Creating a complex strongly-typed context for function components

In this lesson, we'll learn how to implement a strongly-typed context where consumers can change values within it.

WE'LL COVER THE FOLLOWING

- Explicitly setting the context type
- Using the enhanced context in the provider
- Adding an option to change the theme in the `Header` component
- Wrap up

Explicitly setting the context type

In this lesson, we are going to enhance the context from the last lesson so that the theme can be updated by consumers.

In the last lesson, the type for the context was inferred from the default value, which was a simple `string`. The type for our enhanced context is going to be a little more complex:

```
type ThemeContextType = {
  theme: string;
  setTheme: (value: string) => void;
};
```

So, there will be a `theme` property containing the current value for the theme and a `setTheme` method to update the current theme.

React's `createContext` function expects us to supply a parameter for initial context value. We can supply a default value for the `theme` property, but it doesn't make sense to provide a default implementation for the `setTheme` method. So, a simple approach is to pass in `undefined` as the initial value:

```
const ThemeContext = React.createContext(undefined);
```

What do you think is the inferred type for the context value in this case?

Show Answer

So, `ThemeContext` isn't typed as we require at the moment. How do you think we can explicitly specify the type for the context when using `createContext`?

Show Answer

Therefore, we can type our context as follows:

```
const ThemeContext = React.createContext<ThemeContextType | undefined>(
  undefined
);
```

Using the enhanced context in the provider

The modification to the context provider is to the value we provide from it. Instead of a simple `string`, it is now an object containing the `theme` property and the `setTheme` method:

```
export const ThemeProvider = ({
  children
}: Props) => {
  ...

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

Adding an option to change the theme in the `Header` component

The `useTheme` custom hook remains the same as in the last lesson. The `App` component remains the same as well.

We are going to change the `Header` component though so that the user can change the theme:

```
const Header = () => {
  const { theme, setTheme } = useTheme()!;
  return (
    <div style={{ backgroundColor: theme }}>
      <select value={theme} onChange={e => setTheme(e.currentTarget.value
)}>
        <option value="white">White</option>
        <option value="lightblue">Blue</option>
        <option value="lightgreen">Green</option>
      </select>
      <span>Hello!</span>
    </div>
  );
};
```

We destructure the current theme value as well as the `setTheme` method from the `useTheme` hook. Notice that we have put an exclamation mark (`!`) after the call to the `useTheme` hook to tell the TypeScript compiler that its return value won't be `undefined`.

We have also added a drop down that has options to change the theme to *White*, *Blue*, and *Green*. The value of the drop down is set to the current theme value, and when this is changed, it calls the context's `setTheme` method to update this in the shared state.

The full implementation of the context is available by clicking the link below. Give it a try and change the theme value and see the background change color.

Open full implementation

Wrap up

Generally, we will need to create a type for a context and explicitly define the

type when the context is created. Often the initial value for the context is `undefined`, so the context type is usually a union type that contains `undefined`.

Next, we will learn how to create a strongly-typed context with class components.