Next.js Advanced

Version 16 · verified 2026-08-11

Cache Components, use cache, revalidation, Server Actions, streaming, parallel and intercepting routes, and proxy — Next.js 16 in depth.

Contents

Mental model

Next.js 16 is **dynamic by default**: nothing is cached unless you say so. Turning on Cache Components makes that model explicit — `use cache` gives a result a lifetime, `<Suspense>` marks what may arrive late, and the build emits a static shell with dynamic holes, which is Partial Prerendering. What makes a component dynamic is reading the request: `cookies()`, `headers()`, `searchParams`, `params`. Everything below is a decision about *where* that read happens, on top of the React server primitives the framework packages.

The rendering model

What a component does	Where it lands
Pure computation, module reads	The static shell
`use cache` with a lifetime	The shell, revalidated
Uncached read inside `<Suspense>`	Streams at request time

What a component does	Where it lands
`cookies()` or `headers()` read	Streams at request time
`Math.random()`, `Date.now()`	Needs `connection()` or a cache

Enable the model with `cacheComponents: true` in `next.config.ts`. Prerendering then walks the tree and sorts every component into one of those buckets: the shell "can be served directly from a CDN, without going through to the upstream server", and each `<Suspense>` fallback holds a place for content that arrives later. Anything the build cannot resolve and you have not marked raises a build-time insight naming the route, so the failure mode is a message rather than a slow page.

```
import { Suspense, type ReactNode } from "react";


declare function Header(): ReactNode;
declare function Live(): ReactNode;
```

```
export default function Page() {
  return (
    <>
      <Header />
      <Suspense fallback={<p>Loading…</p>}>
        <Live />
      </Suspense>
    </>
  );
}
```

> **Gotcha:** Reading `cookies()` no longer makes the whole route dynamic the way it did before Cache Components — but an *unwrapped* read still blocks the shell for everyone. The fix is a `<Suspense>` boundary around that component, not a route-level opt-out.

use cache

Where you put it	What gets cached
Top of a file	Every exported function
Top of a component	Its rendered output
Top of an async function	Its return value

The cache key is built from the build id, a hash of the function's location, and its serializable arguments — and "when a cached function references variables from outer scopes, those variables are automatically captured and bound as arguments", so a closure silently widens the key. Cached functions must be `async`. They may accept `children` and Server Functions as pass-through props, provided the body never inspects them.

```
import { cacheLife, cacheTag } from "next/cache";


export async function getProducts(category) {
  "use cache";
  cacheLife("hours");
```

```
  cacheTag("products");
  const url = `/api/products?c=${category}`;
  const res = await fetch(url);
  return res.json();
}
```

> **Gotcha:** A cached scope cannot read `cookies()`, `headers()` or `searchParams` anywhere in its call stack — including inside a helper it calls. On a dynamically rendered route "this surfaces when the route runs, so it can pass `next build` and fail under `next start`".

cacheLife profiles

Profile	revalidate	expire
`seconds`	1s	60s
`minutes`	1m	1h
`hours`	1h	1d
`days`	1d	1w

Profile	revalidate	expire
`max`	30d	1y

Name a profile in every `use cache` scope; omitting one applies `default`, which is 5 minutes stale, 15 minutes to revalidate, and no expiry. The three numbers are different audiences: `stale` is how long the browser reuses its copy, `revalidate` when the server refreshes in the background, `expire` when serving stale stops being allowed. A short-lived cache — the `seconds` profile, `revalidate: 0`, or an expiry under five minutes — is deliberately excluded from the prerender and becomes a dynamic hole instead.

```
"use cache";
cacheLife({
  stale: 3600,      // browser reuse window
  revalidate: 7200, // background refresh
  expire: 86400,    // hard limit
});
```

> **Gotcha:** Nesting a short-lived cache inside one that never named a profile fails the build during prerendering. Set `cacheLife` explicitly at each level rather than inheriting whatever the surrounding scope happens to use.

Revalidation

API	Where it runs	Semantics
`revalidateTag(tag, life)`	Actions, handlers	Stale-while-revalidate
`updateTag(tag)`	Server Actions only	Read-your-own-writes
`refresh()`	Server Actions only	Uncached data only
`revalidatePath(path)`	Actions, handlers	Whole route, blunt

Tag what you cache with `cacheTag`, then invalidate by tag. In 16 `revalidateTag` takes a `cacheLife` profile as a

second argument — the single-argument form is deprecated — and serves stale content while refreshing behind the user's back, which is right for a catalogue and wrong for the form the user just submitted. For that case `updateTag` "immediately expires cached data … the user sees their change right away". Prefer tags over paths: `revalidatePath` invalidates everything on the route.

```
"use server";
import { updateTag } from "next/cache";
import { redirect } from "next/navigation";

export async function createPost(formData) {
  const post = await db.post.create({
    data: formData,
  });
  updateTag("posts");
  redirect(`/posts/${post.id}`);
}
```

> **Gotcha:** Revalidating a tag only touches entries that called `cacheTag` with it. A page whose data was never tagged keeps serving its old copy, and nothing warns you — the mutation appears to work everywhere except the one screen the user is looking at.

Server Actions in depth

Concern	What to do
Authorization	Re-check inside every action
Input	Validate; arguments are attacker-controlled
Expected errors	Return them, don't throw
After a write	`updateTag`, then `redirect`

A Server Function is a POST endpoint on the route where it is used, not a private call. That has a sharp consequence for `proxy.ts`: "a matcher change or a refactor that moves a Server Function to a different

route can silently remove Proxy coverage", so the framework's own advice is to verify auth inside each function. Model expected failures as return values so `useActionState` can render them, and keep throws for genuine bugs that should hit an error boundary.

```
type State = { message: string } | null;

export async function createPost(
  _prev: State,
  data: FormData,
): Promise<State> {
  "use server";
  const title = String(data.get("title") ?? "");
  if (!title) {
    return { message: "Title is required" };
  }
  // authorize, validate, then write
  return null;
}
```

> **Warning:** An auth check in `proxy.ts` is not an auth check for your actions. Anyone can POST to the action endpoint with any arguments, in any order, from outside your UI — the only reliable place to stop them is inside the function body.

Streaming and prefetching

Move	Effect
`loading.tsx` in the segment	The whole page streams
`<Suspense>` beside the read	Only that subtree streams
Await `params` deep, not in a layout	A bigger static shell
`<Link prefetch={true}>`	Per-URL cached data, prerendered

“The deeper your async work sits in the tree, the more of the page can be prerendered.” A layout that awaits

`params`, `cookies()` or an uncached fetch in its body cannot be prerendered at all, and blocks navigation until it resolves — pass the promise down and await it inside a boundary instead, so the sidebar and `children` stay in the shell. Bots are detected by user agent and served a fully rendered page rather than a stream, so anything your shell needs at build time must also be reachable at request time.

```
import { Suspense, type ReactNode } from "react";
declare function Title(): ReactNode;
type P = { children: ReactNode };

export default function Layout(p: P) {
  return (
    <div>
      <nav>Sidebar stays in the shell</nav>
      <Suspense fallback={<h1>…</h1>}>
        <Title />
      </Suspense>
```

```
      {p.children}
    </div>
  );
}
```

> **Gotcha:** `loading.tsx` does not cover its own layout. A layout reading uncached data "does not fall back to a same route segment `loading.js`" — it blocks the navigation instead, which looks exactly like a slow server.

Parallel and intercepting routes

Convention	Meaning
`@slot`	Named slot, passed to the layout
`default.tsx`	Fallback on a hard navigation
`(.)folder`	Intercept the same level
`(..)folder`	Intercept the parent level
`(...)folder`	Intercept from the root

Slots are props on the parent layout, not URL segments: `@analytics` never appears in the path. On a client navigation Next.js keeps each slot's active subpage; on a refresh it cannot know them, so it renders `default.tsx` — and since 16 "all parallel route slots require explicit `default.js` files; builds fail without them". Pairing a slot with an interception gives the modal that has a shareable URL and closes on back.

```
app/
  layout.tsx          receives {children, auth}
  @auth/
    default.tsx       null when inactive
    (.)login/page.tsx modal over the list
  login/page.tsx      the real /login page
```

> **Gotcha:** In a conditional slot, both branches still render on the server: “`@admin/page.js` executes its data fetches for every user, and its output is included in the response”. Choosing in the layout hides the UI, not the data — authorize inside each slot.

Proxy and production

Change in 16	What it means
`middleware.ts` → `proxy.ts`	Same logic, Node.js runtime
Turbopack is the default	`next build --webpack` opts out
`next lint` removed	Run ESLint or Biome yourself
Node.js 20.9+, TypeScript 5.1+	New minimum versions
Parallel slots need `default.js`	Otherwise the build fails

`proxy.ts` replaces `middleware.ts`, runs before routes render, and is fixed to the Node.js runtime — setting `runtime` there throws. Keep it to redirects, rewrites and headers: the docs are blunt that it "is recommended to be used as a last resort", and that it should not rely on shared modules or globals. The codemod `npx @next/codemod@canary middleware-to-proxy .` renames both the file and the exported function.

```
import { NextResponse } from "next/server";

export function proxy(request) {
  const url = new URL("/login", request.url);
  return NextResponse.redirect(url);
}

export const config = {
  matcher: ["/dashboard/:path*"],
};
```

> **Warning:** Without a `matcher`, proxy "runs on **every request**, including static files (`_next/static`), image optimizations ... and assets in the `public/` folder". An auth redirect written without one will block your own CSS and JavaScript from loading.

Further reading

- Next.js 16 release notes
- Caching and Cache Components
- use cache directive
- Revalidating
- cacheLife
- Parallel routes
- proxy.js reference