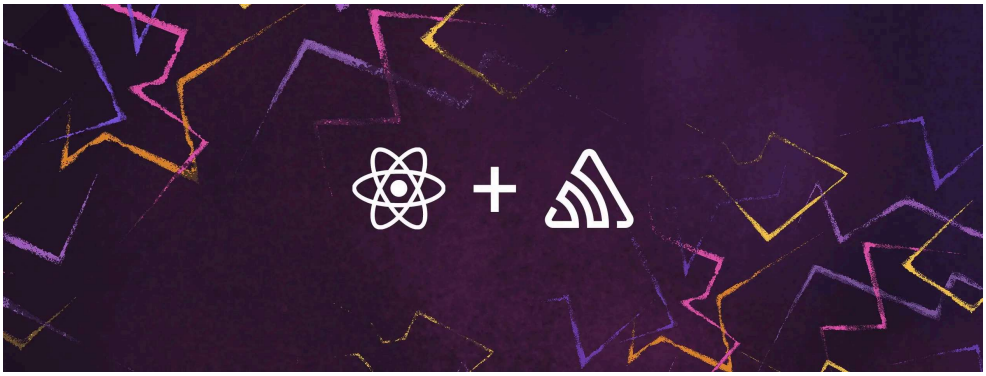
← BACK TO BLOG HOME

# Guide to Error & Exception Handling in React

Armin Ulrich - October 17, 2024



No app is perfect. Even if we try our best to test all possible cases, sometimes things will go wrong. It only takes one failing request to get you in trouble. It's how we handle it that makes the difference between an application crash and a graceful fail.

In this article, we'll cover the basics of error and exception handling in React apps. We'll also explore different kinds of errors and best practices for recovering from them in a user-friendly way.

## Why React error handling is important

Errors in React can be caused by many different issues. It could be network issues, an external API that's currently down, unexpected data inputs, or (as always) coding mistakes. Any of these could cause your app to crash, which we want to avoid at

all costs. Getting to a white page with no way forward impacts your user's trust: Your app will feel unreliable and flaky to them.

Critical tasks like payments and user submissions should especially be handled carefully. You don't want your users to sit in front of a crashed e-commerce app, asking themselves, "Did my order go through anyway?".

*We want our app to fail gracefully, meaning that our UI should still be responsive if an error occurs, and we'll do our best to recover quickly.*

React error handling helps us to:

- Deliver a good UX and reassure our users, even if things go wrong
- Document details about errors and exceptions so we can fix them

## 7 effective React error handling techniques

MENU >

The most important concept to understand for React error handling is React error boundaries, which are like little safety nets for your app.

**Error boundaries are React components that catch JS errors during rendering, in lifecycle methods, and in constructors in the whole tree below them.** You can then log those errors and display a fallback UI.

An error boundary component can define the lifecycle methods `static getDerivedStateFromError(error)` to update the state

and `render()` a fallback UI or/and `componentDidCatch(error, info)` to log error information.

Here's an example of an `ErrorBoundary` class component:

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

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

  componentDidCatch(error, info) {
    console.error('Error caught by ErrorBoundary:', error,

    // Log to an error monitoring service
    errorService.log({ error, info });
  }

  render() {
    const { hasError, error } = this.state;

    if (hasError) {
      // You can render any custom fallback UI
      return (
        Oh no, something went wrong 😕 - try again later!
      );
    }

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

**We can now wrap this component around our error-prone components to prevent an application crash.** You can wrap

small individual components or your entire app – the granularity is up to you.

```
<ErrorBoundary>
    <Component />
</ErrorBoundary>
```

## The limitations of error boundaries

There are some limitations of error boundaries that we should know about. Error boundaries do not catch errors for:

- Event handlers
- Async code
- Server-side rendering
- Errors thrown in the error boundary itself (rather than its children)

So it won't work to just wrap event handlers or asynchronous code in an error boundary component, expecting it to capture errors. For operations like a `fetch` call, you will need to use `try...catch` blocks as a catch-all way of handling errors.

```
async function fetchData() {
  try {
    const response = await fetch('/api/data');
    const data = await response.json();
    // Process data
  } catch (error) {
    // Handle the error
    console.error(error);
```

```
  }
}
```

## Error boundaries & Suspense

However, it does work to combine error boundaries with Suspense, which lets you pause rendering while waiting for asynchronous processes to finish (like fetching data).

If an error occurs during loading, the error boundary can render a fallback UI at that position. For our UX, that means we avoid a complete page refresh and keep our app responsive and stable.

## Using react-error-boundary

While you can write your own ErrorBoundary components for full control, in many cases, it will make sense to check out the `react-error-boundary` package.

It provides a pre-built component for a simple setup and additional features for resetting error boundaries, support for hooks, an easy integration with error logging services and an easy way to specify a fallback component.

As an example on how the package makes handling errors easier, we'll take a look at the `onReset` prop that we can use to reset a component when an error is thrown. This makes it super easy for us to build a retry mechanism for example:

```
function fallbackRender({ error, resetErrorBoundary }) {
  return (
    Oh no, something went wrong 😕: {error.message}
      <button onClick={resetErrorBoundary}>Try again</butt
```

```
  );
}

<ErrorBoundary
  fallbackRender={fallbackRender}
  onReset={() => {
    // Reset the state of your app
  }}
>
  <ExampleApplication />
</ErrorBoundary>;
```

When the `resetErrorBoundary` function is called by clicking the button, the error boundary will be reset, and the rendering will be repeated.

### Error reporting hooks

React 19 additionally provides `onCaughtError` and `onUncaughtError` hooks that are called when React catches an error in an Error Boundary (`onCaughtError`) or when an error is thrown and not caught by an Error Boundary (`unUncaughtError`).

We can add these hooks for centralized error processing (and reporting) at the root level and outside of the scope of Error Boundaries. Going forward, they will help us handle errors globally in a flexible, but also consistent way.

## React error handling best practices

Here are some best practices for React error handling at various levels of your app, so you can use error boundaries strategically and provide a smooth UX.

- **Add error boundaries to cricital components** where any errors shouldn't bubble up further, and components that

may frequently cause errors (because they rely on external data for example)

- **Add a top-level error boundary** around your root component to catch any unhandled errors from the tree below. / Use the error reporting hooks at the root level.
- **Provide error-specific messages and fallback UIs** for the best UX – for example you should handle errors in the shopping cart differently than when loading a news feed.
- **Handle errors within the UI section of the affected component** (sidebar, main feed,...) to isolate them and keep the rest of the UI responsive.
- **Implement a retry logic** for network requests and other data loading operations that might fail occasionally.
- **Log all errors** so you know about them and can fix them in a timely manner.
- **Test your error-handling workflows** with unit / integration tests.

## React error handling tools

Now we know the basics of how to implement error handling; we'll talk about tools that are important for your workflow.

### Console logging & Dev tools

For simple logging, while we are developing, we can use console logs:

```
console.log("Logging things while processes are running");
console.error("An error occured:" + myObject);
```

The console lets you view and inspect your logs, providing detailed error stack traces including line numbers & functions.

The React Developer Tools and Redux DevTools browser extensions further enhance your debugging experience.

Using these tools lets you see which components throw an error in the component tree, check component state (and state changes), monitor props and even track performance issues.

## Error monitoring

While logging is great during development, we will need to persist our logs somewhere to know about errors that happen to our users.

**Using an error monitoring service helps us track, log, and analyze production issues.** We can get notifications and detailed reports about errors, including info about which users are impacted, breadcrumbs, or even session replays. Having an error monitoring service in our workflow definitely brings us peace of mind, as we're running a SaaS product that a lot of people rely on daily.

Sentry, for example, exports a custom `ErrorBoundary` component that captures the original rendering error and attaches the component stack that the Error Boundary generates.

```
<Sentry.ErrorBoundary fallback={An error has occurred}>
  <Example />
</Sentry.ErrorBoundary>;
```

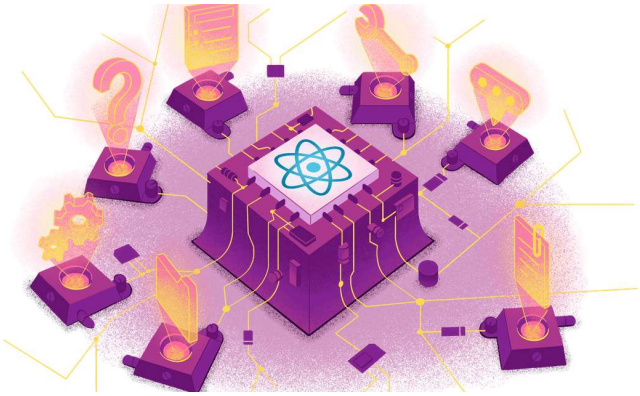
For apps using React 19, you can also use the `onCaughtError` / `onUncaughtError` hooks to capture errors automatically with Sentry.

```
import { createRoot } from "react-dom/client";

const container = document.getElementById("app");
const root = createRoot(container, {
    // Error is thrown and not caught by ErrorBoundary
    onUncaughtError: Sentry.reactErrorHandler((error, erro
        console.warn('Uncaught error', error, errorInfo.co
    }),
    // Error caught in ErrorBoundary
    onCaughtError: Sentry.reactErrorHandler(),
    onRecoverableError: Sentry.reactErrorHandler(),
});

root.render();
```

## Conclusion

Error handling is something that you have to customize for your app and its processes. While there is no one right way to do it, we covered the most important techniques, tools and workflows specifically for React in this article.

We hope you got a nice overview of how to build resilient apps #madewithreactjs!



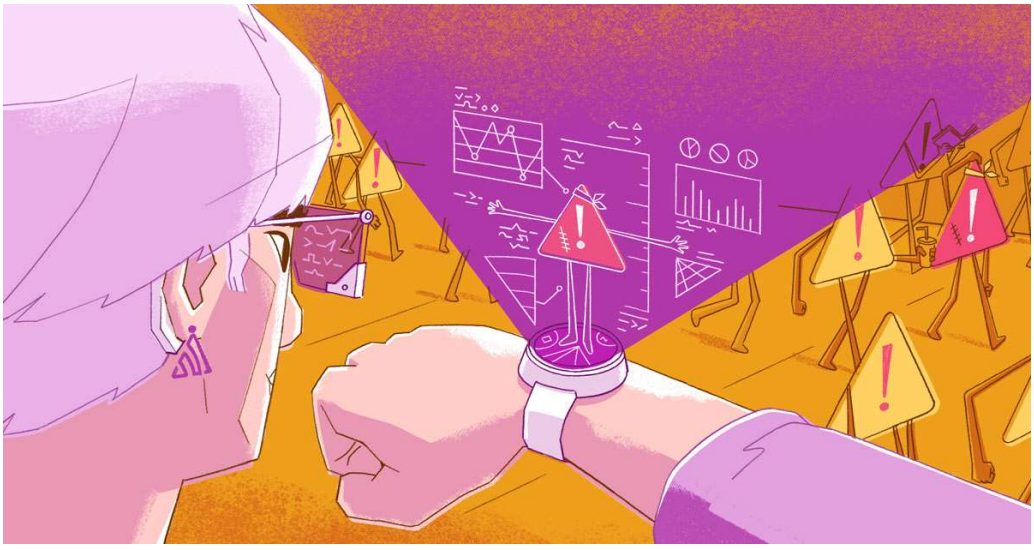
BLOG

## Improve Your React Debugging

In this guide, you'll learn how to identify and solve the most common bugs and performance issues in your React app.

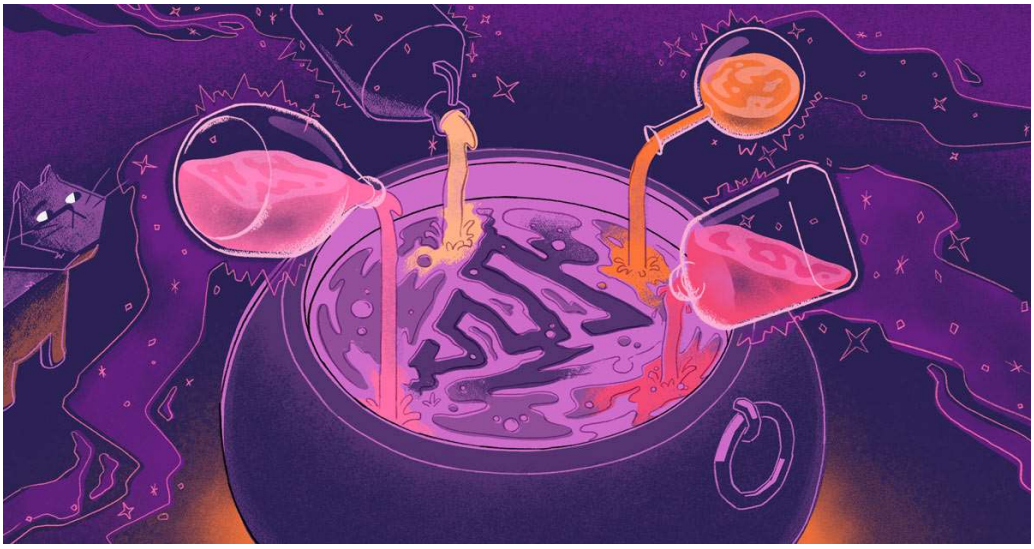
READ MORE

SHARE



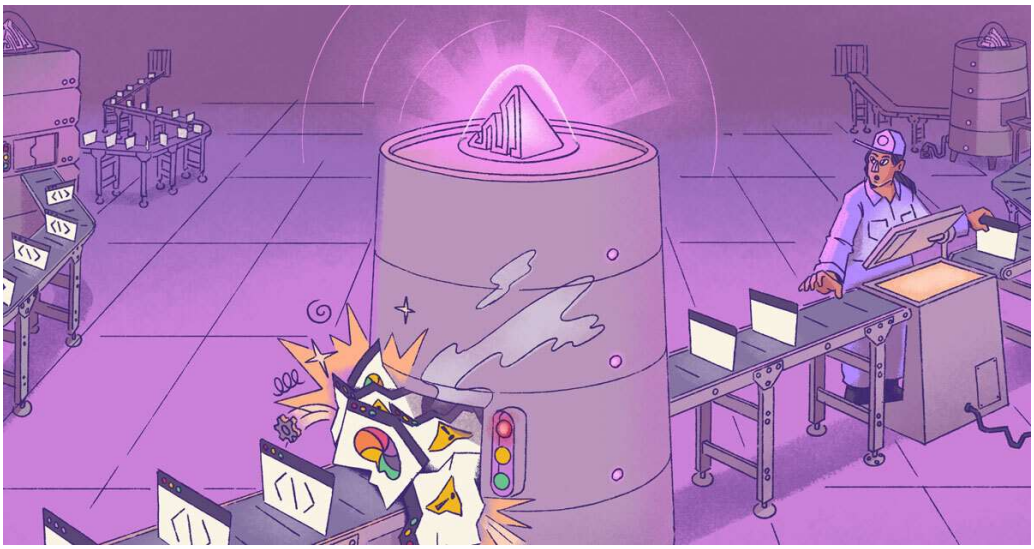
## Cancel the (Issue) Noise

READ MORE



## Debugging a Slack Integration with Sentry's Trace View

READ MORE



## How to deal with API rate limits

READ MORE

CUSTOMERS

COMMUNITY

OPEN SOURCE

CHANGELOG

MEDIA RESOURCES

ENGINEERING BLOG

LARAVEL

IOS

RAILS

NODE

REACT

GO

SEE ALL >

## Information

TRUST CENTER

SECURITY & COMPLIANCE

PRIVACY

CALIFORNIA PRIVACY NOTICE

TERMS

TRANSPARENCY REPORT

SUPPORT

RESOURCES

ANSWERS

TWITTER GITHUB DRIBBBLE LINKEDIN

DISCORD


OPEN
SOURCE
PLEDGE
MEMBER