Context

Context is a concept in React that allows you to share state across the entirety of an application. With what we have done thus far, we learned how to share data from a parent down to its children and how to track state for the entire application. What if we might have data shared with siblings and/or children multiple layers down? While we could absolutely use lifted state to find a common parent and then pass the state down as props, this can cause what is known as prop drilling. Simply put, it is when you have some prop continually passed down to children until it eventually reaches where it needs to be. This is especially true if you're passing them as props through components that don't need the data for any reason other than to pass it down. Example:

Parent -> ChildA -> ChildB -> ChildC

```
import React, { useState } from "react";

function Parent() {
  const [firstName, setFirstName] = useState("firstName");
  const [lastName, setLastName] = useState("LastName");
  return (
    <>
      <div>This is a Parent component</div>
      <ChildA fName={fName} lName={lName} />
    </>
  );
}

function ChildA({ fName, lName }) {
  return (
    <>
      This is ChildA Component.
      <ChildB fName={fName} lName={lName} />
    </>
  );
}

function ChildB({ fName, lName }) {
  return (
    <>
      This is ChildB Component.
      <ChildC fName={fName} lName={lName} />
    </>
  );
}

function ChildC({ fName, lName }) {
  return (
    <>
      This is ChildC component.
      <h3> We can use the data from the parent component like this:</h3>
```

```
        <h4>{fName}</h4>
        <h4>{lName}</h4>
      </>
    );
}

export default Parent;
```

If you're running into that issue, it's usually better to consider something like Context.

Context, which is built into React, allows you to create one parent container and the "context" of data to be passed down. If you think of passing the data down as props to any children, that is basically what you're doing, just without the need to actually bind those properties. It's particularly useful for sharing data that can be considered "global" for a tree of React components, such as the current authenticated user, theme, or preferred language. If you have the need for this pattern, it's actually quite simple to set up. You find the top most level where the state will start (where it would normally be lifted to) and declare your Context there.

- First you start by creating the context in a separate file:

```
import React, { createContext } from "react";
// The name you use should make sense with the state you're storing. Something
like ThemeContext would be perfect if we want to store whether the user's browser
theme is light or dark.
// The value inside of the parenthesis is the starting value. You can set it as a
default value, null, or an empty string.
export const ThemeContext = createContext("light");
```

- Then you import and set the starting value in the top most parent you need it in, and wrap that components JSX in the Context you've created:

```
//...
import { ThemeContext } from "./themeContext";

// IN THE JSX
return (
  <ThemeContext.Provider value={"dark"}>
    <!-- Your standard JSX for the component would be here -->
  </ThemeContext.Provider>
);
```

- Remove any props being passed to children that are now handled by the new Context
- Import the context and pull out any values needed in the child(ren).

```
// ...
// add useContext to your imports from 'react'
import { ThemeContext } from "./themeContext";
```

```
// Inside of your functional component:
// If the context only has one piece of state, you can do this:
const theme = useContext(ThemeContext);
// However, if you have multiple values, you have to de-structure it like this:
const { destructured, state } = useContext(ThemeContext);
```

The above shows how you could add it to an existing component. What if we wanted to wrap the whole application in a context though? These steps simply need to be:

- In the file you're creating your Context in, create a functional component like this:

```
import React, { useContext } from "react";
export const ThemeContext = createContext(null);
// Creating our own hook that just uses the ThemeContext for us. This is not the
only way to do it, but it simplifies things later.
export const useThemeContext = () => useContext(ThemeContext);

// Creating a provider functional component. It needs props as you'll see below
export function ThemeProvider(props) {
  // Any functions or state to share with the application goes in here. Could be
something like:

  // const [theme, setTheme] = useState("dark");

  // We return the created Provider here
  return (
    <ThemeContext.Provider>
      <!-- This line allows for any child components to render // Your child
      components will be the components inside of the provider when called upon
      think child html elements -->
      {props.children}
    </ThemeContext.Provider>
  );
}
```

- You can now wrap the above provider component around the whole app in your index.js:

```
root.render(
  <ThemeProvider>
    <App />
  </ThemeProvider>
);
```

- Any component can now use the context just as in our previous example.