Error Boundaries

Class components that catch errors during rendering in React: getDerivedStateFromError, componentDidCatch, the scope of error boundaries, and what they don't catch, with simple examples.

Advanced 13 min

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

---

Error Boundaries

In this lesson, we'll see our FIRST **class component** in the course. There's no way to write error boundaries with hooks in React -- they can only be written using class components. Everything you've learned so far (hooks, state, props) applied to function components; this is an exceptional, narrowly-scoped topic.

Writing a Basic Error Boundary

When a component throws an error during rendering, React normally unmounts the ENTIRE application (showing a blank screen). An error boundary prevents this:

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

// IMPORTANT: Error boundaries are the FIRST class component we've seen in
// this course. There is NO way to write error boundaries with hooks
// (function components) in React -- only class components can do this,
// via `static getDerivedStateFromError`. That's why, for this ONE topic,
// we use a class component as an exception.
class BasicErrorBoundaryExample extends Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  // WHEN a child component THROWS an error during render, React calls
  // this method -- the value it returns becomes the new state.
  static getDerivedStateFromError() {
    return { hasError: true };
  }

  render() {
    if (this.state.hasError) {
      return <p>Something went wrong.</p>;
    }

    return this.props.children;
  }
}
```

`static getDerivedStateFromError()` is called by React when a child throws an error -- whatever it returns becomes the new state. The `render()` method shows either the normal `children` or a fallback message, based on the `hasError` state.

Logging the Error with componentDidCatch

`getDerivedStateFromError` is ONLY for showing the fallback UI -- sending the error somewhere (logging it) requires a separate method:

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

class ComponentDidCatchExample extends Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError() {
    return { hasError: true };
  }

  // getDerivedStateFromError is used to SHOW the fallback UI;
  // componentDidCatch is used to SEND the error somewhere (e.g. to a
  // logging service) -- the two can work together.
  componentDidCatch(error, errorInfo) {
    console.error("Caught an error:", error, errorInfo.componentStack);
  }

  render() {
    if (this.state.hasError) {
      return <p>Something went wrong.</p>;
    }

    return this.props.children;
  }
}
```

`componentDidCatch(error, errorInfo)` receives the error itself AND `errorInfo.componentStack` (a "stack trace" showing which component the error came from) -- in real applications, this is usually where a request is sent to an error-tracking service (like Sentry).

Using an Error Boundary

We use an error boundary to wrap components that might throw:

```
import { Component, useState } from "react";

class ErrorBoundary extends Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError() {
    return { hasError: true };
  }

  render() {
    if (this.state.hasError) {
      return <p>Something went wrong.</p>;
    }

    return this.props.children;
  }
}

function BuggyCounter({ count }) {
  if (count === 3) {
    // We deliberately throw an error during render -- in a real app this
    // would be an unexpected error, like `undefined.someProperty`.
    throw new Error("Count reached 3!");
  }

  return <p>Count: {count}</p>;
}

function UsingErrorBoundaryExample() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <button onClick={() => setCount(count + 1)}>Increment</button>
      {/* If any component INSIDE it throws an error during render, the
          ErrorBoundary CATCHES that error and REPLACES the normal render
          with the fallback UI -- BuggyCounter itself doesn't have to
          handle the error. */}
      <ErrorBoundary>
        <BuggyCounter count={count} />
      </ErrorBoundary>
    </div>
  );
}
```

`BuggyCounter` deliberately throws an error when `count === 3` -- `ErrorBoundary` catches it and replaces the normal render with a fallback UI. `BuggyCounter` itself doesn't need to handle the error -- that's the error boundary's job.

The Scope of Error Boundaries

Using multiple, SMALL error boundaries is usually better than one big one:

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

class ErrorBoundary extends Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError() {
    return { hasError: true };
  }

  render() {
    if (this.state.hasError) {
      return <p>{this.props.fallbackText}</p>;
    }

    return this.props.children;
  }
}

function BuggyWidget() {
  throw new Error("This widget is broken!");
}

function ErrorBoundaryScopeExample() {
  return (
    <div>
      {/* Two SEPARATE ErrorBoundaries wrap two SEPARATE sections -- even if
          the Sidebar crashes, the MainContent is NOT AFFECTED, the crash
          stays CONTAINED within its own ErrorBoundary. If we used a single
          large ErrorBoundary, any error could turn the ENTIRE page into a
          "Something went wrong" message. */}
      <ErrorBoundary fallbackText="Sidebar failed to load.">
        <BuggyWidget />
      </ErrorBoundary>
      <ErrorBoundary fallbackText="Main content failed to load.">
        <p>Main content (still works fine)</p>
      </ErrorBoundary>
    </div>
  );
}
```

Two separate `ErrorBoundary`s wrap two separate sections -- if one crashes, the other is NOT affected. If we used a single large boundary, any error could turn the ENTIRE page into a fallback message.

What Error Boundaries Don't Catch

Error boundaries have a limit -- they don't catch every kind of error:

```
import { Component, useState } from "react";

class ErrorBoundary extends Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError() {
    return { hasError: true };
  }

  render() {
    if (this.state.hasError) {
      return <p>Something went wrong.</p>;
    }

    return this.props.children;
  }
}

function EventHandlerExample() {
  function handleClick() {
    // Error boundaries only catch errors during RENDER -- an error thrown
    // INSIDE an event handler (like an onClick) is NOT CAUGHT by an error
    // boundary. A regular try/catch is needed for that.
    try {
      throw new Error("Button click failed!");
    } catch (error) {
      console.error("Caught manually:", error.message);
    }
  }

  return <button onClick={handleClick}>Click me</button>;
}

function WhatErrorBoundariesDontCatchExample() {
  const [showInfo, setShowInfo] = useState(false);

  return (
    <div>
      {/* The ErrorBoundary here wraps EventHandlerExample -- but the onClick
          error inside it will still NOT BE CAUGHT, because it happens in
          an event handler, not during render. */}
      <ErrorBoundary>
        <EventHandlerExample />
      </ErrorBoundary>
      <button onClick={() => setShowInfo(true)}>Show limitations</button>
      {showInfo && (
        <p>
          Error boundaries do NOT catch: event handler errors, errors in
```

```
          asynchronous code (setTimeout, fetch callbacks), errors during
          server-side rendering, or errors thrown in the boundary itself.
        </p>
      )}
    </div>
  );
}
```

Error boundaries only catch errors thrown during RENDERING -- they do NOT catch errors in event handlers (like `onClick`), asynchronous code (`setTimeout`, fetch callbacks), server-side rendering, or errors thrown in the boundary itself. Regular `try`/`catch` is used for errors in event handlers.

Summary and Glossary

An error boundary is a class component that defines `static getDerivedStateFromError` (to show a fallback UI) and optionally `componentDidCatch` (to log the error) -- React has no hook equivalent for this. An error boundary catches errors thrown during rendering by any component INSIDE it and replaces the normal render with a fallback UI. Using multiple small boundaries prevents an error in one section from affecting others. Error boundaries do NOT catch errors in event handlers, asynchronous code, or the boundary itself.

Glossary

Error Boundary — A class component that catches errors thrown during rendering by the components inside it and shows a fallback UI.

Fallback UI — Alternative UI shown in place of normal content during an error (or a loading state).