Implementing the State Reducer Pattern

Let's integrate the state reducer pattern into the scenario we studied earlier

WE'LL COVER THE FOLLOWING

Our solution to the hacker's feature request is to implement the state reducer pattern.

When the reader clicks the `Header` element, we'll communicate the changes we propose to be made internally to the hacker, i.e., we'll let the hacker know we want to update the expanded state internally.

The hacker will then check their own application state. If the reader has already viewed the secret resource, the hacker will communicate to our custom hook NOT to allow a state update.

There it is. The power of the state reducer pattern, giving control of the internal state updates to the user.

Implementation

Now, let's write the technical implementation.

First, we'll expect a second argument from the hacker, their own reducer.

```
// Note the second parameter to our hook, useExpanded
function useExpanded (initialExpanded = false, userReducer) {
 ...
}
```

The second parameter to our `useExpanded` hook represents the user's reducer.

After the refactor to `useReducer`, we got the value of the `expanded` state by calling our internal reducer.

```
// Note the use of our "internalReducer"
 const [{ expanded }, setExpanded] = useReducer(internalReducer, initialState)
...
```

Resolving User and Internal Implementation

The problem with this is that our internal reducer always returns our internal proposed new state. This isn't the behavior we want. Before we decide what the `expanded` state will be, we need to communicate our proposed state change to the user, i.e., the hacker.

So, what do we do?

Instead of passing our `internalReducer` to the `useReducer` call, let's pass in another reducer we'll call `resolveChangesReducer`.

The sole purpose of `resolveChangesReducer` is to resolve changes between our internal implementation and what the user suggests.

Every reducer takes in state and action, right?

The implementation of `resolveChangesReducer` begins by receiving the state and action, and holding a reference to our internal change.

```
const resolveChangesReducer = (currentInternalState, action) => {
    // look here 👇
    const internalChanges = internalReducer(currentInternalState, action)
    ...
  }
```

A reducer always returns new state. The value `internalChanges` holds the new state we propose internally by invoking our `internalReducer` with the current state and action.

We need to communicate our proposed changes to the user. The way we do this is by passing this `internalChanges` to the user's reducer, i.e., the second argument to our custom hook.

```
const resolveChangesReducer = (currentInternalState, action) => {
    const internalChanges = internalReducer(currentInternalState, action)
    // look here 👇
    const userChanges = userReducer(currentInternalState, {
      ...action,
      internalChanges
    })
    ...
  }
```

Right now, we invoke the user's reducer with the internal state and an action object.

Note that the action object we send to the user is slightly modified. It contains the action being dispatched and our internal changes!

```
{
 ... action,
 internalChanges: internalChanges
}
```

Every reducer takes in state and action to return a new state. It is the responsibility of the user's reducer to return whatever they deem fit as the new state. The user can either return our changes if they're happy with them OR override whatever changes they need to.

With a reference to the user changes held, the `resolveChangesReducer` now returns the proposed changes from the user NOT ours.

```
// useExpanded.js
...
const resolveChangesReducer = (currentInternalState, action) => {
    const internalChanges = internalReducer(currentInternalState, action)
    const userChanges = userReducer(currentInternalState, {
      ...action,
      internalChanges
    })

   // look here 👇
    return userChanges
  }
...
```

The return value of a reducer depicts new state. `resolveChangesReducer` returns the user's changes as the new state!

That's all there is to do!

Have a look at the complete implementation of the state reducer. I've omitted whatever hasn't changed. You can look in the source yourself at the end of the lesson if you care about the entire source code (and you should!)

```
// necessary imports 👇
import { useCallback, useMemo, useRef, useReducer } from 'react'
...
// our internal reducer 👇
const internalReducer = (state, action) => {
  switch (action.type) {
    case useExpanded.types.toggleExpand:
      return {
        ...state,
        expanded: !state.expanded
      }
    case useExpanded.types.reset:
      return {
        ...state,
        expanded: action.payload
      }
    default:
      throw new Error(`Action type ${action.type} not handled`)
  }
}
// the custom hook 👇
export default function useExpanded (initialExpanded = false, userReducer) {
  const initialState = { expanded: initialExpanded }
  const resolveChangesReducer = (currentInternalState, action) => {
    const internalChanges = internalReducer(
                currentInternalState,
                action
    )
    const userChanges = userReducer(currentInternalState, {
      ...action,
      internalChanges
    })
    return userChanges
  }
  // the useReducer call 👇
  const [{ expanded }, setExpanded] = useReducer(
    resolveChangesReducer,
    initialState
  )
  ...
  // 👇 value returned by the custom hook
  const value = useMemo(
    () => ({
      expanded,
      toggle,
      getTogglerProps,
      reset,
      resetDep: resetRef.current
    }),
    [expanded, toggle, getTogglerProps, reset]
  )
```

```
  return value
}
// 👇 the available action types
useExpanded.types = {
  toggleExpand: 'EXPAND',
  reset: 'RESET'
}
```

Not so hard to understand, huh?

With the pattern now implemented internally, here's how the hacker took advantage to implement their feature.

```
// the user's app 👇
function App () {
  // ref holds boolean value to decide if user has viewed secret or not
  const hasViewedSecret = useRef(false) // 👈 initial value is false
  // hacker calls our custom hook 👇
  const { expanded, toggle, reset, resetDep = 0 } = useExpanded(
    false,
    appReducer // 👈 hacker passes in their reducer
  )

  // The user's reducer
  // Remember that action is populated with our internalChanges
  function appReducer (currentInternalState, action) {
    // dont update "expanded" if reader has viewed secret
    // i.e hasViewedSecret.current === true
    if (hasViewedSecret.current) {
     // object returned represents new state proposed by hacker
      return {
        ...action.internalChanges,
        // override internal update
        expanded: false
      }
    }

    // else, hacker is okay with our internal changes
    return action.internalChanges
  }

  useEffectAfterMount(
    () => {
      // open secret in new tab 👇
      window.open('https://leanpub.com/reintroducing-react', '_blank')
      //after viewing secret, hacker sets the ref value to true.
      hasViewedSecret.current = true
    },
    [resetDep]
  )
 ...
}
```

That's a well-commented code snippet, you agree?

I hope you find it clear enough to understand how a user may take advantage

I hope you find it clear enough to understand how a user may take advantage of the state reducer pattern you implement in your custom hooks/component.

The App So Far

With this implementation, the hacker's request is fulfilled! Here's the app in action.

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

Quick Quiz!

Let's take a quiz.

1

What is the second argument in the `useExpanded` function?

```
function useExpanded (initialExpanded = false, userReducer) {
  ...
}
```

COMPLETED 0%

1 of 2

Now that we understand the meat of this pattern, there are a couple of things that we need to work on to clean the code up. See you in the next lesson!