

N

Frontend · most asked

Next.js interview questions

23 questions · 9 easy · 10 medium · 4 hard. Open one to see the answer, a code example and the likely follow-up, then mark it as known or to review.

1

What is Next.js and why use it instead of plain React?

Easy



#basics

Next.js is a **React framework**. React gives you components; Next.js adds everything around them that a production app needs:

- file-based **routing** with nested layouts
- **server rendering** (SSR, static generation, streaming) for fast first loads and SEO
- **Server Components** and **Server Actions**, so data access and mutations live next to the UI
- API endpoints (**route handlers**) in the same project
- built-in image, font and script optimisation, code splitting and caching
- zero-config bundling (Turbopack) and easy deployment

The short answer: **better performance and SEO with less setup** than a client-only React app built with Vite.

Likely follow-up: When would a plain Vite SPA be the better choice?

2

Explain SSR vs SSG vs ISR vs CSR.

Easy



#rendering #ssr

- **CSR** (client-side rendering): the server sends an almost empty page and JavaScript builds it in the browser. Slow first paint, weak SEO, fine for dashboards behind login.
- **SSG** (static site generation): HTML is built **once at build time** and served from a CDN. Fastest, but data is only as fresh as the last build.
- **ISR** (incremental static regeneration): static pages that are **regenerated in the background** after a time (`revalidate: 60`) or on demand, so you get static speed with fresh-ish data.

- **SSR** (server-side rendering): HTML is generated **on every request**. Always fresh and personalised, but costs server time per request.

In the App Router you do not pick a function per page: a route is static by default and becomes dynamic when it uses request data (`cookies()`, `headers()`, `searchParams`) or uncached fetches.

? **Likely follow-up:** Which would you use for a blog, a product page and a user dashboard?

3

Server Components vs Client Components: what does `'use client'` do?

Medium



#server components #app router

In the App Router, every component is a **Server Component** by default. It runs on the server, can be `async`, can read databases and secrets directly, and ships no JavaScript to the browser.

`'use client'` at the top of a file marks a **boundary**: that file and everything it imports become Client Components. They are still pre-rendered to HTML on the server, then hydrated so they can use state, effects, event handlers and browser APIs.

- Keep `'use client'` as **low in the tree as possible** (the button, not the page)
- Server Components can render Client Components and pass them serialisable props
- To put server content inside a client component, pass it as `children`
- `'use client'` does not mean "client only", it means "also runs on the client"

Example

```
// app/products/page.tsx (Server Component)
export default async function Page() {
  const products = await db.product.findMany() // runs on the server only
  return <ProductList products={products} />
}

// app/products/like-button.tsx
'use client'
export function LikeButton() {
  const [liked, setLiked] = useState(false)
  return <button onClick={() => setLiked(!liked)}>{liked ? 'Liked' : 'Like'}</button>
}
```

🔗 Likely follow-up: Can a Client Component import a Server Component?

4

How does routing work in the App Router? What are layouts?

Easy



#routing #app router

Folders inside `app/` define URL segments, and special files define what each segment renders:

- `page.tsx`: the UI for that route (makes it publicly reachable)
- `layout.tsx`: shared UI that wraps child routes and **keeps its state across navigation**
- `template.tsx`: like a layout but remounts on every navigation
- `loading.tsx`, `error.tsx`, `not-found.tsx`: automatic Suspense and error boundaries

Layouts nest: the root layout (with `<html>` and `<body>`) wraps a section layout, which wraps the page.

Useful conventions: `(group)` folders organise routes without affecting the URL, `_private` folders are ignored, and `@slot` folders create parallel routes. `<Link>` gives client-side navigation with prefetching.

5

How do dynamic routes and `generateStaticParams` work?

Medium



#routing #ssg

A folder in square brackets is a **dynamic segment**: `app/blog/[slug]/page.tsx` matches `/blog/anything`. `[...slug]` is a catch-all and `[ [...slug]]` an optional catch-all.

The page receives `params`, which since Next.js 15 is **a Promise you must await**.

`generateStaticParams` returns the list of params to **pre-render at build time** (SSG).

Paths not in the list are rendered on demand on first request and then cached, unless you set `dynamicParams = false` to return 404 instead.

Example

```
// app/blog/[slug]/page.tsx
export async function generateStaticParams() {
  const posts = await getPosts()
  return posts.map((p) => ({ slug: p.slug }))
}

export default async function Post({ params }: { params: Promise<{ slug: string }> }) {
  const { slug } = await params
  const post = await getPost(slug)
  if (!post) notFound()
  return <article><h1>{post.title}</h1></article>
}
```

6

How do you fetch data in the App Router?

Medium



#data fetching #server components

In **Server Components** you just `await` the data: `fetch`, an ORM query, or a file read. No `useEffect`, no API route in between, and secrets stay on the server.

- Run independent requests **in parallel** with `Promise.all` to avoid waterfalls
- Wrap slow parts in `<Suspense>` so the rest of the page streams first
- Deduplicate shared calls with React `cache()` (`fetch` with the same URL is deduped per request automatically)

On the client, use a library like **TanStack Query** or **SWR** for data that changes with user interaction, or pass a promise from a Server Component and read it with `use()`.

Fetch on the server by default, and go client-side only for highly interactive data.

Example

```
export default async function Dashboard() {
  const [user, orders] = await Promise.all([getUser(), getOrders()]) //
  parallel
  return (
    <>
      <h1>Hi {user.name}</h1>
      <Suspense fallback=<p>Loading stats...</p>>
        <SlowStats userId={user.id} />
      </Suspense>
    </>
  )
}
```

? Likely follow-up: How do you avoid request waterfalls?

7

How does caching and revalidation work in Next.js?

Hard



#caching #data fetching

Since **Next.js 15**, caching is **opt-in**: `fetch` is not cached by default and GET route handlers are dynamic.

Ways to cache and refresh:

- **Time-based**: `fetch(url, { next: { revalidate: 3600 } })` or `export const revalidate = 3600` on a route
- **On-demand**: tag data (`next: { tags: ['posts'] }`) and call `revalidateTag('posts')` or `revalidatePath('/blog')` after a mutation
- **Cache Components** (Next.js 16): the `'use cache'` directive caches a function, component or page, with `cacheLife` and `cacheTag` controlling lifetime and invalidation

Static routes are also cached as HTML at build time, and the client keeps a **router cache** of visited segments. `router.refresh()` refetches the current route.

Example

```
// time-based
const res = await fetch('https://api.example.com/posts', {
  next: { revalidate: 60, tags: ['posts'] },
})

// Next.js 16 with cacheComponents enabled
async function getProducts() {
  'use cache'
  cacheLife('hours')
  cacheTag('products')
  return db.product.findMany()
}
```

? Likely follow-up: What changed about caching defaults between Next.js 14 and 15?

8

What are Server Actions?

Medium



#server actions #mutations

A **Server Action** is an async function marked with `'use server'` that runs on the server but can be called from the client, usually as a form's `action` or from an event handler. Next.js creates the POST endpoint for you.

They replace most "write" API routes: validate input, write to the database, then call `revalidatePath` / `revalidateTag` or `redirect`.

- Forms work **before JavaScript loads** (progressive enhancement)
- Pair with `useActionState` for errors and pending state, `useOptimistic` for instant UI
- Treat them as **public endpoints**: always check authentication and validate input (Zod), because anyone can call them

Example

```
// app/todos/actions.ts
'use server'
export async function addTodo(formData: FormData) {
  const session = await auth()
  if (!session) throw new Error('Unauthorized')
  const title = z.string().min(1).parse(formData.get('title'))
  await db.todo.create({ data: { title, userId: session.user.id } })
  revalidatePath('/todos')
}

// in a component
<form action={addTodo}><input name="title" /><button>Add</button></form>
```

? Likely follow-up: Server Action or route handler: how do you choose?

9

What are route handlers and when do you use them?

Easy



#api #routing

A **route handler** is a `route.ts` file that exports functions named after HTTP methods (`GET`, `POST`, `PUT`, `DELETE` ...). They use the standard Web `Request` and `Response` APIs (plus `NextRequest` / `NextResponse` helpers).

Use them when **something other than your own React UI** calls the server:

- webhooks (Stripe, GitHub)
- a public or mobile API
- OAuth callbacks, file downloads, RSS feeds, streaming responses

For mutations from your own pages, Server Actions are usually simpler. A folder cannot have both `page.tsx` and `route.ts`.

Example

```
// app/api/users/[id]/route.ts
export async function GET(_req: Request, { params }: { params: Promise<{
id: string }> }) {
  const { id } = await params
  const user = await db.user.findUnique({ where: { id } })
  if (!user) return Response.json({ error: 'Not found' }, { status: 404 })
  return Response.json(user)
}
```

10

What is middleware (now called proxy) in Next.js?

Medium



#middleware #auth

It is code that runs **before a request reaches a route**. In Next.js 16 the file was renamed from `middleware.ts` to `proxy.ts` (same idea, clearer name), and it runs on the Node.js runtime.

Typical uses:

- redirecting unauthenticated users to `/login`
- rewrites for A/B tests, i18n locale detection, multi-tenant subdomains
- setting headers or cookies

A `matcher` config limits which paths it runs on. Keep it fast and light: it runs on many requests. Use it for `optimistic checks` (is there a session cookie?), and still verify authorisation where data is read, since a proxy can be misconfigured or bypassed.

Example

```
// proxy.ts (middleware.ts before Next.js 16)
import { NextResponse, type NextRequest } from 'next/server'

export function proxy(request: NextRequest) {
  if (!request.cookies.has('session')) {
    return NextResponse.redirect(new URL('/login', request.url))
  }
  return NextResponse.next()
}

export const config = { matcher: ['/dashboard/:path*'] }
```

11

How do you handle metadata and SEO in the App Router?

Easy



#seo #metadata

Export a `metadata` object from a `layout.tsx` or `page.tsx` for static values, or an async `generateMetadata` function for values that depend on data (a product title). Metadata from layouts is merged with the page's, and a `title.template` like `'%s | Shop'` keeps titles consistent.

Other built-ins:

- `sitemap.ts` and `robots.ts` generate those files
- `opengraph-image.tsx` generates social preview images
- `alternates.canonical` for canonical URLs

SEO also benefits from server rendering itself: crawlers get full HTML, plus semantic markup and fast Core Web Vitals.

Example

```
export async function generateMetadata({
  params: { slug: string },
}): Promise<Metadata> {
  const product = await getProduct((await params).slug)
  return {
    title: product.name,
    description: product.summary,
    openGraph: { images: [product.image] },
  }
}
```

12

What do `next/image` and `next/font` do?

Easy



#performance #images

`next/image` (`<Image>`) optimises images automatically:

- resizes and serves modern formats (WebP/AVIF) per device via `srcset`
- **lazy-loads** by default
- requires `width` / `height` (or `fill`) to **prevent layout shift** (CLS)
- `priority` (or `preload` in Next.js 16) for the hero / LCP image
- remote images need an allow list in `next.config` (`images.remotePatterns`)

`next/font` downloads Google or local fonts **at build time** and self-hosts them, so there is no request to Google at runtime. It sets `font-display` and size-adjusted fallbacks so text does not jump when the font loads.

13

What do `loading.tsx` and `error.tsx` do?

Easy



#routing #errors

They are special files that wrap a route segment automatically:

`loading.tsx` becomes the fallback of a **Suspense boundary** around the page. The layout shows instantly and the loading UI (a skeleton) is visible while the page's data loads, then streams in.

`error.tsx` becomes an **error boundary** around the segment. It must be a Client Component and receives `error` and `reset` (to retry). The layout stays interactive while **only the broken segment shows the fallback**.

- `error.tsx` does not catch errors in the layout of the same segment; the parent's boundary does
- `global-error.tsx` handles errors in the root layout
- `not-found.tsx` renders when you call `notFound()`

Example

```
// app/dashboard/error.tsx
'use client'
export default function Error({ error, reset }: { error: Error; reset: () => void }) {
  return (
    <div>
      <p>Could not load the dashboard.</p>
      <button onClick={() => reset()}>Try again</button>
    </div>
  )
}
```

14

What is streaming, and how does it work with Suspense?

Medium 🔥 #streaming #suspense #performance

Without streaming, SSR must finish **all** data fetching before sending any HTML, so the slowest query blocks the whole page.

With **streaming**, the server sends the HTML shell immediately (layout, header, fallbacks) and then streams in each `<Suspense>` section as its data resolves, in the same response. React hydrates sections as they arrive.

Benefits: faster **TTFB** and first paint, and slow widgets do not block fast ones.

You get it with `loading.tsx` (whole page) or by wrapping individual async Server Components in `<Suspense>` for **finer-grained loading states**.

🔗 Likely follow-up: What is Partial Prerendering?

15

Pages Router vs App Router: what are the main differences?

Medium



#app router #pages router

The **Pages Router** (`pages/`) is the older model: every page is a Client Component, data is loaded with `getServerSideProps`, `getStaticProps` and `getStaticPaths`, APIs live in `pages/api`, and shared layout goes in `_app.tsx`.

The **App Router** (`app/`) is the current recommended model:

- **Server Components** by default, with `async` components fetching data directly
- nested **layouts** that persist across navigation
- streaming, `loading.tsx` and `error.tsx` conventions
- Server Actions and route handlers
- finer caching control

Both can coexist in one project, so teams **migrate route by route**. The Pages Router is still supported but gets few new features.

? Likely follow-up: What replaces `getServerSideProps` in the App Router?

16

How do environment variables work in Next.js? What does `NEXT_PUBLIC_` do?

Easy



#config #security

Next.js loads `.env`, `.env.local`, `.env.development` and `.env.production` automatically (`.env.local` should be git-ignored).

By default, variables are **server-only**: available in Server Components, actions, route handlers and the proxy via `process.env`.

A `NEXT_PUBLIC_` prefix **inlines the value into the client bundle at build time**. That means:

- never put secrets in a `NEXT_PUBLIC_` variable
- changing it requires a rebuild, not just a restart

Import `'server-only'` in modules that touch secrets so an accidental client import fails the build.

17

How does navigation work: `<Link>`, `useRouter` and `redirect`?

Easy



#routing #navigation

`<Link href>` does **client-side navigation**: only the changed segments are fetched and rendered, layouts keep their state, and links in the viewport are **prefetched** in production.

- `useRouter()` from `next/navigation` (Client Components) for programmatic navigation: `push`, `replace`, `back`, `refresh`
- `usePathname()` and `useSearchParams()` read the current URL
- `redirect()` and `notFound()` work in Server Components, actions and route handlers; they **throw internally**, so do not wrap them in `try/catch`

Note: `next/router` is the Pages Router version and does not work in `app/`.

18

What causes a hydration mismatch in Next.js and how do you fix it?

Medium



#hydration #ssr

Client Components are rendered to HTML on the server, then React hydrates them in the browser. If the first client render **produces different markup** than the server, React reports a hydration error.

Typical causes in Next.js apps:

- `new Date()`, timezones and `toLocaleString` differing between server and browser
- reading `window`, `localStorage` or `navigator` during render
- invalid nesting such as `<div>` inside `<p>`
- browser extensions injecting attributes

Fixes:

- move browser-only logic into `useEffect` and render after mount
- `dynamic(() => import('./Map'), { ssr: false })` for components that cannot run on the server
- `suppressHydrationWarning` for a single unavoidable value (common on `<html>` for theme scripts)

19

How would you implement authentication in a Next.js app?

Hard



#auth #security

Usually with a library such as **Auth.js (NextAuth)**, **Clerk**, **Better Auth** or Supabase Auth, storing the session in an **httpOnly, secure cookie**.

Check auth at several layers:

- the **proxy** (middleware) for fast, optimistic redirects of signed-out users
- a **data access layer**: a `verifySession()` function called by every Server Component, Server Action and route handler that reads private data
- the UI only hides buttons; it is never the security boundary

The key point interviewers want: **authorise close to the data**, because Server Actions and route handlers are public endpoints and layouts do not re-render on every navigation.

20

How do you lazy-load components in Next.js?

Medium



#performance #code splitting

`next/dynamic` wraps `React.lazy` and `Suspense`: the component's JavaScript goes into a separate chunk loaded only when it renders.

- `loading` option for a placeholder
- `ssr: false` to skip server rendering for browser-only libraries (maps, charts, editors); this only works inside Client Components

Server Components are already split automatically, and their code never reaches the client, so dynamic imports matter mainly for **heavy Client Components**. You can also

`await import('lib')` inside an event handler to load a library only on click.

Example

```
'use client'
import dynamic from 'next/dynamic'

const Map = dynamic(() => import('./Map'), {
  ssr: false,
  loading: () => <p>Loading map...</p>,
})
```

21

How do you improve performance and Core Web Vitals in a Next.js app?

Hard



#performance #web vitals

Measure first (Lighthouse, Vercel Speed Insights, `useReportWebVitals`), then:

- **LCP**: static or cached rendering, `next/image` with priority/preload on the hero, `next/font`
- **CLS**: image dimensions, font fallbacks, reserved space for skeletons
- **INP**: ship less client JavaScript, keep `'use client'` at the leaves, `useTransition` for heavy updates

Also:

- fetch in parallel and stream slow parts with Suspense
- cache data and revalidate instead of rendering everything per request
- check bundle size with `@next/bundle-analyzer` and lazy-load heavy libraries
- use `next/script` with a sensible `strategy` for third-party scripts

The biggest lever is usually `less JavaScript on the client`.

22

How do you deploy a Next.js app? What are the options?

Medium



#deployment #devops

- **Vercel**: zero config, preview deployments per pull request, edge network, ISR and image optimisation handled for you
- **Self-hosted Node**: `next build` then `next start` on any server or container. `output: 'standalone'` produces a minimal folder ideal for Docker
- **Other platforms** (Netlify, Cloudflare, AWS via OpenNext or Amplify) through adapters
- **Static export**: `output: 'export'` gives plain HTML/CSS/JS for any static host, but no server features (no SSR, Server Actions, ISR, proxy)

When self-hosting with several instances, think about the `shared cache` (a custom cache handler, e.g. Redis) and image optimisation.

23

What are parallel and intercepting routes?

Hard



#routing #app router

Parallel routes use named slots (`@team`, `@analytics`) that a layout receives as props and renders side by side. Each slot has its own loading and error states and can navigate independently. `default.tsx` defines what a slot shows when it has no match.

Intercepting routes use `(.)`, `(..)` or `(...)` folder prefixes to show another route **inside the current layout** during client navigation.

The classic combination is a **modal with a shareable URL**: clicking a photo opens `/photo/1` as a modal over the feed, while a refresh or direct visit to the same URL shows the full photo page.