COURSEWARE

State Management

Contents

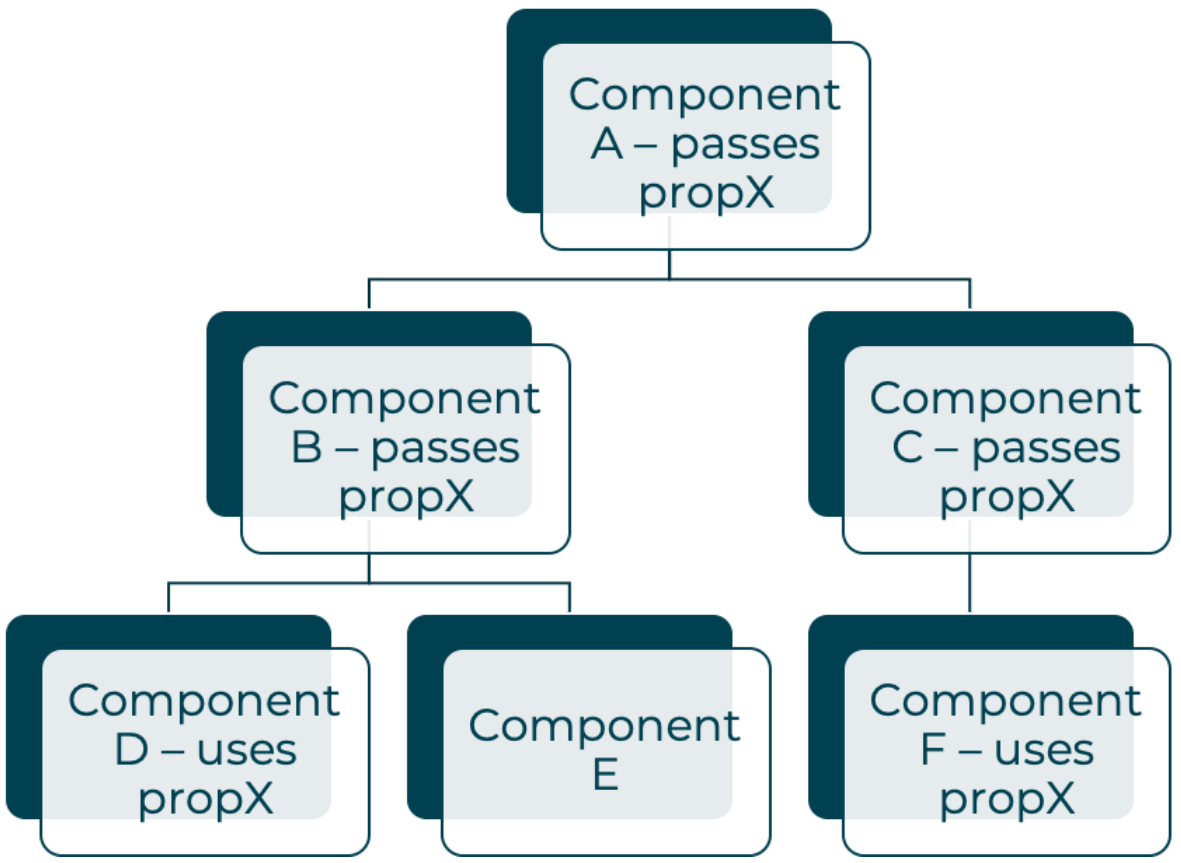
Overview

In this module, we will look at React's state management.

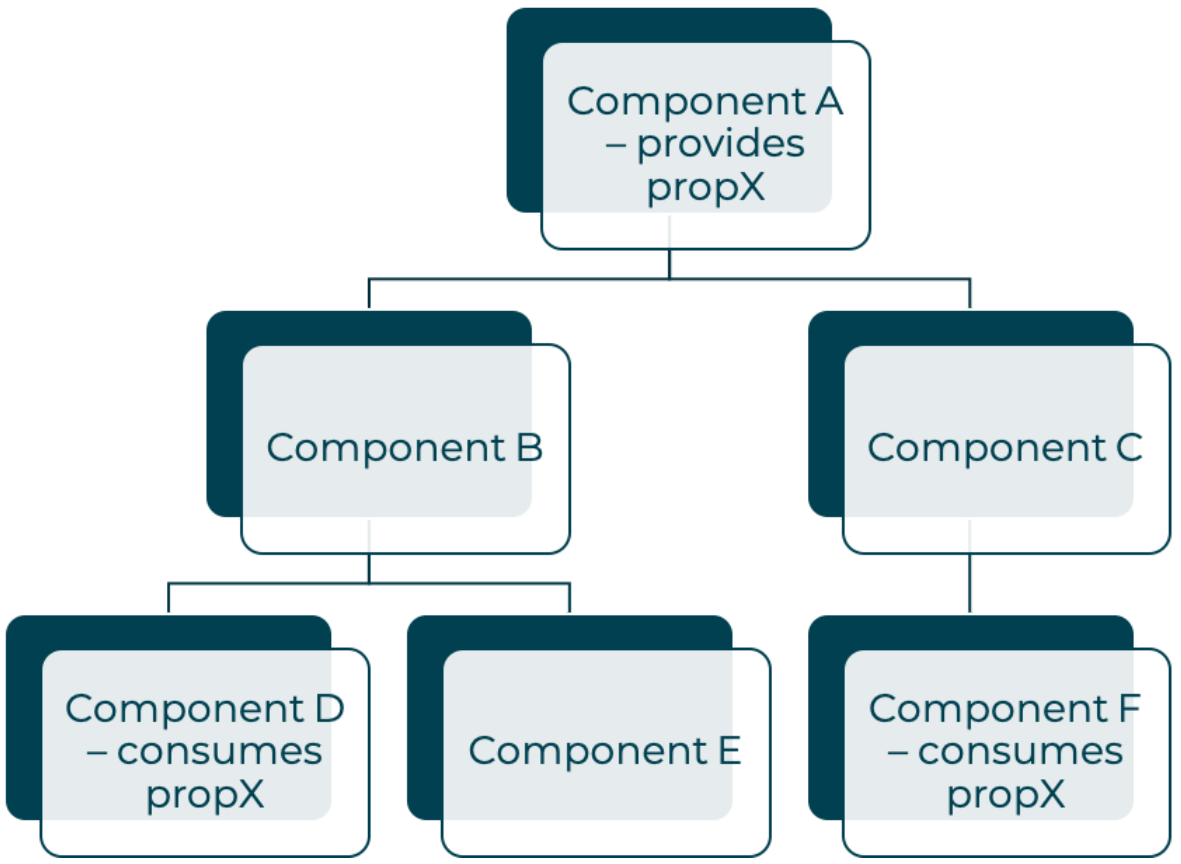
What is Context?

Data is passed top to bottom in React - or parent to child - via props.

This can add complexity into a React app the further you go down the component tree.



Context allows for the sharing of values between components without having to pass a prop through every level of the hierarchy.



When multiple components across component trees need access to the same data, we would use context.

For example, if a set of colour styles are needed for components across an application.

Note: we should not be using context to simply to bypass components in a single tree.

Creating Context

React provides a helper function called `createContext` to actually create the React Context.

```
import {createContext} from `react`;
const MyContext = createContext();
export default MyContext;
```

Context must then be 'provided' to a tree of components;

```
import MyContext `./MyContext`;
import {B, C, D, E} from `MyComponents`;

const A = () => (
  <MyContext.Provider value={myData}>
    <B>
        <D/><E />
    </B>
    <C>
        <F />
    </C>
  </MyContext.Provider>
);
export default A;
```

Context and Hooks

We need to provide a function to use in a component that has a context and uses the `useContext` hook.

`useContext` accepts context object created by calling createContext.

```
import {useContext} from `react`;
import MyContext from `./MyContext`;

const useMyContext = () => {
    const myContext = useContext(MyContext);
    return myContext;
}

export default useMyContext
```

Any child component of the `Provider` can use this context.

The current context value is determined by the value prop of the nearest context `Provider` above the calling component in the tree.

When the `Provider` updates, `useContext` hook triggers re-render with latest context value passed to `Provider`.

```
import useMyContext `./useMyContext`;

const D = () => {
  const { myData } = useMyContext();
  return (<p>{myData}</p>);}
export D;
```

Reducers

Reducers are basically JavaScript functions!

They take 2 arguments; the previous state and an action.

These arguments are then used to return a completely new state.

There are 2 basic principles for using reducers:

- State is never mutated
- The Previous state can be used as part of the new state by using the spread operator `…state`

```
const myReducer = (state, action) => {
    if(action) {
        return {…state, action.payload};
    }
    return state;
}
```

Actions

`action` is usually an object supplied as part of a call to the reducer.

It helps to identify the action that should be taken at this point to produce the new state.

Action usually has at least 2 properties:

- `type` – a string to identify the action by
- `payload` – the data that has been dispatched as part of the update call

useReducer Hook

`useReducer` can be used as an alternative to the `useState` hook.

It needs to be passed a reducer function, and can optionally be passed an initial state:

```
const [state, dispatch] = useReducer(myReducer, initialValue);
```

`state` is paired with a dispatch function that can be called
dispatch is used to send action objects:

```
const getData = async () => {
    const payload = await //some data call
    dispatch({type: `getData`, payload});
}
```

`useReducer` is usually preferable to `useState` when:

- Complex state logic involving multiple sub-values exists
- When next state depends on the previous state

Other Notable Hooks

- `useCallback` – Returns a memorized callback – only changes if one of the dependencies has changed
- `useMemo` – Returns a memoized value – only recomputes memorized value when one of the dependencies changes – performance optimisation technique
- `useLayoutEffect` – identical to `useEffect` but fires synchronously after all DOM mutations – can read layout from DOM and synchronously re-render
- `useDebugValue` – can be used to display a label for custom hooks in the React Developer Tools

It is not recommended to add to every hook – most valuable for custom Hooks that come as part of shared libraries.

Tutorial

In this tutorial, we are going to look at how we can use Context to share data for a tree of React Components.

Below, we manually thread through a "theme" prop in order to style the button component:

```
const App = () => {
    return <NavigationBar theme="dark"/>
}

const NavigationBar = (props) => {
    // The NavigationBar component must take an extra 'theme' props and pass it
to the themeButton
    // This can become painful if every single button in the app needs to know
the theme
    // because it would have to be passed through all the components
    return <ThemeButton theme={props.theme}/>
}

const ThemeButton = (props) => {
    return <Button theme={props.theme}/>;
}
```

Using Context, we can avoid passing props through intermediate elements. Context lets us pass a value deep into the component tree without explicitly threading it through every element.

1. Create a context for the current theme - with light as the default

```
import {createContext} from 'react';
const ThemeContext = createContext('light');
```

2. In a component named `App` use a provider to pass the current theme to the tree below. Feel free to change the current value.

```
const App = () => {
    return(
        <ThemeContext.Provider value="dark">
            <Toolbar/>
        </ThemeContext>
    );
}
```

3. The `Toolbar` component doesn't have to pass the theme down explicitly anymore.

```
const Toolbar = () =>{
    return <ThemedButton/>
}
```

4. Assign a contextType in the `ThemedButton` class to read the current theme context.

```
// React will find the closes theme provider above and use its value.
import ThemeContext from "./ThemeContext";
import {useContext} from 'react';

const ThemeButton = () => {
    const value = useContext(ThemeContext);
    return <button className={`btn btn-${value}`}>{value}</button>
}
```

Let's have a look at a more complex example with dynamic values for the theme.

1. Create a `theme-context.js` file with the following values:

```
//theme-context.js
export const themes = {
    light: {
        background: '#ff0000'
    },
    dark:{
        background: '#696969'
    },
};
```

2. In the same file create a const named `ThemeContext` which sets the default value to 'themes.dark' using `createContext()`:

```
import {createContext} from 'react'
export const ThemeContext = createContext(themes.dark);
```

3. Create a component called `Themed-button.jsx` which imports the `'ThemeContext'` and takes in props as arguments:

```
import ThemeContext from './theme-context'
const ThemedButton = (props) => {

}
```

4. Call `useContext` and supply `ThemeContext` as argument, place the expression in a const:

```
const theme = useContext(ThemeContext);
```

5. In the return method, return a button with the following attributes:
    1. Spread operator which encapsulates all values passed in as props
    2. Style attribute with the background colour set to the current theme background colour

```
...
return(
  <button {...props} style={{backgroundColor:
theme.background}}>Change Theme</button>
);
...
```

6. Export the class as default

7. Create an intermediate function called 'toolbar' which returns 'ThemedButton' and pass in an onClick prop that calls a function that is passed into Toolbar:

```
const Toolbar=(props)=>{
    return(
        <ThemedButton onClick={props.changeTheme}> Change Theme
</ThemedButton>
    );
}
```

8. Create a `Home` Component called `HomePage.jsx`

9. Import the following:

    1. `ThemeContext` from 'theme-context'

    2. `ToolBar`

```
import { ThemeContext} from './theme-context';
import ToolBar from './ToolBar';
```

10. Create a state, give it a default value of `themes.light` (imported from 'theme-context');

```
...
import {useState} from 'react';
import {theme} from './theme-context';

const Home = () => {
   const [theme, setTheme] = useState(themes.light);
}
```

11. Create a `toggle theme` method which sets the state to the opposite of the current value of 'theme':

```
...
const toggleTheme = () => {
    if(theme == themes.dark){
        setTheme(themes.light);
    }else{
        setTheme(themes.dark);
    }
}
```

12. In the return method, return use the `ThemeContext.Provider` with the value of the theme in the state:

```
    return(
        <ThemeContext.Provider value={theme}>

        </ThemeContext.Provider>
    );
```

13. Inside the `ThemeContext.Provider` tag, call the `ToolBar` component with a `changeTheme` prop which passes the function `toggleTheme`

```
...
<ToolBar changeTheme={toggleTheme}/>
...
```

14. Export the Homepage as default, import into `App.js` and run with npm start.

> This tutorial demonstrated how to change the colour of a button using Context.

▶ Code

Exercises

Using Context - Create a simple app that manages users logging in.