React Advanced

Version 19.2 · verified 2026-08-11

Server Components, the client boundary, hydration, Suspense boundaries, use() and streaming — how one React 19 tree spans two runtimes.

Contents

Mental model

A React 19 tree can span two runtimes. Server Components render once, away from the browser, and send their **output** rather than their code; Client Components ship as JavaScript, are rendered to HTML on the server for the first paint, then hydrate. Suspense marks the places where output may arrive late, which is what makes streaming possible. Everything here is about that split — mutations and client-side transitions belong to whichever framework wires it up.

Server Components

Server Component	Client Component
Renders once, ahead of the browser	Runs and re-renders in the browser
Can `await` data in the body	Has state, effects, event handlers
Ships no JavaScript	Ships its bundle and hydrates

Server Component	Client Component
No state, effects, or DOM APIs	No direct database or filesystem

A Server Component “renders ahead of time, before bundling, in an environment separate from your client app or SSR server” — at build time or per request — and its result, not its source, reaches the browser. Because they “do not re-render or hydrate”, they have no state, no effects and no browser APIs, and in exchange they may be `async` and await a query directly in the body. They need no directive: in a Server Component tree, server is the default.

```
declare const db: {
  byUser(id: string): Promise<{ id: string }[]>;
};

async function Notes({ id }: { id: string }) {
  const notes = await db.byUser(id);
```

```
  return (
    <ul>
      {notes.map(n => <li key={n.id}>{n.id}</li>)}
    </ul>
  );
}
```

> **Gotcha:** Server rendering and Server Components are different things. A Client Component is still rendered to HTML on the server for the first paint — then it hydrates and ships its JavaScript. Only a Server Component sends nothing to the browser at all.

The client boundary

Crosses the boundary	Does not
Primitives, plain objects, `Date`	Class instances
Arrays, `Map`, `Set`	Ordinary functions

Crosses the boundary	Does not
JSX elements and Promises	Unregistered symbols
Server Functions	Null-prototype objects

`'use client'` at the very top of a file "marks the boundary between server and client code in the module dependency tree": that module and everything it imports become client code, so the directive is declared once at the seam, never in every file. Props crossing the seam are serialized, which is the whole constraint — and since JSX elements and Promises both serialize, a Server Component can render its own output *into* a Client Component through `children` and keep it on the server.

```
"use client";
import { useState, type ReactNode } from "react";

type Props = { children: ReactNode };
export function Expander({ children }: Props) {
  const [open, setOpen] = useState(false);
```

```
  return (
    <div>
      <button onClick={() => setOpen(!open)}>
        More
      </button>
      {open && children}
    </div>
  );
}
```

> **Gotcha:** An inline callback prop cannot cross the boundary — ordinary functions do not serialize, so `onSelect={() => …}` from a Server Component throws. Move the handler into the client module, or pass a Server Function instead.

Server rendering and hydration

Stage	What the browser gets
Server render	HTML: a fast, non-interactive paint

Stage	What the browser gets
RSC payload	Server output plus client placeholders
Hydration	Handlers attached to the existing DOM
Mismatch	React re-renders on the client instead

`hydrateRoot` “attaches React to existing HTML that was already rendered by React on the server” — it adopts the DOM rather than building it, which is what makes the first paint cheap. The contract is exact: “the React tree you pass to `hydrateRoot` needs to produce the same output as it did on the server.” Differ, and you pay twice — once for HTML the client throws away, once to render it again.

```
import { hydrateRoot } from "react-dom/client";
declare const App: () => null;
```

```
const el = document.getElementById("root")!;
hydrateRoot(el, <App />);
```

> **Gotcha:** The usual culprits are all “correct” code: `Date.now()`, `Math.random()`, `typeof window !== "undefined"` branches, and locale-dependent formatting — the server and the browser simply disagree. For one genuinely dynamic node, `suppressHydrationWarning` is the escape hatch; it works one level deep and patches nothing.

Suspense

Suspends	Does not suspend
`use()` on a pending promise	`fetch` inside an Effect
`lazy()` components	Data set in an event handler

Suspends	Does not suspend
Data streamed from the server	Anything awaited outside render

`<Suspense>` shows its `fallback` until *all* code and data its children need have loaded. Children under one boundary are revealed together, so nesting boundaries is how a page is staged rather than gated: a coarse boundary around the shell, finer ones around the slow parts. During a transition React will not replace content that is already on screen with a fallback, which is the difference between a navigation and a flash of skeletons.

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

declare function Profile(): ReactNode;
declare function Posts(): ReactNode;

export function Page() {
  return (
```

```
    <Suspense fallback={<p>Loading page…</p>}>
      <Profile />
      <Suspense fallback={<p>Loading posts…</p>}>
        <Posts />
      </Suspense>
    </Suspense>
  );
}
```

> **Gotcha:** Suspense only reacts to reads that happen *during render*. A `fetch` in `useEffect` or an event handler never triggers a fallback no matter how many boundaries wrap it — that data needs its own loading state, or a move to `use()`.

use()

`use()`	Hooks
Legal inside `if` and loops	Top level only

`use()`	Hooks
Reads a Promise or a Context	Context only, via `useContext`
Suspends until the value is ready	Never suspend

`use` reads a promise or a context, and is the one React API exempt from the rules of hooks: it may be called conditionally. Reading a pending promise suspends the component, so it wants a Suspense boundary above it and an error boundary for rejection — the docs are explicit that `try` / `catch` around `use` does not work. The intended flow is a Server Component starting the request and passing the promise down, so the server begins work the client finishes awaiting.

```
"use client";

import { use } from "react";
```

```
export function Message(
  { promise }: { promise: Promise<string> },
) {
  return <p>{use(promise)}</p>;
}
```

> **Gotcha:** `use(fetch(url))` in a Client Component creates a new promise on every render, so the component suspends forever and re-fetches endlessly. Client-side promises must come from a cache keyed by input, or from a Server Component's props.

Streaming SSR

Piece	Role
The shell	Everything outside every boundary
A boundary	Streams its HTML once its data resolves
Inline script	Swaps each fallback for real content
`allReady`	Wait for the whole page, for crawlers

Streaming “allows the user to start seeing the content even before all the data has loaded on the server”: the shell is flushed first, then each boundary’s HTML arrives out of order as its data resolves, carrying a script that replaces the fallback. The reveal “does not need to wait for React itself to load in the browser”, and selective hydration then makes the arrived parts interactive without waiting for the rest. React 19.2 batches those reveals briefly so content lands in groups rather than one row at a time.

```
import { renderToReadableStream }
  from "react-dom/server";
declare const App: () => null;

const stream = await renderToReadableStream(
  <App />,
  { bootstrapScripts: ["/main.js"] },
);
```

```
// Crawlers need the finished document:
// if (isCrawler) await stream.allReady;
```

> **Tip:** The shell is everything *outside* every boundary, so it is also everything the user waits for. Push slow data deeper — one boundary near the leaf that needs it — and the page paints sooner without changing a single fetch.

Further reading

- Server Components
- ‘use client’
- hydrateRoot
- Suspense
- use
- renderToReadableStream
- React 19.2 release notes