions without a network connection. Branches are cheap pointers to commits, which is what makes feature branches and experiments practical.

GitHub, GitLab and Bitbucket host a copy of the repository on a server so a team can share it. On top of Git they add pull requests, code review, issue tracking, permissions and automation such as GitHub Actions. You can use Git without GitHub, but not the other way round.

In day-to-day work you commit locally with Git, push your branch to the hosted remote, open a pull request for review, and merge it once checks pass.

In the interview

Git is the version control tool that runs locally and stores the full history in every clone. GitHub is a service that hosts Git repositories and adds pull requests, reviews, issues and CI on top. I use Git to commit and branch, and GitHub to share work and get it reviewed.

Key points

- Git: the version control tool, works offline
- GitHub: a hosting and collaboration platform for Git repositories
- Every Git clone contains the full history

```
git init # start tracking a project
git add src/app.js # stage a change
git commit -m "Add login form"
git remote add origin https://github.com/you/project.git
git push -u origin main # share it on GitHub
```

23. What is the difference between git fetch and git pull?

Git Easy Intern, Junior

git fetch downloads new commits from the remote without changing your working branch; git pull runs a fetch and then merges (or rebases) those commits into your current branch.

git fetch updates your remote-tracking branches such as origin/main. Your own branch and working files are untouched, so it is always safe. You can then inspect what changed with `git log main..origin/main` before deciding how to integrate it.

git pull is git fetch followed by git merge by default. If your branch and the remote have both moved, it creates a merge commit or stops with conflicts. Many teams set `git config pull.rebase true` so pull replays your local commits on top of the remote instead, which keeps history linear.

A good habit before starting work is to fetch, review, then merge or rebase deliberately, rather than pulling blindly into a branch with uncommitted changes.

Key points

- fetch: download only, never touches your branch
- pull: fetch + merge (or rebase with `--rebase`)
- Fetch first when you want to review incoming changes

```
git fetch origin
git log --oneline main..origin/main # what is new on the remote?
git merge origin/main # integrate when ready

# or in one step
git pull --rebase origin main
```

24. What is the difference between git merge and git rebase?

Git Medium Junior, Mid-level

Both integrate changes from one branch into another: merge keeps both histories and joins them with a merge commit, while rebase rewrites your commits so they appear on top of the target branch, giving a linear history.

Merging main into a feature branch creates a merge commit with two parents. Nothing is rewritten, so it is safe on shared branches, and the history shows exactly when branches were combined. The cost is a busier history with many merge commits.

Rebasing a feature branch onto main takes each of your commits and replays it on the latest main, creating new commits with new hashes. The result reads as if you started your work from the current main, which makes git log and git bisect easier to follow.

Because rebase rewrites history, the golden rule is: never rebase commits that other people have already based work on. Rebase your private feature branch to tidy it up, then merge it (often with a squash or fast-forward) into the shared branch.

In the interview

Merge combines two branches with a merge commit and never rewrites history, so it is safe on shared branches. Rebase replays my commits on top of the target branch, which gives a clean linear history but changes commit hashes. I rebase my own feature branch to keep it current and tidy, and I never rebase a branch someone else is

working on.

Key points

- Merge preserves history and is safe on shared branches
- Rebase rewrites commits to produce a linear history
- Never rebase commits that others already use
- Use --force-with-lease, not --force, after rebasing

```
# merge: keeps history, adds a merge commit
git switch feature
git merge main

# rebase: replays feature commits on top of main
git switch feature
git rebase main
git push --force-with-lease # needed because hashes changed
```

25. How do you resolve a merge conflict in Git?

Git Medium Junior, Mid-level

Open the conflicted files, decide what the final code should be between the conflict markers, remove the markers, run the tests, then stage the files and continue the merge or rebase.

A conflict happens when two branches change the same lines, or one deletes a file the other edited. Git pauses the merge and marks each conflicted region with <<<<<<, ===== and >>>>>>. git status lists the files that need attention.

Resolving is not choosing a side. Read both changes, understand why each was made, and write the code that keeps the intent of both. Editors and git mergetool show the two versions side by side, but the decision is yours.

After editing, build and run the tests, because a conflict-free file can still be logically broken. Then git add the resolved files and run git merge --continue or git rebase --continue. If it goes wrong, git merge --abort returns you to the state before the merge.

Key points

- git status shows conflicted files
- Combine the intent of both changes, then remove the markers
- Run tests before continuing
- git merge --abort or git rebase --abort to start over

After editing: git add config.js && git merge --continue

```
<<<<<< HEAD
const timeout = 5000;
=====
const timeout = Number(process.env.TIMEOUT_MS);
>>>>>> feature/config

# resolved: keep the env variable, fall back to the old default
const timeout = Number(process.env.TIMEOUT_MS ?? 5000);
```

26. What is the difference between git reset and git revert?

Git Medium Mid-level, Senior

git revert creates a new commit that undoes an earlier one and is safe on shared branches; git reset moves the branch pointer back to an earlier commit, effectively removing later commits from that branch's history.

git revert <commit> calculates the inverse of a commit and records it as a new commit. History only grows, so teammates who already pulled are not affected. This is how you undo a bad change that has reached main or

production.

git reset moves the current branch to another commit. With --soft the undone changes stay staged, with the default --mixed they stay in your working tree unstaged, and with --hard they are discarded. It is ideal for cleaning up local commits that have not been pushed.

If you reset by mistake, the commits are usually still recoverable for a while through git reflog, which records where your branch pointed before each move.

Key points

- revert: new commit, history preserved, safe when shared
- reset: moves the branch pointer, rewrites history
- --soft keeps changes staged, --mixed unstaged, --hard discards them
- git reflog can recover "lost" commits

```
# undo a pushed commit safely
git revert a1b2c3d

# rewrite local history (not yet pushed)
git reset --soft HEAD~1 # undo commit, keep changes staged
git reset --hard HEAD~1 # undo commit and discard changes

# recover from a bad reset
git reflog
git reset --hard HEAD@{2}
```

Behavioral

27. How do you answer "Tell me about yourself" in a software engineer interview?

Behavioral Easy Intern, Junior

Give a 60-90 second summary in three parts: what you do now, one or two achievements that show your strengths, and why this role is the logical next step.

This question is an invitation to set the agenda for the interview, not a request for your life story. Interviewers want a short, relevant summary that tells them where to dig deeper.

Use a present, past, future shape. Present: your current role or studies and the kind of work you do. Past: one or two concrete things you built or improved, with a result. Future: why this team and role is the next step for you.

Tailor it to the job description. For a backend role, lead with APIs and data work; for a frontend role, with user-facing features and performance. Practise it out loud until it sounds natural rather than memorised.

Key points

- Keep it to about 60-90 seconds
- Present, past, future structure
- Pick achievements relevant to this job
- End with why you want this role

28. How do you answer "Why should we hire you?" as a developer?

Behavioral Easy Intern, Junior

Connect two or three of your strongest, provable skills directly to the needs in the job description, give a short piece of evidence for each, and show genuine interest in the team's work.

This is a closing pitch. The interviewer wants a confident, specific summary of why you fit this role, not a list of adjectives like "hard-working" and "passionate".

Before the interview, pick the two or three requirements in the job description that matter most and match each one with evidence from your experience: a project, a metric, a responsibility you held.

Finish with motivation that is specific to the company: its product, technical challenges or engineering culture. Keep the whole answer under a minute.

Key points

- Match your strengths to their stated needs
 - Back each claim with a concrete example
 - Show specific interest in the company
 - Keep it under a minute
-

29. What are you currently doing to grow as a software engineer?

Behavioral Easy Intern, Junior

Give specific evidence of deliberate learning through projects, feedback, reading, experimentation or mentoring rather than listing technologies.

Interviewers want to know whether your growth is intentional and connected to your current level and future responsibilities.

A strong answer combines a skill you are improving, how you practice it and evidence that your behavior has changed.

For senior candidates, include growth beyond coding: architecture, communication, mentoring, product understanding or leadership.

Key points

- Choose specific growth areas
 - Practice deliberately
 - Use feedback
 - Show evidence of improvement
-

30. Why do you want to work for our company?

Behavioral Easy Intern, Junior

Connect something specific about the company, product, engineering challenges or team to your own experience and career goals.

A good answer demonstrates that you understand what the company actually does and why the role matters to you.

Choose two or three specific reasons and connect each to evidence from your background or interests.

Avoid generic statements that could be copied into any interview.

Key points

- Research the company
 - Be specific
 - Connect to your experience
 - Show realistic motivation
-

31. How do you answer "Tell me about a difficult project you worked on"?

Behavioral Medium Junior, Mid-level

Choose a real project with a technical or organisational challenge, explain your specific role and decisions with the STAR method, and finish with the result and what you learned.

The interviewer is testing how you handle complexity and pressure, and whether you can explain technical work to someone who was not there. The difficulty itself matters less than how you reasoned through it.

Pick a project where you made meaningful decisions. Briefly set the context, then spend most of the answer on

your actions: how you broke the problem down, what options you considered, why you chose one, and how you worked with others.

Close with an honest result. If something went wrong, say so and explain what you would do differently. Senior candidates should also show how they reduced risk for the team, for example by phasing a migration or adding monitoring.

Key points

- Use STAR: Situation, Task, Action, Result
 - Spend most of the time on your actions
 - Say "I" for your work and "we" for the team's
 - Include what you learned
-

32. How do you answer "Tell me about a production bug you fixed"?

Behavioral Medium Junior, Mid-level

Walk through how you noticed the issue, limited the impact, found the root cause, fixed it, and what you changed afterwards so the same class of bug could not happen again.

This question tests your debugging process and how you behave under pressure. Interviewers want a calm, methodical story, not a hero story.

A strong answer has a clear order: detection (alert, logs, user report), mitigation first (rollback, feature flag, scaling), investigation (reproducing it, narrowing the cause with logs, metrics or git bisect), the actual fix, and prevention (tests, monitoring, a post-mortem).

Be specific about the technical cause, at a level the interviewer can follow. A memorable detail, such as a timezone edge case or a missing index that only hurt at peak traffic, makes the story credible.

Key points

- Mitigate impact before hunting for the root cause
 - Explain how you narrowed the cause down
 - Finish with prevention: tests, alerts, process changes
 - Stay blameless
-

33. Tell me about a mistake you made and what you learned from it.

Behavioral Medium Junior, Mid-level

Choose a real mistake, take ownership, explain the impact, what you did to recover, and what you changed so it would not happen again.

Interviewers are testing accountability, self-awareness and whether you learn from mistakes rather than whether you have a perfect record.

Choose a mistake that is meaningful but safe to discuss. Spend more time on your response and learning than on the mistake itself.

A strong answer shows prevention: a test, checklist, monitoring, review practice, documentation or process improvement.

Key points

- Take ownership
 - Explain impact honestly
 - Show recovery
 - Explain prevention
-

34. Tell me about a time you received difficult or critical feedback.

Behavioral Medium Junior, Mid-level

Describe feedback that was uncomfortable but useful, explain how you responded, and show the measurable or observable improvement that followed.

The interviewer wants evidence that you can accept feedback without becoming defensive and turn it into better performance.

Choose feedback that was genuinely actionable. Explain what you initially thought, what you understood after reflection, and what you changed.

For senior candidates, show how the experience changed the way you give feedback to others as well.

Key points

- Listen without defensiveness
 - Ask clarifying questions
 - Act on feedback
 - Show improvement
-

35. Tell me about a project or decision that failed.

Behavioral Medium Junior, Mid-level

Choose a genuine failure, explain why it failed, what you owned, how you recovered, and what you changed in future projects.

Good answers demonstrate resilience and learning rather than trying to make the failure sound like a success.

Be precise about the original assumption, the signal that showed it was wrong, and the decision you made once you knew.

Senior candidates should explain how they changed team practices or reduced similar risk afterwards.

Key points

- Be honest
 - Identify the failed assumption
 - Recover quickly
 - Show systemic learning
-

36. Tell me about a time you had to deliver under a very tight deadline.

Behavioral Medium Junior, Mid-level

Show how you prioritized the essential outcome, communicated trade-offs, reduced risk and delivered without pretending everything could be done.

The interviewer is testing prioritization and communication under pressure, not willingness to work unlimited hours.

Explain what you cut, deferred or simplified and why. If quality or risk was temporarily accepted, explain how you contained it.

Strong answers include early communication when the original scope was unrealistic.

Key points

- Define the must-have outcome
 - Prioritize ruthlessly
 - Communicate trade-offs
 - Protect critical quality
-

37. Tell me about a time you disagreed with a code review comment.

Behavioral Medium Junior, Mid-level

Show that you can defend a technical decision with evidence while remaining open to a better argument.

Code review is collaborative engineering, not a contest of authority.

Explain the concern behind the comment before explaining your position. A strong answer may end with you changing your implementation.

If the disagreement remained, describe how you used project conventions, documentation, tests or a short experiment to resolve it.

Key points

- Understand the concern
 - Use evidence
 - Stay respectful
 - Accept better reasoning
-

38. Why are you looking to leave your current company?

Behavioral Easy Mid-level, Senior

Keep the answer professional and future-focused: explain what you are looking for next without attacking your current or former employer.

Interviewers are checking motivation, professionalism and whether your expectations match the role.

Focus on positive reasons such as greater ownership, a different technical challenge, product scale or opportunities to grow.

If there were genuine difficulties, describe them factually and briefly, then return to what you want to build next.

Key points

- Stay positive
 - Be truthful
 - Focus on the future
 - Connect to the role
-

39. How do you answer "Tell me about a time you disagreed with a teammate"?

Behavioral Medium Mid-level, Senior

Describe a real technical or process disagreement, show that you listened and argued with evidence rather than opinion, explain how a decision was reached, and how you supported it afterwards even if it was not your idea.

Interviewers want to know whether you can disagree productively. Teams need people who raise concerns, but also people who commit once a decision is made.

Choose a disagreement about something that mattered, such as an architecture choice, a deadline or code review standards. Avoid personal conflicts. Show that you first understood the other person's reasoning, then brought data: a benchmark, a prototype, an incident report.

The outcome does not need to be that you won. "We went with their approach, and I helped make it work" is a strong answer if you explain what you learned.

Key points

- Pick a professional, not personal, disagreement
 - Show you listened first
 - Use evidence: prototypes, data, constraints
 - Disagree and commit
-

40. Tell me about a time you took ownership of something that was not explicitly assigned to you.

Describe a problem you noticed, why you decided to act, how you coordinated with others, and the impact of taking ownership.

This question separates people who only complete assigned tickets from engineers who notice and solve important problems.

Do not describe work you simply volunteered for without explaining why it mattered to users, reliability, delivery or the team.

Show initiative without suggesting that you ignored ownership boundaries or other people.

Key points

- Identify an unowned problem
 - Explain why it mattered
 - Coordinate before changing shared systems
 - Show impact
-

41. Tell me about a time you had to work with unclear or changing requirements.

Explain how you identified ambiguity, clarified the important decisions, made reasonable assumptions and kept delivery moving.

Real engineering work often starts with incomplete information. Interviewers want to see structured thinking rather than waiting passively for perfect requirements.

Explain which questions you asked, which assumptions you documented and which decisions you intentionally kept reversible.

Strong candidates distinguish between uncertainty that must be resolved now and uncertainty that can be handled later.

Key points

- Clarify goals
 - Document assumptions
 - Prioritize unknowns
 - Keep decisions reversible where possible
-

42. Tell me about a time you had several important tasks competing for your attention.

Explain how you assessed urgency, impact, dependencies and risk, then communicated the resulting priorities.

Strong engineers do not prioritize only by who asks first. They consider customer impact, deadlines, dependencies, severity and reversibility.

Show how you made the priority decision visible to stakeholders so expectations were aligned.

For senior candidates, explain how you helped the team prioritize rather than managing only your own queue.

Key points

- Assess impact and urgency
 - Consider dependencies
 - Communicate priorities
 - Revisit when conditions change
-

43. Tell me about a time you disagreed with your manager or technical lead.

Behavioral Medium Mid-level, Senior

Show respectful disagreement, evidence-based reasoning, willingness to listen and commitment after the final decision.

Disagreeing professionally is healthy. The key is to challenge the decision rather than the person.

Explain what evidence you brought, what you learned from their perspective and how the final decision was made.

If the decision went against your recommendation, explain how you supported it afterwards.

Key points

- Challenge ideas, not people
 - Use evidence
 - Listen to constraints
 - Disagree and commit
-

44. Tell me about a process you improved for your team.

Behavioral Medium Mid-level, Senior

Identify a recurring source of wasted time or risk, make a focused improvement, and demonstrate its impact.

This question looks for engineers who improve the system around the code, not just the code itself.

Choose a process with repeated cost: deployment, code review, testing, incident response, onboarding or local setup.

Quantify the improvement when possible, but a clear reduction in errors or friction is also useful.

Key points

- Identify recurring pain
 - Make a focused change
 - Get team buy-in
 - Measure impact
-

45. Tell me about a repetitive task you automated.

Behavioral Medium Mid-level, Senior

Explain the manual problem, why automation was worthwhile, how you implemented it safely and what time or error reduction resulted.

Automation is valuable when it reduces recurring human effort or prevents avoidable errors.

Explain why you chose the level of automation you did rather than building a complicated system for a small problem.

Mention maintenance, failure handling and how users or teammates adopted the automation.

Key points

- Identify recurring cost
 - Choose proportional automation
 - Handle failures
 - Measure savings
-

46. Tell me about a time you explained a complex technical concept to a non-technical person.

Show how you adapted your explanation to the audience, focused on the outcome and avoided unnecessary technical jargon.

Senior engineers often communicate with product, support, sales and leadership. The ability to make technical risk understandable is important.

Explain the audience, what they needed to decide and how you translated the technical issue into business impact.

Avoid presenting technical simplification as hiding important information; explain the right level of detail for the decision.

Key points

- Understand the audience
 - Explain impact first
 - Avoid unnecessary jargon
 - Confirm understanding
-

47. Tell me about a problem you prevented before it reached production.

Show how you noticed a risk early, validated it, communicated it and added a safeguard.

Preventative engineering demonstrates maturity because the best incident is often the one that never happens.

Explain the signal that made you suspicious and how you verified the risk instead of relying on intuition alone.

Finish with the lasting safeguard: test, validation, monitoring, review, rollout control or design change.

Key points

- Spot risk early
 - Validate before escalating
 - Communicate clearly
 - Add a lasting safeguard
-

48. Tell me about a time you had to work with a difficult stakeholder.

Describe how you understood the stakeholder's goals, handled disagreement professionally and reached a workable outcome.

The interviewer is evaluating communication and emotional maturity, not whether you can complain about difficult people.

Describe the stakeholder's legitimate concern before explaining the conflict. This makes the story balanced and credible.

Focus on shared goals, explicit trade-offs and communication rather than personality differences.

Key points

- Understand their goals
 - Stay professional
 - Use facts and options
 - Find shared outcomes
-

49. Tell me about a technical debt decision you made.

Explain why you accepted, reduced or paid down technical debt and how you managed the associated

risk.

Technical debt is not automatically bad. Interviewers want to see whether you understand its cost and deliberately manage it.

Describe the business constraint, the engineering risk and why the chosen approach was appropriate at that time.

Strong answers include a trigger or plan for addressing the debt instead of leaving it invisible.

Key points

- Identify the debt
 - Understand its cost
 - Make the trade-off explicit
 - Create a follow-up trigger
-

50. Tell me about a time you introduced a new technology or tool to a project.

Behavioral Hard Mid-level, Senior

Start with the problem, evaluate alternatives, validate the technology with a small experiment and explain how you managed adoption and risk.

Interviewers want to know that you choose technology because it solves a problem, not because it is fashionable.

Discuss the existing approach, why it was insufficient, alternatives considered, proof of concept and operational implications.

Senior candidates should include how they helped the team adopt and maintain the new technology.

Key points

- Start with the problem
 - Evaluate alternatives
 - Prototype before committing
 - Plan adoption and maintenance
-

51. Tell me about a time you mentored or helped another developer grow.

Behavioral Medium Senior

Explain how you identified what the developer needed, coached rather than simply solved the problem, and measured their progress.

Mentoring is about increasing another engineer's ability to solve problems independently.

Strong answers describe questions, pairing, feedback, documentation or progressively larger ownership rather than simply giving answers.

Senior candidates should show patience and adaptation to the individual developer's level.

Key points

- Understand their starting point
 - Teach reasoning, not only solutions
 - Give actionable feedback
 - Increase independence
-

52. Tell me about a time you influenced a technical decision without having formal authority.

Behavioral Hard Senior

Show how you built trust, understood competing concerns, used evidence and helped the group reach a decision.

Senior engineers often influence architecture and engineering practices without being the manager or final decision-maker.

The strongest stories show persuasion through clarity and evidence rather than relying on seniority or force of personality.

Explain what happened after the decision and whether you supported the team's final choice.

Key points

- Build alignment
 - Use evidence
 - Understand other constraints
 - Support the final decision
-

53. Tell me about a time you had to choose between two good technical approaches.

Behavioral Hard Senior

Explain the constraints, alternatives, trade-offs, decision criteria and why the chosen option was appropriate for the context.

Senior engineering decisions rarely have a universally correct answer. Interviewers want to see explicit trade-off reasoning.

Compare options using factors such as reliability, complexity, performance, cost, team expertise, delivery time and future change.

A strong answer acknowledges what you gave up by choosing the preferred option.

Key points

- Define decision criteria
 - Compare alternatives
 - Make trade-offs explicit
 - Choose for the context
-

54. Tell me about a major production incident you helped handle.

Behavioral Hard Senior

Cover detection, communication, mitigation, investigation, recovery and the changes made afterwards to reduce recurrence.

Incident questions test calm decision-making and ownership under pressure.

Separate immediate mitigation from root-cause analysis. During an incident, restoring service is usually more important than finding the perfect explanation immediately.

For senior candidates, include how you coordinated people, communicated status and improved the system after the incident.

Key points

- Protect users first
 - Communicate clearly
 - Investigate systematically
 - Prevent recurrence
-

55. Tell me about a time you decided not to use a technology or framework.

Behavioral Hard Senior

Explain why the technology looked attractive, what problem it actually solved, and why its cost or complexity was not justified in your context.

Knowing when not to add technology is an important engineering skill.

A strong answer compares the expected benefit against learning cost, operational complexity, team capability, migration cost and long-term maintenance.

Avoid making the decision sound ideological; show that you would reconsider if the constraints changed.

Key points

- Define the actual problem
 - Compare total cost
 - Avoid unnecessary complexity
 - State when the decision could change
-

56. Tell me about a time you had to balance speed of delivery with technical quality.

Behavioral Hard Senior

Explain the deadline, the quality risks, what you protected, what you simplified and how you managed any remaining debt.

Engineering teams constantly balance delivery and quality. Strong answers show deliberate trade-offs rather than treating one as always correct.

Separate reversible shortcuts from dangerous shortcuts. Never present security, data integrity or critical reliability risks as acceptable just to ship faster.

Explain what you documented or scheduled for follow-up.

Key points

- Identify non-negotiable quality
 - Make trade-offs explicit
 - Contain risk
 - Track follow-up work
-

57. Tell me about a technical decision you made that turned out to be wrong.

Behavioral Hard Senior

Explain the original reasoning, the evidence that changed your view, how you corrected course and what decision-making practice improved afterwards.

This tests intellectual honesty and the ability to update your beliefs when reality disagrees with your assumptions.

Do not rewrite the story as if the decision was obviously correct. Explain what information was available at the time and what you missed.

The strongest answers show a change in process, such as better experiments, metrics or review criteria.

Key points

- Own the decision
 - Explain original assumptions
 - Recognize evidence
 - Improve the decision process
-

58. Tell me about a project you led from idea to production.

Behavioral Hard Senior

Walk through how you shaped the problem, planned the work, aligned stakeholders, managed risks and measured the outcome.

For senior engineers, leadership means creating clarity and reducing risk, not simply assigning tasks.

Cover discovery, design, execution, testing, rollout, monitoring and follow-up. Make your individual leadership contribution clear.

Include a moment where you made a meaningful trade-off or changed the plan based on new information.

Key points

- Define the problem
 - Align stakeholders
 - Plan and de-risk
 - Lead delivery
 - Measure outcome
-

59. Tell me about a time you worked with a teammate who was struggling.

Behavioral Hard Senior

Show empathy, clear expectations, practical support and appropriate escalation rather than blame.

The interviewer wants to see whether you can help a teammate without becoming their manager or doing their work for them.

Start by understanding the cause: unclear requirements, missing knowledge, workload, personal circumstances or process problems.

For senior candidates, explain how you balanced support with accountability and involved the appropriate lead when necessary.

Key points

- Understand the cause
 - Set clear expectations
 - Offer targeted support
 - Maintain accountability
-

These answers are written for real interview conversations. Prefer explaining a trade-off out loud over reciting a definition. Full explanations and related questions: <https://www.fredevtemplates.com/interview-questions/mobile>