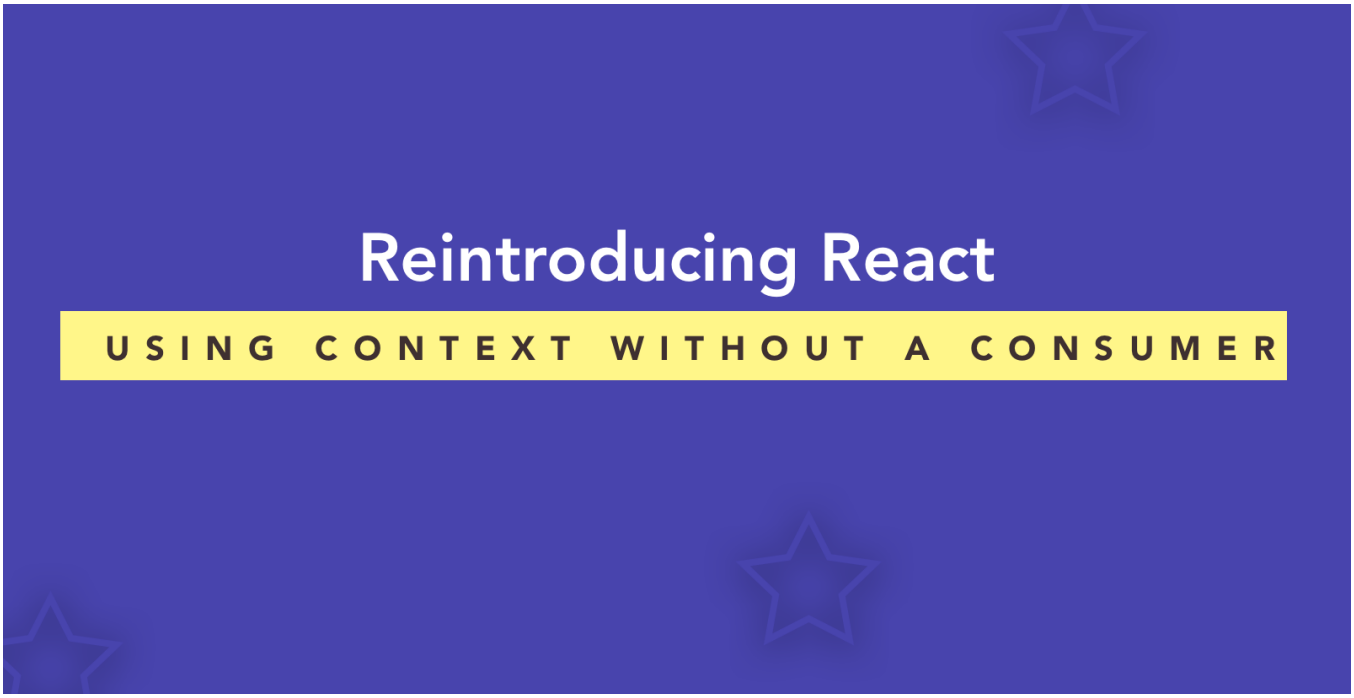
Using Context without a Consumer

In this lesson, we will see that issues can occur if we are using Context without consumers.

WE'LL COVER THE FOLLOWING

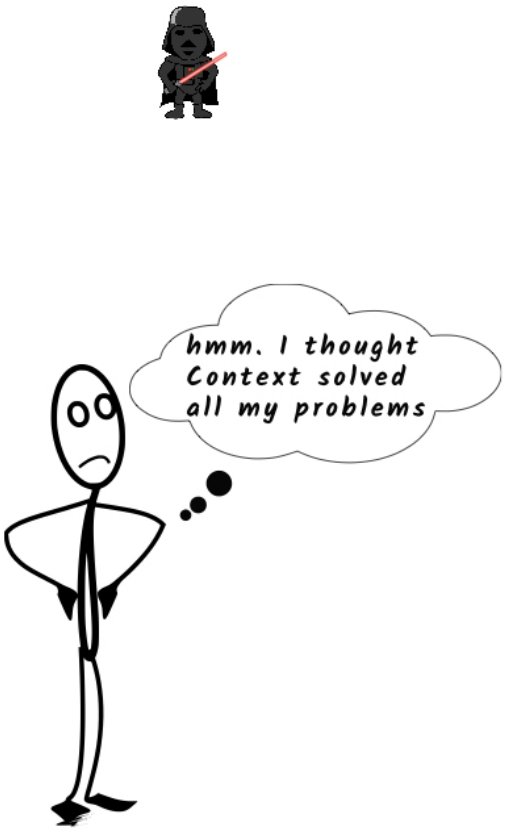


So far, John has had a great experience with the Context API. Many thanks to Mia who recommended such a great tool!

However, there's a little problem.

As John uses the context API more often, he begins to recognize a problem.

A Problem Arises




```
const Benny = () => {
  return <Consumer>
    {(position) => <GameLevelConsumer>
        {(gameLevel) => <svg />}
    </GameLevelConsumer>}
  </Consumer>
}
```

When you have multiple `Consumers` within a component, it results in having a lot of nested, not-so-pleasant code.

Here's an example.

While working on the *Benny Home Run* application, John had to create a new context object to hold the game level state of the current user.

```
// initial context object
const BennyPositionContext = createContext({
        x: 50,
        y: 50
})

// another context object for game level i.e Level 0 - 5
const GameLevelContext = createContext(0)
```

Remember, it's common practice to split related data into different context objects for reusability and performance, owing to the fact that every consumer is re-rendered when values within a `Provider` change.

With these context objects, John goes ahead and uses both `Consumer` components within the `Benny` component as follows.

```
//grab consumer for PositionContext
const { Consumer: PositionConsumer } = BennyPositionContext

// grab consumer for GameLevelContext
const { Consumer: GameLevelConsumer } = GameLevelContext

// use both Consumers here.
const Benny = () => {
  return <PositionConsumer>
    {position => <GameLevelConsumer>
        {gameLevel => <svg />}
    </GameLevelConsumer>}
  </PositionConsumer>
}
```

Do you notice that consuming values from both context objects results in a very nested code?

This is one of the more common problems with consuming data with the `Consumer` component. With multiple consumer components, you begin to have a lot of nesting.

What can we do about this?

First, when we learn about Hooks in a later chapter, you'll come to see an almost perfect solution to this. In the meantime, let's consider the solution available to `class` components via something called `contextType`.