Suspense

Suspense's fallback prop, nested Suspense boundaries, integrating Suspense with React 19's use() hook, and what Suspense doesn't do automatically -- with simple examples.

Advanced 11 min

Last updated: August 18, 2026
Read online: https://www.learnforgex.com/en/topics/suspense

Suspense

In Lazy Loading, we only saw `Suspense` paired with `lazy()`. This lesson takes a closer look at `Suspense` itself -- what it does, how it can be nested, and what it does NOT do automatically.

The fallback Prop

`Suspense` shows a `fallback` while something INSIDE it isn't ready yet:

```
import { lazy, Suspense } from "react";

const CourseDetails = lazy(() => import("./CourseDetails.jsx"));

function SuspenseFallbackExample() {
  return (
    // `fallback` can be any JSX -- not just text, but also a spinner, a
    // skeleton screen, or another component. In the Lazy Loading lesson
    // we only saw Suspense used together with lazy() -- this lesson
    // takes a closer look at Suspense itself.
    <Suspense fallback={<p className="spinner">Loading course...</p>}>
      <CourseDetails />
    </Suspense>
  );
}
```

`fallback` can be ANY JSX, not just text -- a spinner, a skeleton screen, or another component. Once the component inside (here, `CourseDetails`) is ready, `fallback` is automatically REPLACED with the real content.

Nested Suspense Boundaries

Multiple `Suspense` boundaries can be nested at different levels:

```
import { lazy, Suspense } from "react";

const CourseHeader = lazy(() => import("./CourseHeader.jsx"));
const CourseReviews = lazy(() => import("./CourseReviews.jsx"));

function NestedSuspenseExample() {
  return (
    // The outer Suspense shows a fallback for the WHOLE page until
    // CourseHeader loads. Once CourseHeader appears, the INNER Suspense
    // only takes the place of CourseReviews -- the rest of the page
    // (including CourseHeader) does NOT go back to a "loading" state.
    <Suspense fallback={<p>Loading page...</p>}>
      <CourseHeader />
      <Suspense fallback={<p>Loading reviews...</p>}>
        <CourseReviews />
      </Suspense>
    </Suspense>
  );
}
```

The outer `Suspense` shows a fallback for the WHOLE page until `CourseHeader` loads. Once `CourseHeader` appears, the INNER `Suspense` only covers `CourseReviews` -- the rest of the page does NOT go back to a "loading" state. This gives users a smoother experience: instead of everything disappearing and reappearing at once, only the part that's still waiting shows "loading."

Suspense with the use() Hook

The `use()` hook in React 19 can integrate a Promise DIRECTLY with Suspense:

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

function fetchCourse() {
  return fetch("http://localhost:3000/courses/1").then((response) => response.json());
}

// To avoid creating a NEW Promise on every render, we call this once
// OUTSIDE the component, while the module loads.
const coursePromise = fetchCourse();

function CourseName() {
  // Unlike normal hooks, use() can also be called CONDITIONALLY. When
  // given a Promise, if the Promise has NOT resolved yet, use() tells
  // React "I need to wait" -- this shows the fallback of the nearest
  // Suspense; once the Promise resolves, it returns the actual value.
  const course = use(coursePromise);
  return <p>Course: {course.name}</p>;
}

function UsePromiseWithSuspenseExample() {
  return (
    <Suspense fallback={<p>Loading course...</p>}>
      <CourseName />
    </Suspense>
  );
}
```

Unlike other hooks, `use()` can also be called CONDITIONALLY. Given a Promise, if it hasn't RESOLVED yet, it tells React "I need to wait" -- this shows the nearest `Suspense`'s `fallback`; once the Promise resolves, `use()` returns the actual value and the component renders normally.

What Suspense Doesn't Do Automatically

An important gotcha: not every asynchronous operation triggers Suspense automatically:

```
import { Suspense, useEffect, useState } from "react";

function CourseListWithEffect() {
  const [courses, setCourses] = useState(null);

  // IMPORTANT: useEffect + fetch (the pattern from the API & Data
  // Fetching lesson) does NOT AUTOMATICALLY TRIGGER Suspense -- Suspense
  // only works with a Promise source that React DIRECTLY recognizes,
  // like use(). That's why, while the `courses` state is null here, the
  // outer Suspense's fallback does NOT show -- the component renders
  // `null`, and Suspense never even finds out.
  useEffect(() => {
    fetch("http://localhost:3000/courses")
      .then((response) => response.json())
      .then((data) => setCourses(data));
  }, []);

  if (!courses) {
    // Not Suspense's fallback -- our own manual loading check.
    return null;
  }

  return (
    <ul>
      {courses.map((course) => (
        <li key={course.id}>{course.name}</li>
      ))}
    </ul>
  );
}

function SuspenseLimitationsExample() {
  return (
    <Suspense fallback={<p>This fallback never shows for CourseListWithEffect.</p>}>
      <CourseListWithEffect />
    </Suspense>
  );
}
```

The `useEffect` + `fetch` pattern from API & Data Fetching does NOT automatically trigger Suspense -- Suspense only works with a Promise source that React DIRECTLY recognizes, like `use()`. A component using `fetch` inside `useEffect` still needs to manage its own `loading` state ITSELF.

Summary and Glossary

`Suspense` shows a `fallback` while something inside it isn't ready yet; once the content is ready, it's automatically replaced. Multiple `Suspense` boundaries can be nested to show loading states at different granularities. The `use()` hook in React 19 integrates a Promise with Suspense. But "classic" data-fetching patterns like `useEffect` + `fetch` do NOT automatically trigger Suspense -- only sources React directly supports, like `use()`, do.

Glossary

Suspense — A React component that shows a fallback UI while a resource inside it (a lazy component, a Promise) isn't ready yet.

Suspense Boundary — The area covered by a `<Suspense>` component, with its own fallback.