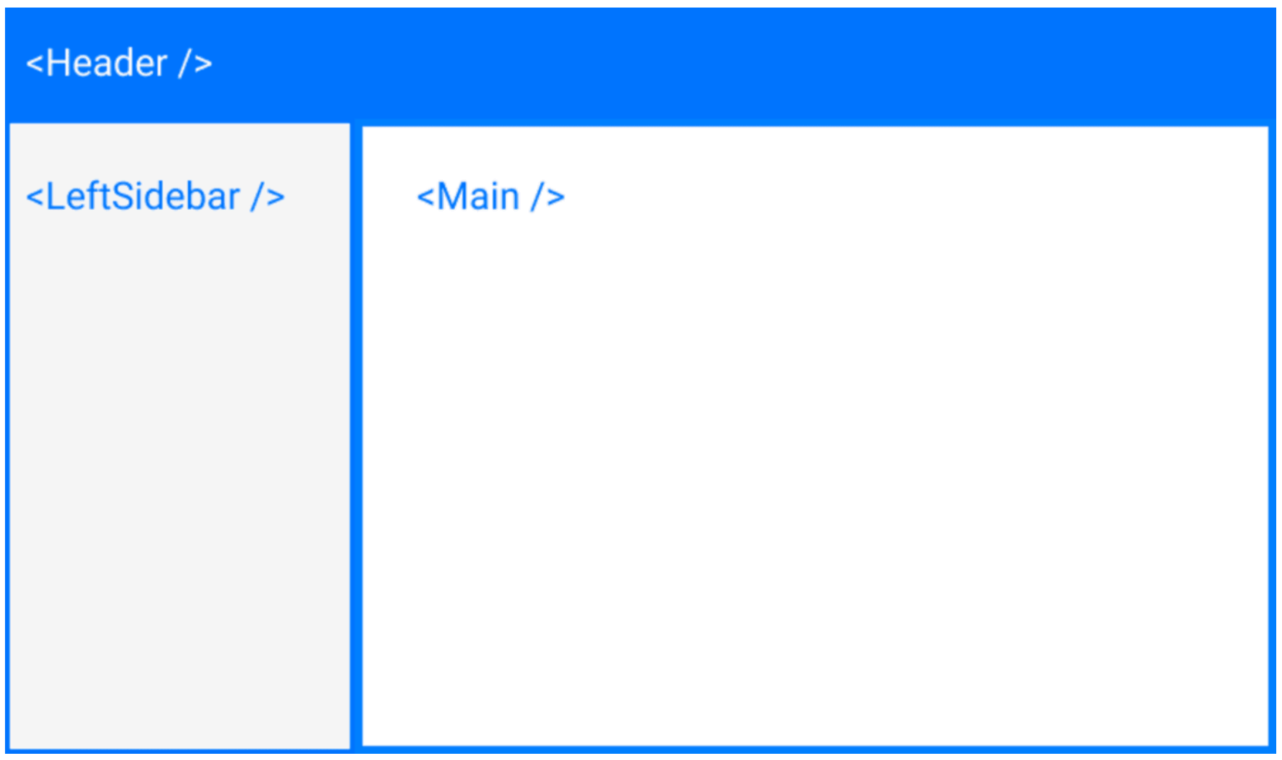
React Error Handling And Reporting With Error Boundary And Sentry

What Is An Error Boundary And Why Do We Need It?
A picture, they say, is worth a thousand words. For that reason, I'd like to talk about error boundaries using — you guessed it — pictures.
The illustration below shows the component tree of a simple React app. It has a header, a sidebar on the left, and the main component, all of which is wrapped by a root `<App />` component.

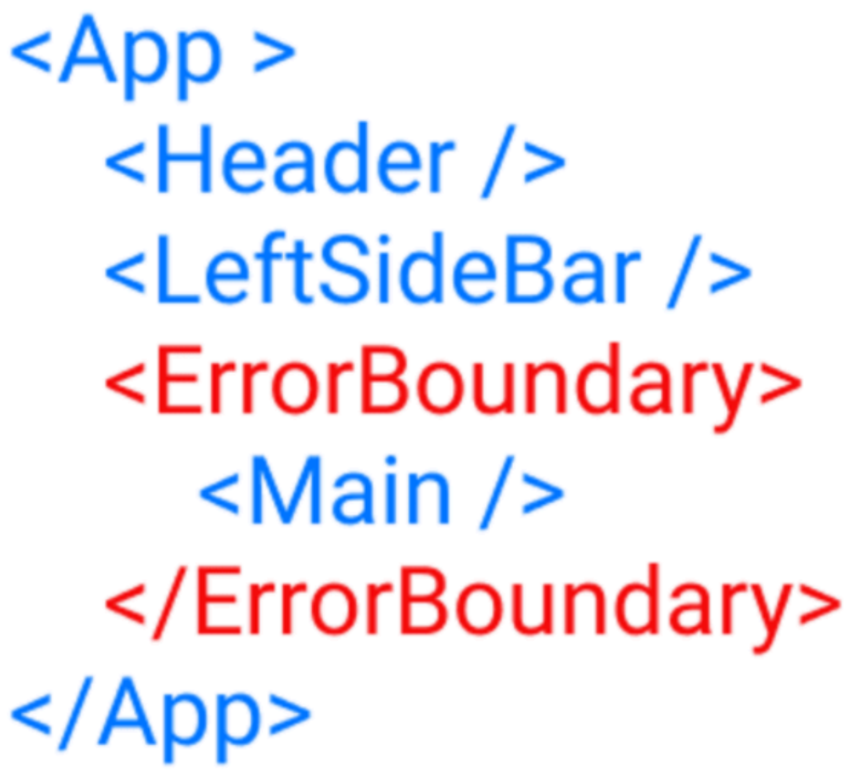
On rendering these components, we arrive at something that looks like the picture below.



In an ideal world, we would expect to see the app rendered this way every single time. But, unfortunately, we live in a non-ideal world. Problems, (bugs), can surface in the frontend, backend, developer’s end, and a thousand other ends. The problem could happen in either of our three components above. When this happens, our beautifully crafted app comes crashing down like a house of cards. React encourages thinking in terms of components. Composing multiple smaller components is better than having a single giant component. Working this way helps us think about our app in simple units. But aside from that won’t it be nice if we could contain any errors that might happen in any of the components? Why should a failure in a single component bring down the whole house?

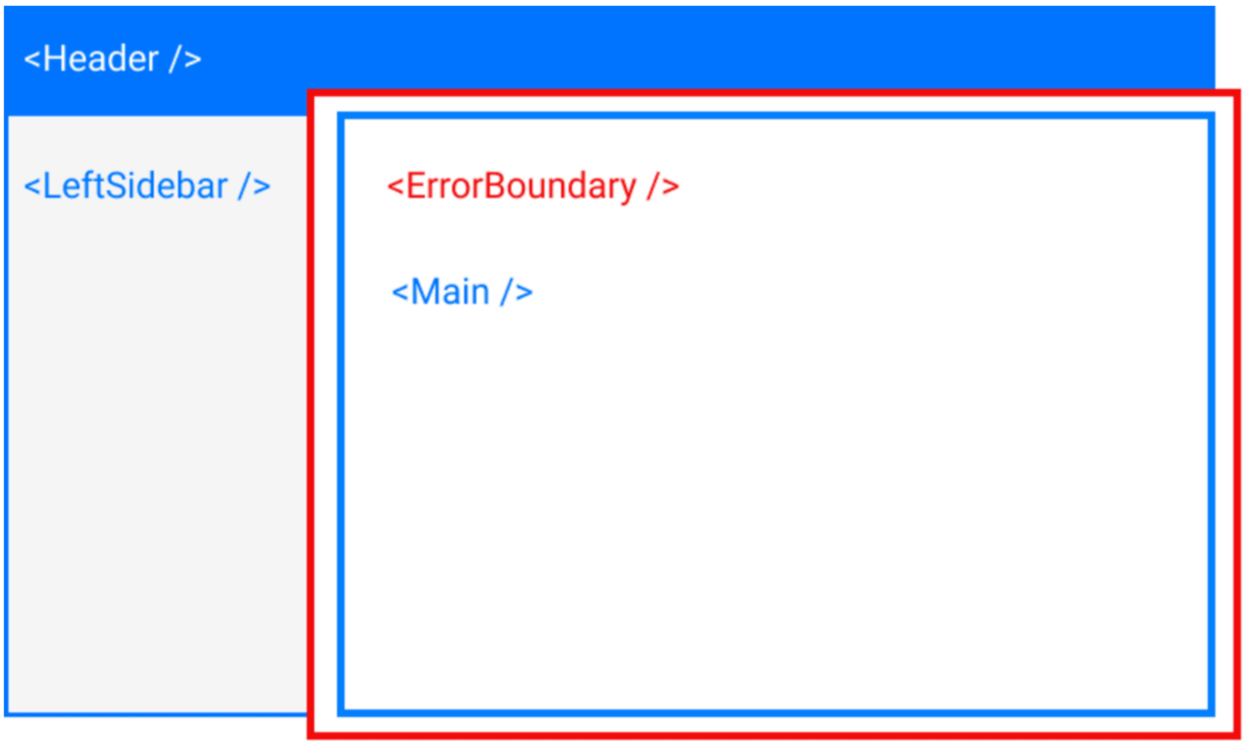
In the early days of React, this was very much the case. And worse, sometimes you couldn’t even figure out what the problem was.
React 16 came to the rescue with the concept of an “error boundary”. The idea is simple. Erect a fence around a component to keep any fire in that component from getting out.

The illustration below shows a component tree with an `<ErrorBoundary />` component wrapping the `<Main />` component. Note that we could certainly wrap the other components in an error boundary if we wanted. We could even wrap the `<App />` component in an error boundary.



```
<App >
  <Header />
  <LeftSideBar />
  <ErrorBoundary>
    <Main />
  </ErrorBoundary>
</App>
```

The red outline in the below illustration represents the error boundary when the app is rendered.



As we discussed earlier, this red line keeps any errors that occur in the `<Main />` component from spilling out and crashing both the `<Header />` and `<LeftSideBar />` components. This is why we need an error boundary. Now that we have a conceptual understanding of an error boundary, let's now get into the technical aspects.

What Makes A Component An Error Boundary?

As we can see from our component tree, the error boundary itself is a React component. According to the docs, *A class component becomes an error boundary if it defines either (or both) of the lifecycle methods* `static getDerivedStateFromError()` *or* `componentDidCatch()`.

There are two things to note here. Firstly, only a class component can be used as an error boundary. Even if you're writing all your components as function, you still have to make use of a class component if you want to have an error boundary. Secondly, it must define either (or both) of `static getDerivedStateFromError()` or `componentDidCatch()`. Which one(s) you define depends on what you want to accomplish with your error boundary.

Functions Of An Error Boundary
An error boundary isn't some dumb wall whose sole purpose in life is to keep a fire in. Error boundaries do actual work. For starters, they catch javascript errors. They can also log those errors, and display a fallback UI. Let's go over each of \these functions one after the other.

Catch Javascript Errors

When an error is thrown inside a component, the error boundary is the first line of defense. In our last illustration, if an error occurs while rendering the `<Main />` component, the error boundary catches this error and prevents it from spreading outwards.

LOGS THOSE ERRORS

This is entirely optional. You could catch the error without logging it. It is up to you. You can do whatever you want with the errors thrown. Log them, save them, send them somewhere, show them to your users (you really don’t want to do this). It’s up to you.
But to get access to the errors you have to define the `componentDidCatch()` lifecycle method.

RENDER A FALLBACK UI

This, like logging the errors, is entirely optional. But imagine you had some important guests, and the power supply was to go out. I’m sure you don’t want your guests groping in the dark, so you invent a technology to light up the candles instantaneously. Magical, hmm. Well, your users are important guests, and you want to afford them

the best experience in all situations. You can render a fallback UI with `static getDerivedStateFromError()` after an error has been thrown.
It is important to note that error boundaries do not catch errors for the following situations:

- Errors inside event handlers.
- Errors in asynchronous code (e.g. `setTimeout` or `requestAnimationFrame` callbacks).
- Errors that happen when you're doing some server-side rendering.
- Errors are thrown in the error boundary itself (rather than its children). You could have another error boundary catch this error, though.

Example :

https://reactjs.org/docs/error-boundaries.html

https://codepen.io/gaearon/pen/wqvxGa?editors=0010