Creating a Counter App with useReducer

In this lesson, we create a tiny counter app with useReducer instead of useState, as in the previous lesson.

WE'LL COVER THE FOLLOWING

Why useReducer

The `useState` hook is fine for most cases. The `useReducer` hook is a variant of `useState` which allows updating logic separately; it is also known as a **reducer**.

While it's a preference whether or not you choose to use the *reducer*, it's still good to learn the difference.

The code

```
import React, { createContext, useContext, useReducer } from 'react';
```

We import `useReducer` instead of `useState`.

```
const initialState = {
  count1: 0,
  count2: 0,
};
```

This is the same as the previous example. Note: If we add `count3`, we'd want

to add it in the `initialState` too, although this is not necessary.

```
const reducer = (state, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return {
        ...state,
        [action.name]: state[action.name] + 1,
      };
    case 'DECREMENT':
      return {
        ...state,
        [action.name]: state[action.name] - 1,
      };
    default:
      return state;
  }
};
```

This defines state updating logic. This pattern is popular in Redux.

Note that using `action.name` for the state property might not be a recommended pattern. It would be better to be more specific in reducers.

```
const useValue = () => useReducer(reducer, initialState);
```

This is a handy custom hook that we can use in the provider below.

```
const Context = createContext(null);

const useGlobalState = () => {
  const value = useContext(Context);
  if (value === null) throw new Error('Please add GlobalStateProvider');
  return value;
};

const GlobalStateProvider = ({ children }) => (
  <Context.Provider value={useValue()}>{children}</Context.Provider>
);
```

The code above is the same as in the previous lesson.

```
const Counter = ({ name }) => {
  const [state, dispatch] = useGlobalState();
```

```
  const [state, dispatch] = useGlobalState();
  return (
    <div>
      {state[name]}
      <button onClick={() => dispatch({ type: 'INCREMENT', name })}>+1</bu
tton>
      <button onClick={() => dispatch({ type: 'DECREMENT', name })}>-1</bu
tton>
    </div>
  );
};
```

In this component, we use `dispatch` instead of `setState`.

```
const App = () => (
  <GlobalStateProvider>
    <h1>Count1</h1>
    <Counter name="count1" />
    <Counter name="count1" />
    <h1>Count2</h1>
    <Counter name="count2" />
    <Counter name="count2" />
  </GlobalStateProvider>
);

export default App;
```

The code for the app component is the same as in the previous lesson.

The working example

Check out the app below.

```
import React from 'react';
require('./style.css');

import ReactDOM from 'react-dom';
import App from './app.js';

ReactDOM.render(
  <App />,
  document.getElementById('root')
);
```

Exercise

How would you add a double button “x2”?

Notes for Redux users

If you are familiar with Redux, you might wonder why we aren't using action creators. In this lesson, we prefer simpler dispatch. Even in practical apps, we wouldn't need action creators if we had a type system like TypeScript.

Next

In the next chapter, we will introduce React Tracked and learn about render optimization.