Creating a simple strongly-typed context for function components

In this lesson, we will learn what context is and discover how TypeScript can infer the type for it.

WE'LL COVER THE FOLLOWING

Understanding React context

React context allows several components in a tree to share some data. It's more convenient than passing the data via props down the component tree.

The context is *provided* at a point in the component tree, and then all the children of the provider can access the context if they wish.

Creating a context

A common use case for using context is to provide theme information to components in an app. We are going to provide a color value in a context that components can use.

Let's start by creating our theme using Reacts `createContext` function:

```
const defaultTheme = "white";
const ThemeContext = React.createContext(defaultTheme);
```

We are required to provide a default value for the context, which in our case is `"white"`.

What do you think the type of the context value has been inferred as?

Show Answer

Creating a Context provider

Next, we are going to create the provider component:

```
type Props = {
  children: React.ReactNode
};
export const ThemeProvider = ({ children }: Props) => {
    const [theme, setTheme] = React.useState(defaultTheme);

    React.useEffect(() => {
        // We'd get the theme from a web API / local storage in a real app
        // We've hardcoded the theme in our example
        const currentTheme = "lightblue";
        setTheme(currentTheme);
    }, []);

  return (
    <ThemeContext.Provider value={ theme }>
      {children}
    </ThemeContext.Provider>
  );
};
```

We hold the theme value in the state. This means that when it changes, React will automatically re-render the provider's children with the new theme.

We get the current theme value using React's `useEffect` hook and update the `theme` state value.

Our theme provider component returns the `Provider` component within the context with our theme value. The provider is wrapped around all the children in the component tree.

children in the component tree.

Creating a custom hook for consuming the context

We can create a custom hook that will allow function components to consume our context:

```
const useTheme = () => React.useContext(ThemeContext);
```

What do you think the return type of `useTheme` is?

Show Answer

Adding the provider to the component tree

The `ThemeProvider` component can now be placed in an appropriate position in the component tree.

```
const App = () => (
    <ThemeProvider>
        <Header />
    </ThemeProvider>
);
```

The way that React context works is that components below its provider will have access to the context, but components above it won't. So, the `Header` component will have access to the context.

Consuming the context

The `Header` component can access the context by using the `useTheme` hook we created. This allows the header component to render an element that has its background color set to the theme color:

```
const Header = () => {
    const theme = useTheme();
    return (
        <div style={{backgroundColor: theme}}>Hello!</div>
```

```
    );
}
```

A working version of `ThemeContext` is available on the link given below. When the app is run, *Hello* will appear with a light blue background.

Open working version

Wrap up

The type for the context is inferred correctly if a sensible default is provided when it is created. If the context is providing values that consumers don't change, then this is fine. However, what if we want the user to change the theme? In this case, our context would need to provide a function for updating the theme, and this can't really be provided as a default value.

In the next lesson, we will extend our theme context so that consumers can update the value.