Cleanups

You finally made it! We're almost done. Let's clean a few things up.

WE'LL COVER THE FOLLOWING

Using the Custom Function Without a Reducer

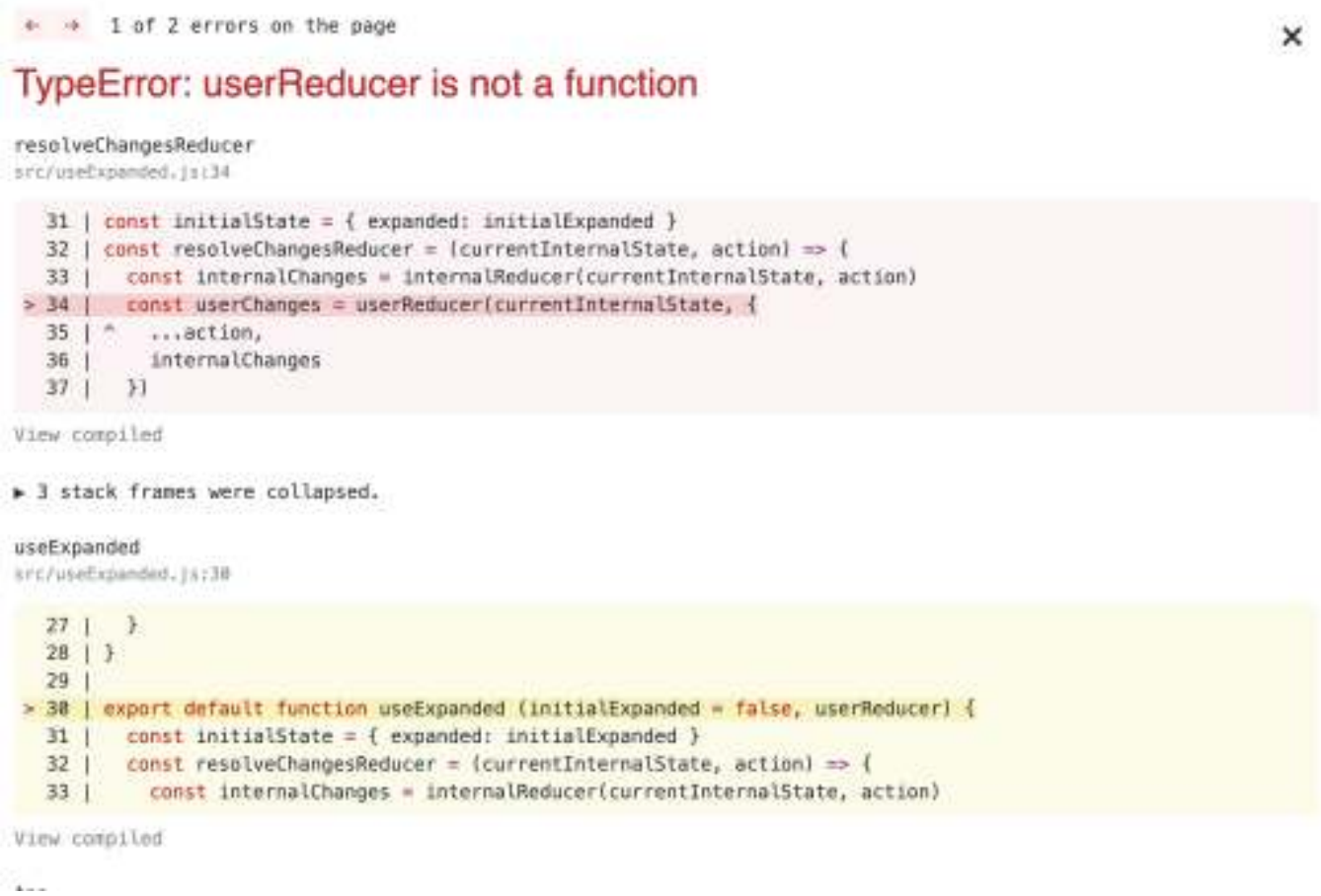
Right now, if some user other than this crazy hacker uses our custom function without passing a reducer, their app breaks.

```
useExpanded(false) // 👈 user doesn't need reducer feature
```

Try running this below to see how not passing a reducer breaks the app.

```
.Expandable-panel {
    margin: 0;
    padding: 1em 1.5em;
    border: 1px solid hsl(216, 94%, 94%);;
    min-height: 150px;
  }
```

Running the code above will get you an error like this:



Providing a Default Reducer

Let's provide a default reducer within our custom hook for users who don't need this feature.

```
// userExpanded.js
function useExpanded (
  initialExpanded = false,
  userReducer = (s, a) => a.internalChanges // 👈 default
) {
  ...
}
```

Using the ES6 default parameters a default reducer has been provided. The default user reducer takes in state and action, then returns our `internalChanges`.

With this fix, users who don't need the reducer feature don't get an error.

The Final Result

```
.Expandable-panel {
    margin: 0;
    padding: 1em 1.5em;
```

```
  border: 1px solid hsl(216, 94%, 94%);;
  min-height: 150px;
}
```

This has been such an interesting discourse!

Remember, the benefit of the state reducer pattern is the fact that it allows the user “control” over how internal state updates are made.

Our implementation has made our custom hook so much more reusable. However, how we update state is now part of the exposed API and if you make changes to that, it may be a breaking change for the user.

Regardless, this is still such a good pattern for complex hooks and components.

Quick Quiz!

1 Why have we provided a default user reducer?

COMPLETED 0%

1 of 2

We’ve finally reached the end of this course! Congrats on making it this far 😃