Error Boundaries

In the past, JavaScript errors inside components used to corrupt React's internal state and cause it to emit cryptic errors on next renders. These errors were always caused by an earlier error in the application code, but React did not provide a way to handle them gracefully in components, and could not recover from them.

Introducing Error Boundaries

A JavaScript error in a part of the UI shouldn't break the whole app. To solve this problem for React users, React 16 introduces a new concept of an "error boundary".

Error boundaries are React components that **catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI** instead of the component tree that crashed. Error boundaries catch errors during rendering, in lifecycle methods, and in constructors of the whole tree below them.

Note
Error boundaries do **not** catch errors for:

- Event handlers (learn more)
- Asynchronous code (e.g. `setTimeout` or `requestAnimationFrame` callbacks)
- Server side rendering
- Errors thrown in the error boundary itself (rather than its children)

A class component becomes an error boundary if it defines either (or both) of the lifecycle methods `static getDerivedStateFromError()` or `componentDidCatch()`. Use `static getDerivedStateFromError()` to render a fallback UI after an error has been thrown. Use `componentDidCatch()` to log error information.

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

  static getDerivedStateFromError(error) {
    // Update state so the next render will show the fallback UI.
    return { hasError: true };
  }

  componentDidCatch(error, errorInfo) {
    // You can also log the error to an error reporting service
    logErrorToMyService(error, errorInfo);
  }

  render() {
    if (this.state.hasError) {
      // You can render any custom fallback UI
      return <h1>Something went wrong.</h1>;
    }

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

Then you can use it as a regular component:

```
<ErrorBoundary>
  <MyWidget />
</ErrorBoundary>
```

Error boundaries work like a JavaScript `catch {}` block, but for components. Only class components can be error boundaries. In practice, most of the time you'll want to declare an error boundary component once and use it throughout your application.

Note that **error boundaries only catch errors in the components below them in the tree**. An error boundary can't catch an error within itself. If an error boundary fails trying to render the error message, the error will propagate to the closest error boundary above it. This, too, is similar to how catch {} block works in JavaScript.

Live Demo

Check out this example of declaring and using an error boundary with React 16.

Where to Place Error Boundaries

The granularity of error boundaries is up to you. You may wrap top-level route components to display a "Something went wrong" message to the user, just like server-side frameworks often handle crashes. You may also wrap individual widgets in an error boundary to protect them from crashing the rest of the application.

New Behavior for Uncaught Errors

This change has an important implication. **As of React 16, errors that were not caught by any error boundary will result in unmounting of the whole React component tree.**

We debated this decision, but in our experience it is worse to leave corrupted UI in place than to completely remove it. For example, in a product like Messenger leaving the broken UI visible could lead to somebody sending a message to the wrong person. Similarly, it is worse for a payments app to display a wrong amount than to render nothing.

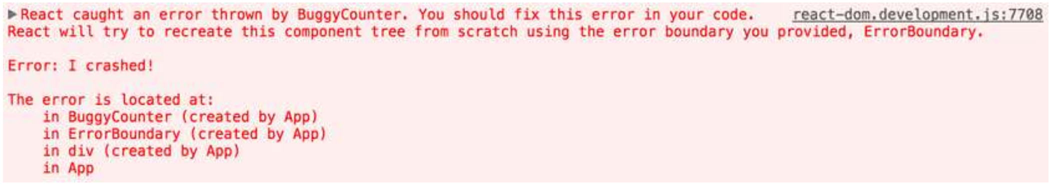
This change means that as you migrate to React 16, you will likely uncover existing crashes in your application that have been unnoticed before. Adding error boundaries lets you provide better user experience when something goes wrong.

For example, Facebook Messenger wraps content of the sidebar, the info panel, the conversation log, and the message input into separate error boundaries. If some component in one of these UI areas crashes, the rest of them remain interactive.

We also encourage you to use JS error reporting services (or build your own) so that you can learn about unhandled exceptions as they happen in production, and fix them.

Component Stack Traces

React 16 prints all errors that occurred during rendering to the console in development, even if the application accidentally swallows them. In addition to the error message and the JavaScript stack, it also provides component stack traces. Now you can see where exactly in the component tree the failure has happened:

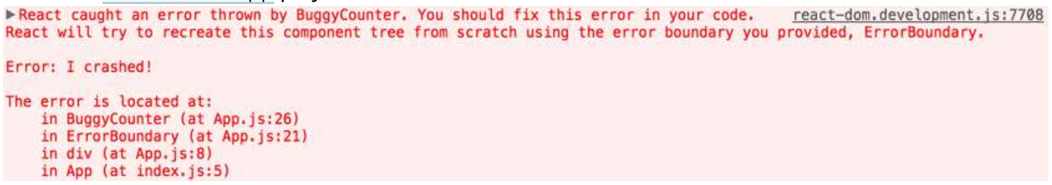


```
▶React caught an error thrown by BuggyCounter. You should fix this error in your code.    react-dom.development.js:7708
React will try to recreate this component tree from scratch using the error boundary you provided, ErrorBoundary.

Error: I crashed!

The error is located at:
    in BuggyCounter (created by App)
    in ErrorBoundary (created by App)
    in div (created by App)
    in App
```

You can also see the filenames and line numbers in the component stack trace. This works by default in Create React App projects:



```
▶React caught an error thrown by BuggyCounter. You should fix this error in your code.    react-dom.development.js:7708
React will try to recreate this component tree from scratch using the error boundary you provided, ErrorBoundary.

Error: I crashed!

The error is located at:
    in BuggyCounter (at App.js:26)
    in ErrorBoundary (at App.js:21)
    in div (at App.js:8)
    in App (at index.js:5)
```

If you don't use Create React App, you can add this plugin manually to your Babel configuration. Note that it's intended only for development and **must be disabled in production**.

> **Note**
> Component names displayed in the stack traces depend on the `Function.name` property. If you support older browsers and devices which may not yet provide this natively (e.g. IE 11), consider including a `Function.name` polyfill in your bundled application, such as `function.name-polyfill`. Alternatively, you may explicitly set the `displayName` property on all your components.

How About try/catch?

`try` / `catch` is great but it only works for imperative code:

```
try {
  showButton();
} catch (error) {
  // ...
}
```

However, React components are declarative and specify *what* should be rendered:

```
<Button />
```

Error boundaries preserve the declarative nature of React, and behave as you would expect. For example, even if an error occurs in a `componentDidUpdate` method caused by a `setState` somewhere deep in the tree, it will still correctly propagate to the closest error boundary.

How About Event Handlers?

Error boundaries **do not** catch errors inside event handlers.

React doesn't need error boundaries to recover from errors in event handlers. Unlike the render method and lifecycle methods, the event handlers don't happen during rendering. So if they throw, React still knows what to display on the screen.

If you need to catch an error inside event handler, use the regular JavaScript `try` / `catch` statement:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = { error: null };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    try {
      // Do something that could throw
    } catch (error) {
      this.setState({ error });
    }
  }

  render() {
    if (this.state.error) {
      return <h1>Caught an error.</h1>
    }
    return <button onClick={this.handleClick}>Click Me</button>
  }
}
```

Note that the above example is demonstrating regular JavaScript behavior and doesn't use error boundaries.

Naming Changes from React 15

React 15 included a very limited support for error boundaries under a different method name: `unstable_handleError`. This method no longer works, and you will need to change it to `componentDidCatch` in your code starting from the first 16 beta release.

For this change, we've provided a codemod to automatically migrate your code.

Edit this page