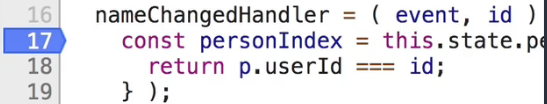
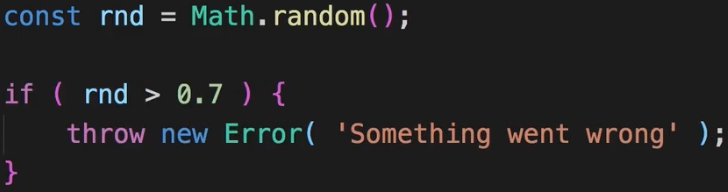
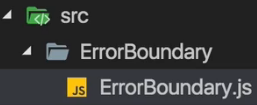
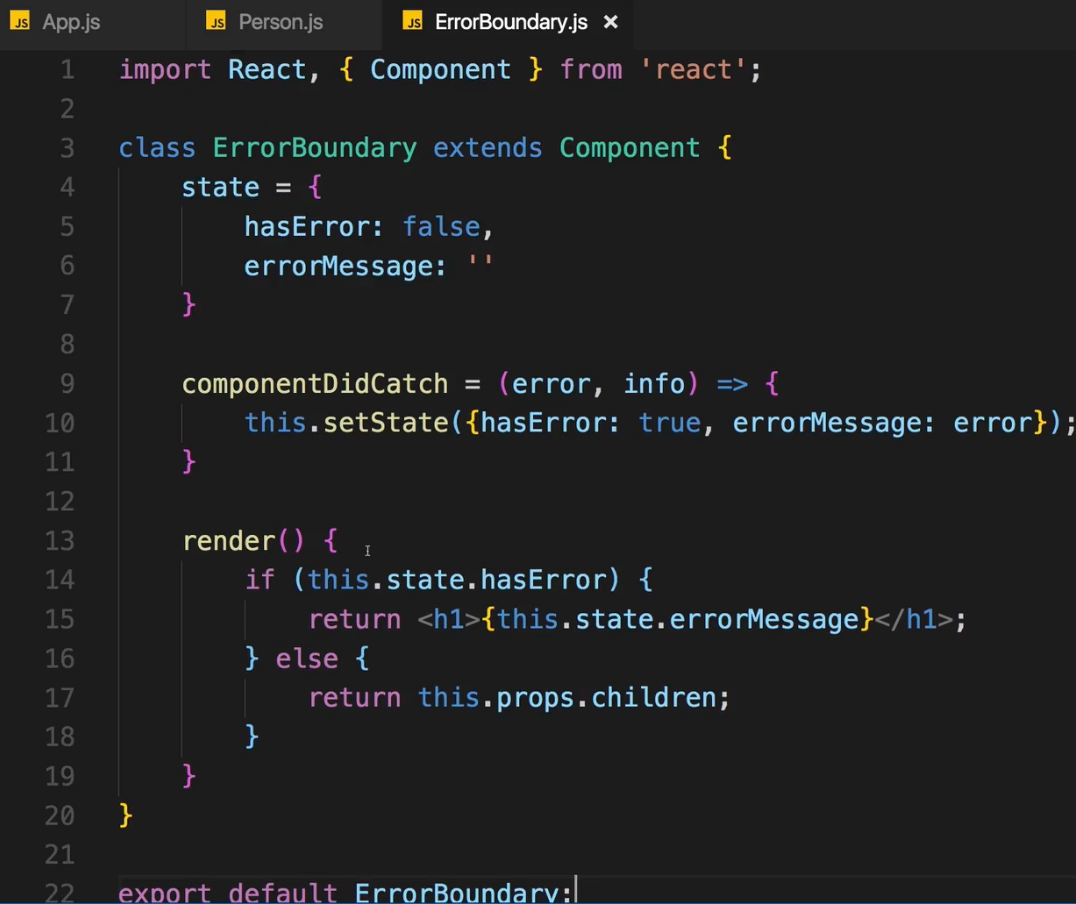
**Section 6 done: 7/7 Debugging React Apps**  
**Finding Logical Errors by Using Dev Tools & Sourcemaps**  
\* You don’t get an error message but somehow it’s not working.  
\* For example instead of p.Id === id we do p.userId === id. It will not throw an error because p.userId is undefined and you can compare undefined to something.  
\* Now what can we do about that? We can walk through our code step by step - for that I recommend using the Chrome Dev Tools. There you can go to the Sources tab. In there you will find your code structure shown under localhost. There you find a `src` folder and you actually find your code here. Now this is possible due to `sourcemaps` which are generated. Basically you could say translation files allowing the browser dev tools to go into your code as you wrote it and allow you to debug that code even though the code which is shipped to the browser will be a different one - an optimized and bundled one.  
\* We know it has to do with the nameChangedHandler so we set a break-point there by clicking the line number.  
  
=> When we type 1 character it’s now pausing this.   
\* Now inside we can use step into. We can for example aim at the `p` to see what it is.  
\* We can also see that the `p.userId` is undefined. If we hover over the `p` part we see there’s no property called userId.  
=> So using the Chrome Debugger Tools with sourcemaps which are generated for you automatically, is a powerful feature for detecting logical errors. You can walk through your code as you wrote it even though it’s not the code running in the browser.  
**Working with the React Developer Tools**  
\* Sometimes you want to know the current state of your React app.  
\* There is a cool extension you can add to Chrome - the React Developer Tools.  
\* Then in the Developer Tools you can choose React.  
\* You now see your React app as you created it. You see your components too.   
\* Down there at the bottom you also see `props` and `state` of that component.  
**Using Error Boundaries (React 16+)**  
\* Sometimes you have code which might fail at runtime and you know that but you can’t guarantee that it always works. In this case you probably wanna show a nice custom error message to the user.  
\* Let’s get an error with 30% chance.  
  
\* For this I will create a new component and this is a new feature which was added with React 16.  
\* I will name a folder `ErrorBoundary` because you can create so called ErrorBoundary components in there. The name is up to you though. The concept is called ErrorBoundary.  
  
**componentDidCatch = (error, info) => {}**  
=> This will be executed whenever a component we wrap with the Error Boundary throws an error.  
**this.props.children**  
=> That’s important when accessing props in a class component you need to add `this` in front of it.  
  
\* Now in App.js:  
  
=> And we wrap our Person component with <ErrorBoundary></ErrorBoundary>

=> Now the ErrorBoundary is so called `**higher order component**`. It’s a component which simply wraps a component with the goal of handling any errors that component might throw in the case of Error Boundary, that’s its job.  
=> **Now with the Person wrapped with ErrorBoundary we need to move this `key` to the ErrorBoundary** because this is now the outer element which we map. And the `key` always has to be on the outer element in a .map() method because that’s the element we actually replicate.  
\* **Now when you get the error you still see the error unfortunately - now that only happens to you in Development Mode, once you build this for production and ship it to a real server, it will not overwrite your page like this**. Instead, you will see whatever you render inside your ErrorBoundary.  
\* This does not mean that you should cluster your whole application with Error Boundaries - only use them when it makes sense. So basically only use them if you have some code you know that may fail. And you can’t control that.  
\* Now with all these tools you should have a rich toolset to create hardened and safe and well-working React applications.  
Resources:  
Error Boundaries: <https://reactjs.org/docs/error-boundaries.html>  
Chrome Devtool Debugging: <https://developers.google.com/web/tools/chrome-devtools/javascript/>