The ultimate guide to the React `useReducer` Hook

March 2, 2021 · 12 min read

Introduction

`useReducer` is one of the additional Hooks that shipped with React 16.8. An alternative to the `useState` Hook, it helps you manage complex state logic in React applications. When combined with other Hooks like `useContext`, `useReducer` can be a good alternative to Redux or MobX — indeed, it can sometimes be an outright better option.

This is not to knock Redux and MobX, as they are usually the best options to manage global state in large React applications. But more often than necessary, many React developers jump into these third-party state management libraries when they could have effectively handled their state with Hooks.

Coupled with the complexity of getting started with a third-party library like Redux and the amount of boilerplate code needed, managing state with React Hooks and the Context API is quite an appealing option since there's no need to install an external package or add a bunch of files and folders to manage global state in our application.

But the golden rule still remains: component state for component state, Redux

`useReducer` is used to store and update states, just like the `useState` Hook. It accepts a `reducer` function as its first parameter and the initial state as the second.

`useReducer` returns an array that holds the current state value and a `dispatch` function, to which you can pass an action and later invoke. This is similar to the pattern Redux uses but with a few differences.

For example, the `useReducer` function is tightly coupled to a specific reducer. We dispatch action objects to that reducer only, whereas in Redux, the dispatch function sends the action object to the store. At the time of dispatch, the components don't need to know the reducer that will process the action.

For those who may be unfamiliar with Redux, we'll explore this concept a bit further. There are three main building blocks in Redux:

- A store — an immutable object that holds the applications state data
- A reducer — a function that returns some state data, triggered by an action `type`
- An action — an object that tells the reducer how to change the state. It must contain a `type` property, and it can contain an optional `payload` property

Let's see how these building blocks compare to managing state with the `useReducer` Hook. Here's an example of what a store looks like in Redux:

```
import { createStore } from 'redux'

const store = createStore(reducer, [preloadedState], [enhancer])
```

And here's how to initialize state with `useReducer`:

```
const initialState = { count: 0 }

const [state, dispatch] = useReducer(reducer, initialState)
```

The reducer function in Redux will accept the previous app state and the action being dispatched, calculate the next state, and return the new object.

Reducers in Redux follow this syntax:

```
(state = initialState, action) => newState
```

Consider the following example:

```
// notice that the state = initialState and returns a new state

const reducer = (state = initialState, action) => {
   switch (action.type) {
      case 'ITEMS_REQUEST':
         return Object.assign({}, state, {
            isLoading: action.payload.isLoading
         })
      case 'ITEMS_REQUEST_SUCCESS':
         return Object.assign({}, state, {
            items: state.items.concat(action.items),
            isLoading: action.isLoading
         })
      default:
         return state;
   }
}
export default reducer;
```

React doesn't use the `(state = initialState, action) => newState` Redux pattern, so the reducer function works a bit differently. Here's how you'd create reducers with React:

```
function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    default:
      throw new Error();
  }
}
```

Now here's an example of an action that can be carried out in Redux:

```
{ type: ITEMS_REQUEST_SUCCESS, payload: { isLoading: false } }

// action creators
export function itemsRequestSuccess(bool) {
   return {
      type: ITEMS_REQUEST_SUCCESS,
      payload: {
      isLoading: bool,
    }
   }
}

// dispatching an action with Redux
dispatch(itemsRequestSuccess(false))    // to invoke a dispatch
function, you need to pass action as an argument to the dispatch
function
```

```
// not the complete code
switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    default:
      throw new Error();
  }

// dispatching an action with useReducer
 <button onClick={() => dispatch({type:
'increment'})}>Increment</button>
```

If the action type in the code above is `increment`, our state object is increased by 1.

The reducer function

The `reduce()` method in JavaScript executes a reducer function on each element of the array an and then returns a single value. The `reduce()` method accepts a reducer function, which itself can accept up to four arguments. Here is a code snippet to illustrate how a reducer works:

```
const reducer = (accumulator, currentValue) => accumulator +
currentValue;
[2, 4, 6, 8].reduce(reducer)
// expected output: 20
```

This is essentially what happens with `useReducer` in React: it accepts a reducer function that returns a single value.

The reducer function itself accepts two parameters and returns one value. The first parameter is the current state, and the second is the action. The state is the data we are manipulating. The reducer function receives an action, which is executed by a `dispatch` function.

```
function reducer(state, action) { }

dispatch({ type: 'increment' })
```

The action is like an instruction you pass to the reducer function. Based on the action specified, the reducer function execute the necessary state update. If you have used a state management library like Redux then you must have come across this pattern of state management.

Specifying the initial state

The initial state is the second argument passed to the `useReducer` Hook, and it represents the default state.

```
const initialState = { count: 1 }

// wherever our useReducer is located
const [state, dispatch] = useReducer(reducer, initialState, initFunc)
```

Note that if you don't pass a third argument to `useReducer`, it will take the second argument as the initial state. The third argument, which is the `init` function, is optional.

This pattern also follows one of the golden rules of Redux state management: the state should be updated by emitting actions. Never write directly to the state.

It's worth noting, however, that the Redux `state = initialState` convention doesn't work the same way with `useReducer`. This is because the initial value sometimes depends on props.

Creating the initial state lazily

In programming, lazy initialization is the tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed.

As mentioned above, `useReducer` can accept a third parameter, which is an optional `init` function for creating the initial state lazily. It lets you extract logic for calculating the initial state outside of the reducer function, as you can see below:

```
const initFunc = (initialCount) => {
    if (initialCount !== 0) {
        initialCount=+0
    }
  return {count: initialCount};
}

// wherever our useReducer is located
const [state, dispatch] = useReducer(reducer, initialCount, initFunc);
```

The `initFunc` above will reset the `initialCount` to `0` on page mount if the value is not `0` already, and then return the state object. Notice that this `initFunc` is a function, not just an array or object.

The `dispatch` method

The action to be executed is specified in our reducer function, which in turn is passed to the `useReducer`. The reducer function will then return the updated state.

The actions that will be dispatched by our components should always be represented as one object with the `type` and `payload` key, where `type` stands as the identifier of the dispatched action and `payload` is the piece of information that this action will add to the state.

The `dispatch` is the second value returned from the `useReducer` Hook and can be used in our JSX to update the state.

```
// creating our reducer function
function reducer(state, action) {
  switch (action.type) {
   // ...
      case 'reset':
          return { count: action.payload };
    default:
      throw new Error();
  }
}

// wherever our useReducer is located
const [state, dispatch] = useReducer(reducer, initialCount, initFunc);

// Updating the state with the dispatch functon on button click
<button onClick={() => dispatch({type: 'reset', payload:
initialCount})}> Reset </button>
```

Notice how our reducer function makes use of the payload that is passed from the `dispatch` function. It sets our state object to the payload, i.e., whatever the

Of particular note is the fact that we can pass the `dispatch` function to other components through props. This simple fact alone is what allows us to replace Redux with `useReducer`.

Let's say we have a component we want to pass our dispatch function to as props. We can easily do that like this from the parent component:

```
<Increment count={state.count} handleIncrement={() => dispatch({type:
'increment'})}/>
```

Now, in the child component, we receive the props, which, when emitted, will trigger the dispatch function and update the state:

```
<button onClick={handleIncrement}>Increment</button>
```

Bailing out of a dispatch

If the `useReducer` Hook returns the same value as the current state, React will bail out without rendering the children or firing effects. This is because it uses the `Object.is` comparison algorithm.

Building a simple counter app with `useReducer`

Now let's put our knowledge to work by building a simple counter app with `useReducer`:

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

const initialState = { count: 0 }
 // The reducer function
function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 }
    case 'decrement':
      return { count: state.count - 1 }
    case 'reset':
      return {count: state.count = 0}
    default:
     return { count: state.count  }
  }
}

const Counter = () => {
  const [state, dispatch] = useReducer(reducer, initialState)
```

First, we initialize the state with `0`, then we create a reducer function that accepts the current state of our count as an argument and an action. The state is updated by the reducer based on the action type. `increment`, `decrement`, and `reset` are all action types that, when dispatched, update the state of our app accordingly.

So, to increment the state count `const initialState = { count: 0 }`, we simply set `count` to be `state.count + 1` when the `increment` action type is dispatched.

`useState` vs. `useReducer`

you could use `useReducer` for everything you can do with `useState`.

However, there are some major differences between these two Hooks. `useReducer` lets you avoid passing down callbacks through different levels of your component, instead allowing you to pass a provided `dispatch` function, which in turn will improve performance for components that trigger deep updates.

This does not imply that the `useState` updater function is newly called on each render. What it means is that when you have a complex logic to update state, you simply won't use the setter directly to update state; instead, you will write a complex function, which in turn would call the setter with updated state.

Therefore, it is recommended to use `useReducer`, which returns a `dispatch` method that doesn't change between re-renders, and you can have the manipulation logic in the reducers.

It is also worth noting that with `useState`, the state updater function is invoked to update state, but with `useReducer`, the `dispatch` function is invoked instead, and an action with at least a type is passed to it.

Let's take a look into how both Hooks are declared and used:

Declaring state with `useState`

```
const [state, setState] = useState('default state');
```

`useState` returns an array that holds the current state value and a `setState` method for updating the state.

`useReducer` returns an array that holds the current state value and a `dispatch` method that logically achieve the same goal of `setState`, i.e., updating the state.

Updating state with `useState`

```
<input type='text' value={state} onChange={(e) =>
setState(e.currentTarget.value)} />
```

Updating state with `useReducer`

```
<button onClick={() => dispatch({ type:
'decrement'})}>Decrement</button>
```

We'll discuss the `dispatch` function in greater depth a bit later. Optionally, an action object may also have a `payload`:

```
<button onClick={() => dispatch({ type: 'decrement', payload:
0})}>Decrement</button>
```

`useReducer` can be handy when managing complex state shape. For example, when the state consists of more than primitive values, like nested arrays or objects:

```
const [state, dispatch] = useReducer(loginReducer,
  {
    users: [
      { username: 'Philip', isOnline: false},
      { username: 'Mark', isOnline: false },
      { username: 'Tope', isOnline: true},
      { username: 'Anita', isOnline: false },
    ],
    loading: false,
    error: false,
  },
);
```

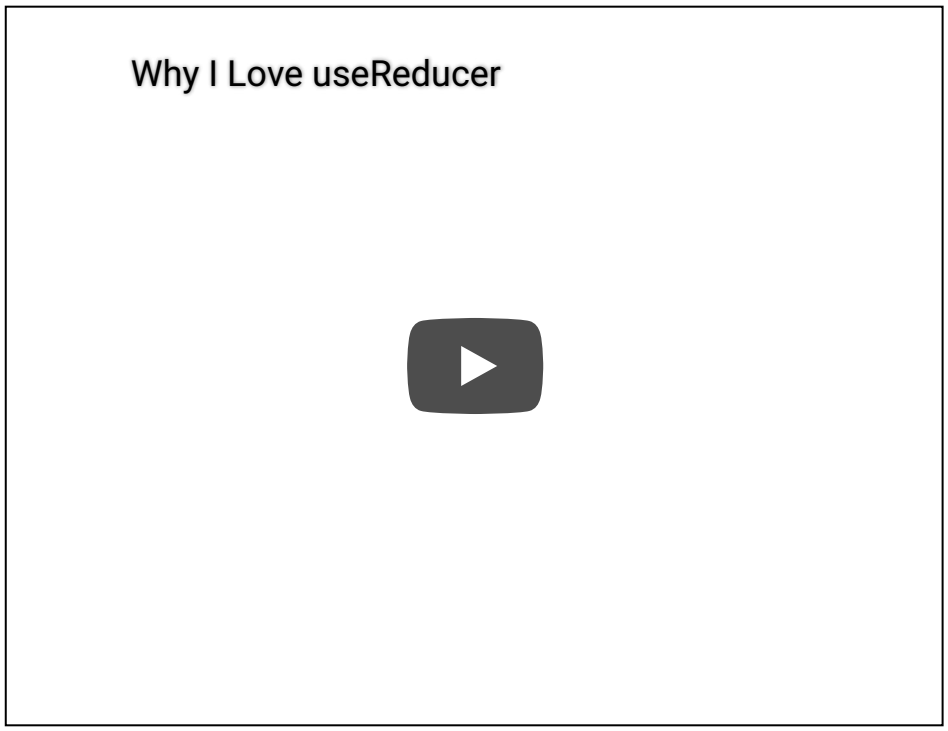
It is easier to manage this local state because the parameters depend on each other, and all the logic could be encapsulated into one reducer.

When to use the `useReducer` Hook

Once your application grows in size, you'll most likely deal with more complex state transitions. At this point, you will be better off using `useReducer` as it gives us more predictable state transitions than `useState`. This becomes more important when state changes become so complex that you want to have one place (i.e., the render function) to manage state.

A good rule of thumb is that when you move past managing primitive data (i.e., a string, integer, or Boolean) and instead must manage a complex object (e.g., with arrays and additional primitives), you're likely better off using `useReducer`.

If you're more of a visual learner, the video below gives a detailed explanation with practical examples of when to use the `useReducer` Hook.



Now to continue on our discussion, let's create a login component and compare how we'd manage state with both the `useState` and `useReducer` Hooks to get a better understating of when to use `useReducer` .

First, the login component with `useState` :

```
              value={username}
              onChange={(e) => setUsername(e.currentTarget.value)}
            />
            <input
              type='password'
              placeholder='password'
              autoComplete='new-password'
              value={password}
              onChange={(e) => setPassword(e.currentTarget.value)}
            />
            <button className='submit' type='submit' disabled=
{isLoading}>
              {isLoading ? 'Logging in...' : 'Log In'}
            </button>
          </form>
        )}
      </div>
    </div>
  );
}
```

Notice how we are dealing with all these state transitions (`username` , `password` , `isLoading` , `error` , `isLoggedIn`) when we really should be more focused on the action that the user wants to take on the login component.

We made use of five `useState` Hooks, and we had to worry about when each of these states is transitioned. We can refractor the code above to use `useReducer` and encapsulate all our logic and state transitions in one reducer function:

```
              autoComplete="new-password"
              value={password}
              onChange={(e) =>
                dispatch({
                  type: 'field',
                  fieldName: 'password',
                  payload: e.currentTarget.value,
                })
              }
            />
            <button className='submit' type='submit' disabled=
{isLoading}>
              {isLoading ? 'Logging in...' : 'Log In'}
            </button>
          </form>
        )}
      </div>
    </div>
  );
}
```

Notice how the new implementation with `useReducer` has made us more focused on the action the user is going to take. For example, when the `login` action is dispatched, we can see clearly what we want to happen. We want to return a copy of our current state, set our `error` to empty string and set `isLoading` to true:

```
case 'login': {
      return {
        ...state,
        error: '',
        isLoading: true,
      };
    }
```

Now here is the beautiful thing about this current implementation: we no longer have to focus on state transition, and instead, we are keen on the actions to be executed by the user.

When not to use the `useReducer` Hook

Despite being able to use the `useReducer` Hook to handle complex state logic in our app, it is important to note that there are certainly some scenarios in which a third-party state management library like Redux may be a better option. But how do you know if React Hooks won't cut out for what you're building?

One simple answer to this question is that you should avoid using Redux or any other third-party state management library until you have problems with vanilla React. If you're still confused as to whether you need it, chances are, you likely don't. Here are some specific cases where it makes more sense to use a library like Redux or MobX:

When your application needs a single source of truth

Centralizing your application's state and logic with a library like Redux make creating universal application state a breeze since the state from the server can easily be serialized to the client app. Having a single source of truth also enables powerful capabilities like undo/redo features a breeze to implement.

When you want a more predictable state

Using a library like Redux helps you write applications that behave consistently when running in different environments. If the same state and action are passed to a reducer, the same result is always produced because reducers are pure functions. Also, state in Redux is read-only, and the only way to change the state is to emit an action, an object describing what happened.

Using a library like Redux is best suited when keeping everything in a top-level React component's state are no longer sufficient.

State persistence

With libraries like Redux and MobX, state can easily be saved to `localStorage` and made available to end users even after page refresh

With all these benefits, it is also worth noting that using a library like Redux as opposed to pure React with `useReducer` comes with some tradeoffs. Redux has a hefty learning curve, and it's definitely not the fastest way to write code. Rather, it is intended to give you an absolute and predictable way of managing state in your app.

Conclusion

We have just completed our examination of the `useReducer` Hook, which is used for managing complex state logic in our React functional component. We also looked at some use cases where `useReducer` is a great fit for managing state logic.

In building a React app, it is important to note that no single Hook can solve all our problems. It's all about understanding where each of the Hooks are best suited, using them accordingly, and combining different Hooks that we get to manage state in our app in a predictable fashion.

As we've also discussed, the `useReducer` Hook may not always be the best option. It is most important to use the method that best aligns with the size and architecture of your application.

Ejiro Asiuwhu Follow

Software developer with industry experience in building scalable and performant applications that run on the web and smartphones with cutting-edge technology.

#react

ArizonaMangoJuice Says:
March 3, 2021 at 4:39 pm

Reply

Very Helpful Article. It goes well in depth about useReducer and differences between useState. Thanks!

Ajay Kumar Says:
May 30, 2021 at 7:15 am

Reply

Amazing explanation about useReducer. Thanks you so much!!!

Ejiro Asiuwhu Says:
June 1, 2021 at 9:34 am

Reply

I'm glad you find it useful

DINESH JAI Says:
June 24, 2021 at 10:16 pm

Reply

Amazing effort to put a in depth explanation with examples. Thank You!!

Rakesh Ranjan Says:
July 31, 2021 at 12:55 pm

Reply

Well explained. Thank you..

Krom Whisperer Says:
August 13, 2021 at 12:34 pm

Reply

I am currently stuck. I am using useReducer because I have form with multiple inputs. I capture the inputs and call an API. I am getting the results back.

The part I am stuck on is how to get that data into a table of sorts. But before I can do that I need to update the state with the data I get back (I think). I am having a hard time finding how to do that. Most examples just have everything on one page: the reducer, the axios call, the update to one field because they are using useState, and just display the results in a console log. That's great but who does that? #1) Just call a get API on page load and #2) Just plop the data into the console log. While informative and helps with some understanding, it is this last piece of the puzzle that I am stuck on. I am a little frustrated at the moment. If you know of a site that shows this last piece of the puzzle of if you have an example somewhere please let me know. Thanks for your time and sharing this info for newbs!

Leave a Reply

Enter your comment here...