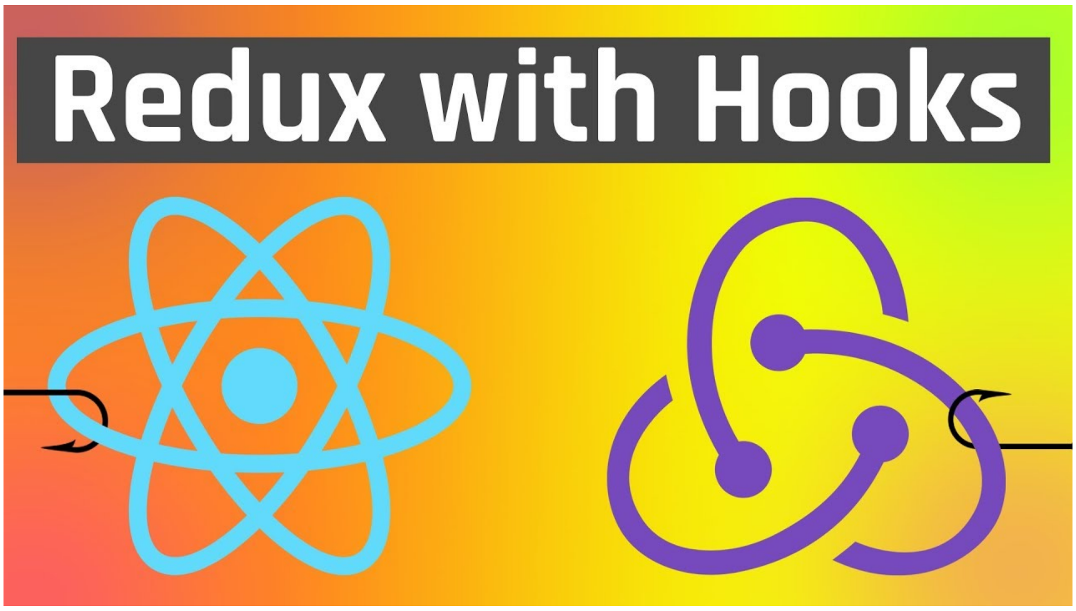
Redux with Hooks

When React released the Hooks update, redux followed it by offering a set of hook APIs as an alternative to the existing connect() Higher Order Component. These APIs allow you to subscribe to the Redux store and dispatch actions without having to wrap your components in connect().

These hooks were first added in v7.1.0.



Provider and create createStore

Provider" and "createStore()": As with connect(), we should start by wrapping our entire application in a <Provider> component to make the store available throughout the component.

```
const store = createStore(rootReducer)
ReactDOM.render(
 <Provider store={store}>
   <App />
 </Provider>,
 document.getElementById('root')
```

useSelector()

`mapStateToProps` and `connect` argument:

We replace our famous `mapStateToProps` and `connect` argument with this selector hook:

`useSelector()` which allows us to extract data from the Redux state store.

It will be called with the entire state of the Redux store as its only argument.

It will be executed each time the function component is rendered (unless its reference has not changed since the previous rendering of the component which results in a set of cache that can be returned by the hook without re-executing the selector).

useSelector() also subscribes to the Redux store and runs our selector each time an action is sent.

With `mapStateToProps`, all individual fields are returned in a combined object. It doesn't matter if the return object is a new reference or not. connect () just compares the individual fields. With `useSelector()`, returning a new object every time will always force a re-render by default. If you want to retrieve multiple values from the store.

When using `useSelector` with an inline selector, a new instance of the selector is created whenever the component is rendered.

This works as long as the selector does not maintain any state. However, memoizing selectors (e.g. created via `useSelector()` from reselect) have an internal state and that is why they must be used with care.
When the selector only depends on the state, ensure that it is declared outside of the component so that the same selector instance is used for each render.

useSelector(): exemple

Example : In the example below, we are going to use the `createSelector` nad `useSelector`.

```
import React from "react";
import { useSelector } from "react-redux";
import { createSelector } from "reselect";

const selectNumOfDoneTodos = createSelector(
 state => state.todos,
 todos => todos.filter(todo => todo.isDone).length
);
```

```
export const DoneTodosCounter = () => {
  const NumOfDoneTodos = useSelector(selectNumOfDoneTodos);
  return <div>{NumOfDoneTodos}</div>;
};

export const App = () => {
  return (
    <>
      <span>Number of done todos:</span>
      <DoneTodosCounter />
    </>
  );
};
```

useDispatch()

mapDispatchToProps: This function is replaced by a **“useDispatch”** function. This function returns a reference to the dispatch function from the Redux store. We may use it to dispatch actions as needed.

```
import React from "react";
import { useDispatch } from "react-redux";

export const CounterComponent = ({ value }) => {
  const dispatch = useDispatch();

  return (
    <div>
      <span>{value}</span>
      <button onClick={() => dispatch({ type: "increment-counter" })}>
        Increment counter
      </button>
    </div>
  );
};
```

useCallback()

Pass an inline callback and an array of dependencies. `useCallback` will return a memoized version of the callback that only changes if one of the dependencies has changed.

This is useful when passing callbacks to optimized child components that rely on reference equality to prevent unnecessary renders.

```
const memoizedCallback = useCallback(
  () => {
    doSomething(a, b);
  },
  [a, b],
);
```

Custom context

The `<Provider>` component allows us to specify an alternate context via the context prop. This is useful if we're building a complex reusable component and we don't want our store to collide with any Redux store our consumers' applications might use.

To access an alternate context via the hooks API, use the hook creator functions:

```
import React from "react";
import {
 Provider,
 createStoreHook,
 createDispatchHook,
 createSelectorHook
} from "react-redux";

const MyContext = React.createContext(null);

// Export your custom hooks if you wish to use them in other files.
export const useStore = createStoreHook(MyContext);
export const useDispatch = createDispatchHook(MyContext);
export const useSelector = createSelectorHook(MyContext);
```

```
const myStore = createStore(rootReducer);


export function MyProvider({ children }) {
 return (
   <Provider context={MyContext} store={myStore}>
     {children}
   </Provider>
 );
}
```

‹ Previous   next ›