Andrew Franks

Eric Lingren

FSW-123

11/4/2021

Researching the useContext and useReducer Hooks in React

According to reactjs.org, hooks are defined as functions that “let you use state and other React features without writing a class”. As a JavaScript library that has revolutionized frontend Web Development, React.js has come far in the world of component-based development. Hooks are just another example of evolving solutions that make the work of those who them easier, and more efficient.

Using React props to pass data down the component tree has been known as standard, but can quickly make things complicated as the React tree becomes larger and larger. Known as “prop-drilling”, numerous conversations have been sparked, and multiple technologies have been formed to create simpler methods; this is where useContext comes in. The useContext hook allows for function components to interact with the Context API. This solves prop-drilling by simplifying state management.

Within useContext, two other components are included that allow the relevant sharing of data, Provider, and Consumer. Within the parent component (where state is declared), useContext's Provider component can be given a prop-value. This value will be seen as global, and can be passed to any component enclosed within the said Provider component. Where the data needs to be used, the useContext hook can be imported from React, and created context from another file can be used as an argument to set context variables. These variables can then be used in the return method to render accordingly on the DOM.

A reducer is a mathematical pattern that reworks a statement into something simpler. In programming, reducers can be used to combine things. For example, a reduced array could return the sum of every element in its index. With regards to React.js, the useReducer hook can use this design to control state. Similar to useState, useReducer can set state and trigger re-renders, but allows for custom logic. With useReducer, function components can manage state using the reducer pattern.

The useReducer hook is capable of setting state, and updating state with an “action dispatcher”. This action dispatcher calls a “reducer”, which is a function declared outside of the useReducer hook. The reducer function works logic on state, and returns the result to the useReducer hook. For example, useReducer can be used to keep track of how many times a button is clicked. As an argument, the useReducer hook can call a reducer function, which increments our current state (set at zero, in this case) by one, and then updates state.

Like useState, useReducer is a powerful tool that can not only help with code simplification, but can also eliminate the need for third party state management libraries like Redux. The Todo assignment can be re-factored using one or both of these hooks. To useContext within the project, we first have to create context in another file and import it where it’s needed. In our App component, we can wrap the TodoList component with the context.Provider component, and pass in our “todos” list as a value prop. This same method can be used in our TodoList component to pass down other data to different child components.

The useReducer hook can be used in our project to keep track of the todo list itself and its checkboxes. After importing useReducer, we can create a reducer function that toggles the checked status of a checkbox. Now we can appropriately manage state in order update whether a certain list item is completed or not, which would result in a relevant CSS change.