## Refactoring to 'useReducer'

Let's modify our custom hook to use 'useReducer' instead of 'useState'.

#### WE'LL COVER THE FOLLOWING ^

- Refactoring to useReducer
  - Reducers
- End Result of This Chapter

## Refactoring to useReducer #

The most intuitive way to make the state reducer pattern work is using the useReducer hook to manage the internal state.

First, let's refactor our custom hook to use useReducer instead of useState.

To refactor our custom hook to use the useReducer hook for state updates, everything within the custom hook stays the same except how the expanded state is managed.

Here's what's to be changed.

First, we'll change the state from just a boolean value, true or false, to an object like this: {expanded: true || false}.

```
// useExpanded.js
export default function useExpanded (initialExpanded = false) {
  const initialState = { expanded: initialExpanded }
  ...
}
```

Once we do that, we'll invoke useReducer to manage the state.

```
const [expanded, setExpanded] = useState(initialExpanded)

// now
const [{ expanded }, setExpanded] = useReducer(internalReducer, initialState)
...
```

Note how we need to destructure expanded from the state object in the new implementation on **line** 7. It's worth mentioning that setExpanded returned from the useReducer call now works as a dispatcher. I'll show the implication of this soon.

If you're new to useReducer, it is typically called with a reducer and an initial state value.

```
useReducer(internalReducer, initialState)
```

#### Reducers #

If you've worked with Redux in the past, then you're most likely familiar with the concept of reducers. If not, a reducer is just a function that receives state and action to return a *new state*.

action usually refers to an object, but useReducer isn't strict about this. If we stick to the Redux convention, a state update is always made by "dispatching" an action.

The "dispatcher" is the second value returned by the useReducer call. Also, in order for the reducer to identify each action being dispatched, a type property always exists in the action.

```
// for example
action = {
   type: "AN_ACTION_TYPE"
}
```

I don't want to turn this into a redux guide, but if you need a refresher, I wrote a visual guide that will make the concepts of reducers and actions all sink in.

With the useReducer call in place, we need to actually write the reducer used within the call, i.e., internalReducer

Here's the implementation:

```
// useExpanded.js
                                                                                        6
const internalReducer = (state, action) => {
  switch (action.type) {
    case useExpanded.types.toggleExpand:
     return {
        ...state,
        expanded: !state.expanded //toggle expand state property
    case useExpanded.types.reset:
     return {
        ...state,
        expanded: action.payload // reset expanded with a payload
    default:
      throw new Error(`Action type ${action.type} not handled`)
}
export default function useExpanded (initialExpanded = false) {
const [{ expanded }, setExpanded] = useReducer(internalReducer, initialState)
```

Remember, I said actions typically have a type property. The reducer checks this type property and returns a new state based on the type.

To stay consistent and prevent unwanted typos, the available types for the useExpanded custom hook are centralized within the same file.

```
// useExpanded.js
...
useExpanded.types = {
  toggleExpand: 'EXPAND',
  reset: 'RESET'
}
```

That explains why the types are used this way in the code as shown below:

```
const internalReducer = (state, action) => {
   switch (action.type) {
     case useExpanded.types.toggleExpand: //<>     look here
        return {
                ...state,
                    expanded: !state.expanded
        }
     ...
}
```

Note that the returned value from the reducer represents the new state of the

expanded state. For example, here's the returned value for the type,

useExpanded.types.toggleExpand.

```
return {
     ...state,
     expanded: !state.expanded
    }
...
```

For this to come full circle, consider how the reset function now works.

```
// before
setExpanded(initialExpanded)
// now
...
setExpanded({
  type: useExpanded.types.reset,
  payload: initialExpanded // pass a payload "initialExpanded"
})
```

setExpanded is now a "dispatcher" so it's called with an action object. An action object has a type and a payload. Redux users must be familiar with the payload property which keeps a reference to a value required by this action to make a state update.

Note how the reducer returns the new state based on this dispatched reset action:

```
case useExpanded.types.reset:
    return {
        ...state,
        expanded: action.payload // reset expanded with the payload
    }
...
```

Here, the payload passed in is the initialExpanded variable.

We've made decent progress!

Below is a quick summary of the changes made:

```
// keep initial state in a const variable
// NB: state is now an object e.g {expanded: false}
const initialState = { expanded: initialExpanded }

// the useReducer call for state updates 
const [{ expanded }, setExpanded] = useReducer(internalReducer, initialState)
// \{\bar{\Delta} \text{ Note how we destructured state variable, expanded}

// setExpanded is now a dispatcher. Toggle fn refactored 
const toggle = useCallback(

// Every setExpanded call should be passed an object with a type
() => setExpanded({ type: useExpanded.types.toggleExpand }),

[]
)
...
}
```

The available types for state updates are then centralized to prevent string typos.

```
// useExpanded.js
...
useExpanded.types = {
  toggleExpand: 'EXPAND',
  reset: 'RESET'
}
```

Here's how the reset is performed internally:

```
// before
setExpanded(initialExpanded)
// now
...
setExpanded({
   type: useExpanded.types.reset,
   payload: initialExpanded // pass a payload "initialExpanded"
})
...
```

The result of this is a complete refactor of useReducer — no additional functionality has been added.

Does anything confuse you?

Please have a look one more time. This will only prove difficult if you don't know how useReducer works. If that's the case, I suggest you quickly check that out.

Having completed this refactor, we need to implement the actual state

reducer pattern.

Right now, the custom hook is invoked like this:

```
// user's app
...
const { expanded, toggle, reset, resetDep} = useExpanded(false)
```

Where the value passed to useExpanded is the initial expanded state.

Remember that we need to communicate our internal state changes to the user. The way we'll do this is by accepting a second parameter, the user's own reducer.

For example:

```
// user's app
function App () {
  const { expanded, toggle, reset, resetDep } = useExpanded(
    false,
    appReducer // < the user's reducer
  )
  ...
}</pre>
```

# End Result of This Chapter #

Before we go ahead with the rest of the implementation, let's see what the user will be able to do at the end. Have a look!

```
.Expandable-panel {
    margin: 0;
    padding: 1em 1.5em;
    border: 1px solid hsl(216, 94%, 94%);;
    min-height: 150px;
}
```

Let's learn how this particular user used our custom hook piece-by-piece in the rest of this chapter!