Next JS Interview Questions & Answers

A short, App Router-first study PDF for interviews.

What this PDF covers

- Rendering basics: SSR vs SSG vs ISR vs CSR
- App Router mental model (Server vs Client Components)
- Caching + revalidation: the gotchas interviewers test
- High-frequency Q&A; across Fresher, Experienced, Senior levels
- Scenario prompts + mini coding tasks

Tip: Practice the short answers out loud, then expand with 2-3 bullets.

Quick Cheat Sheet

SSR vs SSG vs ISR vs CSR

Mode	Meaning	Best for	One-liner tradeoff
SSR	Render per request	Auth pages, highly dynamic	Fresh but costs per request
SSG	Render at build time	Marketing/docs	Fast but can go stale
ISR	SSG + background refresh	Blogs/catalogs	Fast + fresh-ish with revalidate strategy
CSR	Fetch in browser	Heavy interactivity	More JS + weaker initial UX/SEO

App Router mental model (rules of thumb)

- **Server Components by default**: fetch data, read cookies/headers, keep secrets server-side.
- **Client Components only when needed**: hooks/state, event handlers, browser APIs, client-only libraries.
- **Boundary rule**: client components cannot import server components - keep server above, pass serializable props down.

Caching defaults + #1 gotcha

Be explicit about caching. The most common interview bug: data updates in the database, but the UI stays stale because the wrong cache layer was invalidated (data vs route output vs client router cache).

- After mutations, invalidate: **revalidateTag** for tagged data or **revalidatePath** for a specific route.
- If you read request-specific data (cookies/headers), your route likely becomes dynamic.
- Keep cache tags consistent (e.g., products, product:123, category:shoes).

High-Frequency Next JS Interview Questions & Answers

Freshers (quick fundamentals)

Question: What is the App Router?
Short answer: The App Router is the modern routing system under app/ with nested layouts and server-first rendering.
Deep answer:
- Routes are folders with page.tsx; shared UI is layout.tsx.
- Special files: loading.tsx, error.tsx, not-found.tsx.
- Server Components are default; client is opt-in via 'use client'.

Common misconception: Wrong: 'App Router is just Pages Router with a new folder name.'
Follow-up: Explain layout vs template and how it affects state.

Question: What does 'use client' do?
Short answer: It marks a file as a Client Component so you can use hooks, event handlers, and browser APIs.
Deep answer:
- Creates a client boundary; keep it small for bundle size.
- Client components cannot import server components.
- Prefer Server parent -> Client leaf for performance.

Common misconception: Wrong: 'It makes the whole app client-side.'
Follow-up: How do you reduce client JS in a complex UI?

Question: What is a Route Handler?
Short answer: A Route Handler is an API endpoint in app/**/route.ts using Web Request/Response.
Deep answer:
- Use for webhooks, BFF endpoints, and integrations.
- Prefer over pages/api in App Router apps.
- Choose runtime (edge/node) based on dependencies.

Common misconception: Wrong: 'APIs only belong in pages/api.'
Follow-up: When would you split APIs into a separate service?

Question: What is hydration?
Short answer: Hydration is when React attaches event handlers to server-rendered HTML in the browser.
Deep answer:
- Mismatches happen when server and client render different initial markup.
- Common causes: Date/locale randomness or window branching during render.
- Fix by making initial render deterministic or moving logic to useEffect.

Common misconception: Wrong: 'Hydration warnings are harmless.'
Follow-up: Give a step-by-step plan to debug a mismatch.

Experienced (caching, routing, actions)

Question: Explain loading.tsx vs Suspense fallback.
Short answer: loading.tsx is segment-level loading UI; Suspense fallback is component-level and helps stream parts independently.
Deep answer:
- Use loading.tsx for a route segment skeleton.
- Use Suspense boundaries to isolate slow widgets.
- Avoid one giant boundary that blocks everything.

Common misconception: Wrong: 'loading.tsx replaces Suspense.'
Follow-up: Where would you place boundaries on a 6-widget dashboard?

Question: revalidatePath vs revalidateTag - when do you use each?
Short answer: Use revalidatePath to refresh a route; use revalidateTag to refresh all cached data requests with a tag.
Deep answer:
- Tags scale better across many pages.
- Call invalidation right after successful mutations.
- Design tags with stable names and limited cardinality.

Common misconception: Wrong: 'Next refreshes automatically after POST.'
Follow-up: Design tags for product list + product detail pages.

Question: What are Parallel Routes and Intercepting Routes used for?
Short answer: They enable multiple slots in one layout and URL-driven modals with deep linking.
Deep answer:
- Parallel Routes define slots like @modal or @sidebar.
- Intercepting routes let a detail route render as a modal over a list.
- Refresh/back should work as users expect.

Common misconception: Wrong: 'Modals should be only client state.'
Follow-up: Explain (.) vs (..) interception with an example.

Question: Server Actions: key constraints and best use case?
Short answer: Server Actions run on the server and are great for first-party form mutations, but require serializable inputs and strong validation.
Deep answer:
- Use form action={myAction} for CRUD flows.
- Inputs must be serializable (or FormData).
- Still enforce auth, validation, rate limits, and revalidation after writes.

Common misconception: Wrong: 'Server Actions replace public APIs.'
Follow-up: When would you prefer a Route Handler over an Action?

Senior (tradeoffs and architecture)

Question: How do you prevent cache leakage in auth/personalized pages?
Short answer: Treat personalized output as dynamic or carefully scope caching by user/tenant, otherwise data can leak across sessions.
Deep answer:
- Avoid global caching for user-specific responses.
- Keep auth checks server-side (actions/handlers/layout).
- Use tags for shared public data and separate them from private data.

Common misconception: Wrong: 'Caching is always safe because it is server-side.'
Follow-up: Describe a guardrail you would implement to prevent this class of bug.

Question: Edge vs Node runtime: when do you choose which?
Short answer: Use edge for fast routing/token checks; use node for DB access and full library support.
Deep answer:
- Edge is low latency but limited Node APIs.
- Node supports DB drivers and heavier work.
- Common pattern: edge gate + node fetch/mutate.

Common misconception: Wrong: 'Edge is always faster overall.'
Follow-up: How would you design auth so middleware stays lightweight?

Scenarios + Mini Coding Tasks

Scenario prompts (practice explaining your debugging steps)

- **Stale UI after mutation:** DB updated but list page still shows old data. Explain which cache layers you would check and what invalidation you would add.
- **Modal deep link breaks on refresh:** modal works via client state, but refreshing closes it. Explain how Parallel + Intercepting routes fix this.
- **Production-only Server Action failure behind proxy:** works locally, fails in production. Explain what headers/origin checks you would inspect and how you would safely configure allowed origins.

Mini coding tasks (pseudocode)

Task: Build a Route Handler GET /api/health
- Create app/api/health/route.ts.
- Export async function GET() that returns Response.json({ ok: true }).

Task: Add tag-based caching and invalidate after update
- Fetch with next: { tags: ['products'] } and cache: 'force-cache'.
- After mutation, call revalidateTag('products').

Task: Make time rendering hydration-safe
- Render placeholder on server.
- In a small client component, set time in useEffect.

More practice: Read the full study guide on the site.
Home: https://interviewquestions.guru