**🧠 What are Error Boundaries?**

Error boundaries are **special React components** (only available in **class components**) that **catch JavaScript errors** in their **child components** during rendering, lifecycle methods, and constructors — and **prevent the entire app from crashing**.

**🔥 Why we need them**

Without error boundaries, if **one component throws an error**, the **whole React app** can crash or go blank.

Error boundaries help you:

* Show a **fallback UI** (like “Something went wrong 😢”)
* Log errors for debugging
* Keep the rest of the app working

**⚙️ React’s Limitation**

React’s **built-in error boundary** feature **only works with class components**, because React’s internal lifecycle methods (getDerivedStateFromError and componentDidCatch) are **not available** in functional components.

So —  
❌ You can’t write a true error boundary directly using hooks.  
✅ But we can **use libraries** or **workarounds** to achieve the same behavior.

**✅ Option 1: Use a Library (Recommended)**

The most popular one is [react-error-boundary](https://www.npmjs.com/package/react-error-boundary) by Kent C. Dodds.  
It gives a **functional way** to use error boundaries.

**🧠 1. What exactly is an Error Boundary?**

An **Error Boundary** is like a **safety net** 🕸️ for your React app.

Imagine this:  
You have a big React app with 10 components.  
If one of those components crashes (throws an error while rendering),  
👉 normally the **entire app** would break — you’d just see a blank white screen ❌.

So React introduced **Error Boundaries** to catch those crashes **gracefully**, so your app doesn’t die.

**🧩 In simple terms:**

**Error Boundaries catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI instead of breaking the whole app.**

**🧱 2. Example without Error Boundary**

function BuggyComponent() {

throw new Error("Something went wrong!");

}

function App() {

return (

<div>

<h1>My App</h1>

<BuggyComponent />

</div>

);

}

👉 When this runs, the app **crashes completely**.  
Nothing will render — just a blank page and an error in console.

**✅ 3. Example with Error Boundary**

class ErrorBoundary extends React.Component {

constructor(props) {

super(props);

this.state = { hasError: false };

}

static getDerivedStateFromError(error) {

return { hasError: true };

}

componentDidCatch(error, info) {

console.log("Error caught:", error, info);

}

render() {

if (this.state.hasError) {

return <h2>Something went wrong 😢</h2>;

}

return this.props.children;

}

}

Now use it like this:

<ErrorBoundary>

<BuggyComponent />

</ErrorBoundary>

✅ Now if BuggyComponent crashes, only that part shows “Something went wrong” —  
the rest of the app keeps running smoothly

**⚙️ 4. When do you need Error Boundaries?**

You don’t need them everywhere — only where **crashes are possible** or **isolating errors matters**.

🧩 Common use cases:

| **Use Case** | **Example** |
| --- | --- |
| 1️⃣ Catch component render errors | Large dashboards, 3rd-party widgets |
| 2️⃣ Prevent the whole app from crashing | Wrap main routes or layout |
| 3️⃣ Fallback UIs | “Something went wrong” pages |
| 4️⃣ External integrations | Embedding iframes, charts, SDKs |
| 5️⃣ Lazy loaded or dynamic components | Suspense, async imports |

**🧰 5. Real-life example (practical view)**

Imagine your app has:

* Navbar
* Sidebar
* Dashboard with charts (using Chart.js)

If Chart.js suddenly fails to render due to a data issue,  
you don’t want the **whole app to go white**.  
You just want to show a message like “Chart temporarily unavailable”.

So you’d wrap **only that Chart component** in an Error Boundary:

<ErrorBoundary>

<ChartComponent data={chartData} />

</ErrorBoundary>

Everything else (Navbar, Sidebar) continues working fine ✅

**⚡ 6. Summary**

| **Concept** | **Explanation** |
| --- | --- |
| **Purpose** | Catch rendering errors so UI doesn’t break completely |
| **Works for** | Render, lifecycle, and constructor errors in child components |
| **Doesn’t catch** | Errors in event handlers, async code, or backend calls |
| **Usage** | Wrap risky parts of UI or the entire app |
| **Functional way** | Use react-error-boundary |