# Next.js Basic

Version 16 · verified 2026-08-11

App Router routing, layouts, navigation, server and client components, data fetching, error UI, metadata, images and route handlers.

## Contents


Mental model ..... 2

Routing by file system ..... 2

Pages and layouts ..... 4

Navigation ..... 5

Server and Client Components ..... 7

Fetching data ..... 9

Loading and error UI ..... 11

Metadata and images ..... 13

Route handlers ..... 16

Further reading ..... 17

## Mental model

Next.js is React plus a server, and the `app` directory **is** the router: folders are URL segments and a handful of reserved filenames — `page`, `layout`, `loading`, `error`, `route` — are the boundaries React needs. Everything is a Server Component until a file says `'use client'`, so the default output is HTML that ships no component JavaScript, and interactivity is opted into one file at a time. Learn the file conventions and most of the framework follows; the component model underneath is plain React.

## Routing by file system

| Path | URL |
| --- | --- |
| `app/page.tsx` | `/` |
| `app/blog/page.tsx` | `/blog` |
| `app/blog/[slug]/page.tsx` | `/blog/hello` |
| `app/shop/[...slug]/page.tsx` | `/shop/a/b` |
| `app/(marketing)/about/page.tsx` | `/about` |
| `app/blog/_lib/db.ts` | Not routable |

Folders define segments, but a segment “is **not** publicly accessible until a `page.js` or `route.js` file is added” — so everything else can be colocated beside the route that uses it. Square brackets make a segment dynamic: `[slug]` matches one, `[...slug]` catches all, `[[...slug]]` makes it optional. Parentheses create a route group that organizes files without appearing in the URL, and a leading underscore makes a private folder the router ignores entirely.

```
app/
  layout.tsx          wraps every route
  page.tsx            /
  blog/
    layout.tsx        wraps /blog and below
    page.tsx          /blog
    [slug]/page.tsx   /blog/hello
```

> **Gotcha:** A folder with only a `layout.tsx` is not a route — the URL 404s until a `page.tsx` sits beside it. Layouts wrap routes; they never create one.

## Pages and layouts

| Prop or file | What it gives you |
|---|---|
| `params` | Promise of the dynamic segments |
| `searchParams` | Promise of the query string |
| `layout.tsx` | UI kept mounted across child routes |
| `template.tsx` | Same, but remounted every navigation |

A page receives `params` and `searchParams`, and in Next.js 16 both are Promises — synchronous access was removed, so you `await` them. A layout wraps its segment and everything below it, keeps state and scroll position across navigations within that subtree, and never receives `searchParams`. The root layout is the one that must render `<html>` and `<body>`.

```tsx
// app/blog/[slug]/page.tsx
export default async function Page({
  params,
}: {
  params: Promise<{ slug: string }>;
}) {
  const { slug } = await params;
  return <h1>{slug}</h1>;
}
```

> **Gotcha:** `params.slug` without `await` is not a string, it is a property of a Promise. Code carried over from Next.js 14 compiles in JavaScript and silently renders `undefined`.

## Navigation

| API | Use it for |
| --- | --- |
| `<Link href>` | Client transition, with prefetch |

| API | Use it for |
| --- | --- |
| `useRouter()` | Programmatic navigation, client only |
| `usePathname()` | The current path, client only |
| `redirect()` | Server-side redirect during render |

`<Link>` prefetches routes as they enter the viewport or on hover, then swaps content client-side while shared layouts stay mounted and interactive. How much is prefetched depends on the target: a static route is prefetched whole, while “prefetching is skipped, or the route is partially prefetched if `loading.tsx` is present” for a dynamic one. Set `prefetch={false}` on long lists of links to stop spending bandwidth on routes nobody will open.

```
import Link from "next/link";


export default function Nav() {
  return (
```

```
    <nav>
      <Link href="/blog">Blog</Link>
      <Link href="/reports" prefetch={false}>
        Reports
      </Link>
    </nav>
  );
}
```

> **Gotcha:** A plain `<a href="/blog">` to an internal route forces a full document load — client state, scroll position and the mounted layout are all discarded. It is the single easiest way to make a fast app feel slow.

## Server and Client Components

| You need | Component type |
| --- | --- |
| Database, secrets, heavy libraries | Server |
| `useState`, `onClick`, effects | Client |

| You need | Component type |
| --- | --- |
| `localStorage`, `window` | Client |
| `metadata` / `generateMetadata` | Server only |

Layouts and pages are Server Components by default. Adding `'use client'` marks a boundary, and “all of its imports and the components it directly renders are included in the client bundle” — so the directive belongs on the interactive leaf, not on the layout above it. Server Components passed as `children` are exempt: they are not in the client module graph, they render on the server and arrive as finished output, which is how a server-rendered `<Cart>` lives inside a client `<Modal>`.

```
// Only this file's JavaScript reaches the
browser.
"use client";
import { useState } from "react";


export function Like({ start }: { start: number })
```

```
{
  const [n, setN] = useState(start);
  const bump = () => setN(n + 1);
  return <button onClick={bump}>{n}</button>;
}
```

> **Gotcha:** `'use client'` at the top of `app/layout.tsx` turns the whole application into a client bundle, silently undoing the reason to use the App Router. Push the directive down to the smallest component that needs it.

## Fetching data

| Where | How |
| --- | --- |
| Server Component | `await` a fetch, ORM or SDK |
| Client Component | `use(promise)`, SWR, React Query |

| Where | How |
| --- | --- |
| Same data, many components | `cache()` from React |
| Independent requests | `Promise.all` to parallelize |

Server Components fetch with plain `await`, and because they run on the server the credentials and query never reach the browser. Identical `fetch` calls in one render pass are memoized, so fetch where you need the data instead of drilling props. Two awaits written in sequence *are* sequential — start both requests first, then await them together.

```
type Get = (id: string) => Promise<string[]>;
declare const getArtist: Get;
declare const getAlbums: Get;

export async function Page({ id }: { id: string })
{
```

```
  // Sequential: the second waits on the first.
  const first = await getArtist(id);
  // Parallel: both are already in flight.
  const [a, b] = await Promise.all([
    getArtist(id),
    getAlbums(id),
  ]);
  return <p>{a.length + b.length}</p>;
}
```

> **Gotcha:** `fetch` results "are not cached by default and will block the page from rendering until the request is complete". Code written for Next.js 13 that relied on automatic caching now hits the origin on every single request.

## Loading and error UI

| File | Wraps the segment in |
|---|---|
| `loading.tsx` | A Suspense boundary |

| File | Wraps the segment in |
| --- | --- |
| `error.tsx` | An error boundary, client only |
| `not-found.tsx` | The UI for `notFound()` |
| `global-error.tsx` | A replacement for the root layout |

`loading.tsx` is sugar: Next.js wraps the page in `<Suspense>` with it as the fallback, so navigation feels instant and the shared layout stays interactive while the page renders. `error.tsx` must be a Client Component, receives `error` and `retry`, and catches anything thrown below it — errors bubble to the nearest boundary, so place them per segment. Expected failures like form validation should be **return values**, not thrown errors.

```
// app/dashboard/error.tsx
"use client";

export default function ErrorPage({
  error,
  retry,
```

```
}: {
  error: Error & { digest?: string };
  retry: () => void;
}) {
  console.error(error);
  return <button onClick={retry}>Try again</
button>;
}
```

> **Gotcha:** Error boundaries only catch errors thrown *during render*. A rejected promise in an `onClick` handler escapes every `error.tsx` you have — catch it and put the message in state yourself.

## Metadata and images

| Export or prop | Effect |
|---|---|
| `export const metadata` | Static `<head>` tags |
| `generateMetadata()` | Dynamic tags, Server only |

| Export or prop | Effect |
| --- | --- |
| `title.template` | Applies to child segments only |
| `<Image>` `width` / `height` | Reserves space, prevents shift |

Metadata is merged from the root layout down, shallowly, with later segments replacing keys — define `title.template` in a layout and a plain `title` in each page. Because the tags must exist before the page renders, both exports are “only supported in Server Components”. `next/image` needs `alt` plus explicit dimensions (or `fill`), and Next.js 16 tightened its defaults: `quality` is coerced to the closest value in `images.qualities`, now `[75]`.

```
import Image from "next/image";

export const metadata = {
  title: { template: "%s | Acme", default:
```

```
"Acme" },
};
export default function Page() {
  return (
    <Image
      src="/hero.png"
      alt="The product, on a desk"
      width={800}
      height={400}
    />
  );
}
```

> **Gotcha:** `title.template` applies to children, never to the segment that declares it — which is why `title.default` is required alongside it. Without the default, the layout's own route renders no title at all.

## Route handlers

| Item | Rule |
|---|---|
| Exports | `GET`, `POST`, `PUT`, `PATCH`, `DELETE` … |
| `context.params` | A Promise; await it |
| `page.tsx` + `route.ts` | Cannot share one segment |
| `NEXT_PUBLIC_` prefix | The only env vars sent to the browser |

A route handler is a plain function from a Web `Request` to a `Response`, so `request.json()` and `request.formData()` are all the body parsing there is — "you do not need to use `bodyParser`". `GET` handlers are dynamic by default. Remember that only environment variables prefixed `NEXT_PUBLIC_` are inlined into the client bundle; everything else is replaced with an empty string there.

```
// app/api/search/route.ts
export async function GET(req: Request) {
```

```
  const { searchParams } = new URL(req.url);
  const q = searchParams.get("q") ?? "";
  return Response.json({ q });
}
```

> **Tip:** Do not add a route handler just so a Server Component can fetch it. That is an HTTP round trip into your own process — call the data function directly and keep handlers for genuinely external callers like webhooks and third-party clients.

## Further reading

- Project structure
- Layouts and pages
- Server and Client Components
- Fetching data
- Linking and navigating
- Error handling
- route.js reference