

Next.js

App Router, server components, rendering strategies, caching

1

What Next.js adds to React

Definition

Next.js is a React framework: file-based routing, server rendering, data fetching, caching, API routes, image/font optimisation and bundling in one tool.

App Router file conventions

app/	
layout.tsx	shared UI (wraps every page), keeps state across
navigation	
page.tsx	the "/" route
loading.tsx	Suspense fallback while the route loads
error.tsx	error boundary for the segment ('use client')
not-found.tsx	404 UI
blog/	
page.tsx	"/blog"
[slug]/page.tsx	"/blog/:slug" – dynamic segment
(marketing)/about/	route group: organises files, not part of the URL
api/users/route.ts	Route Handler: GET/POST/... → JSON

2

Server and Client Components

Server Component (default)	Client Component ('use client')
Runs on the server only; its code is not sent to the browser	Rendered on the server for HTML, then hydrated in the browser
Can be <code>async</code> , read the DB or secrets directly	Can use state, effects, event handlers, browser APIs
No <code>useState</code> , <code>onClick</code> , <code>window</code>	Can't be <code>async</code> ; can't import server-only code

Fetch on the server, interact on the client

```
// app/products/page.tsx - Server Component
import { AddToCart } from './add-to-cart'

export default async function Page() {
  const products = await db.product.findMany() // runs on the server
  return products.map((p) => (
    <div key={p.id}>{p.name} <AddToCart id={p.id} /></div>
  ))
}

// app/products/add-to-cart.tsx
'use client'
export function AddToCart({ id }: { id: string }) {
  const [added, setAdded] = useState(false)
  return <button onClick={() => setAdded(true)}>{added ? 'Added' : 'Add'}</button>
}
}
```

Push `'use client'` down to the leaves (buttons, forms) so most of the page stays server-rendered with zero JS.

Rendering strategies

Strategy	HTML made	Good for
CSR	in the browser	dashboards behind login
SSR (dynamic)	on every request	personalised or always-fresh pages
SSG (static)	at build time	marketing pages, docs, blogs
ISR	at build, then regenerated after a time or on demand	product pages, large catalogs
Streaming	shell first, slow parts stream in via Suspense	pages with slow data

- Hydration = React attaching event handlers to server-rendered HTML.
- Hydration mismatch: server and client render different output (dates, `window`, random values).
- Partial prerendering: a static shell is served instantly, and dynamic holes stream in.

Data fetching, caching and mutations

Server Action for a form

```
// app/todos/actions.ts
'use server'
import { revalidatePath } from 'next/cache'

export async function addTodo(formData: FormData) {
  const title = String(formData.get('title') ?? '').trim()
  if (!title) return { error: 'Title is required' } // validate on the
server
  await db.todo.create({ data: { title } })
  revalidatePath('/todos') // refresh cached data
}

// app/todos/page.tsx
<form action={addTodo}><input name="title" /><button>Add</button></form>
```

- ✎ Server Actions are public endpoints — always validate input and check auth inside them.
- ✎ Cache invalidation: `revalidatePath`, `revalidateTag` (on-demand) or time-based revalidation.
- ✎ Newer versions make caching explicit (`'use cache'`, Cache Components); request APIs like `cookies()`, `headers()` and `params` are async.
- ✎ Route Handlers (`route.ts`) for webhooks and public APIs.

Feature	Why
<code>next/link</code>	client-side navigation + prefetching
<code>next/image</code>	resizing, lazy loading, modern formats, no layout shift
<code>next/font</code>	self-hosted fonts, no layout shift
Metadata API	<code>export const metadata</code> / <code>generateMetadata</code> for SEO tags
Middleware / proxy	run before a request: auth redirects, rewrites, i18n
<code>generateStaticParams</code>	pre-build dynamic routes

Common mistake

Reading `window` or `localStorage` during render in a Client Component — it also renders on the server. Read browser-only values in effects or event handlers.

Revision

App Router files = routes. Server Components by default, `'use client'` only for interactivity. SSG/ISR/SSR/streaming per page. Server Actions + revalidate for mutations; validate everything on the server.

Q. Server Components vs Client Components?

A. Server Components run only on the server (can be async, access the DB, ship no JS). Client Components ('use client') can use state, effects and event handlers and are hydrated in the browser.

Q. SSR vs SSG vs ISR?

A. SSR renders on every request; SSG renders at build time; ISR serves static pages and regenerates them after a time or on demand.

Q. What is hydration?

A. React attaching event handlers and state to server-rendered HTML so it becomes interactive.

Q. What causes a hydration mismatch?

A. Server and client rendering different output — dates, random values, window checks or browser-only data during render.

Q. What are Server Actions?

A. Async server functions ('use server') you can call from forms or client code to mutate data; they must validate input and check auth.

Q. What does `loading.tsx` do?

A. It becomes the Suspense fallback for the route segment while its content loads.

Q. Why use `next/image`?

A. Automatic resizing, lazy loading, modern formats and reserved dimensions that prevent layout shift.

Q. How do you invalidate cached data after a mutation?

A. `revalidatePath` or `revalidateTag` (on demand), or time-based revalidation.

Q. Where should 'use client' go?

A. As low in the tree as possible — on the interactive leaves — so most of the page stays server-rendered.