React Reducer

Reducer

In an React application you might have a lot of state that you're maintaining across multiple event handler and it can get overwhelming. You can consolidate those state update logic outside of the component into a single function called a reducer. The reducer will take care of the corresponding state updating for you, and within your application you just need to invoke it.

In order to really understand Reducer we will need an example application, and in this case we will be exploring a sample todo list to observe the benefit of how utilizing reducer help with consolidating with the management of states.

The todo list app can be sample here here. Read through the codebase and understand how the flow of this simple todo list goes:

- **TaskApp**: This is the main component that starts up the chain. This is where the task data is initialized as state as well as keeping track of the id of the task. They are just hard-coded as global variable data which persist after re-rendering. In real-world example these would probably come from a context that's initialized through a database call
- **AddTask:** Add task is the first component first **TaskApp,** it takes in the function that adds on a new task to the state that's within **TaskApp**. When the user finishes entering the detail of the new task that they would like to add, it calls the event handler of **handleAddTask** which will add a new task, incrementing the id counter and pull in the text detail that was entered in **AddTask**
- **TaskList:** This is the other component that's within **TaskApp** it takes in the list of task object, the handler for both updating the task and deleting the task as properties. It renders out each of the tasks object as **Task** component and pass in the handlers to each of the **Task** component
- **Task:** This is the main component which both displays, edit, and deletes the task from the task state. It declares a state that dictates whether or not text of the task is being edited right now, each update to the input calls the **handleChangeTask** handler which just updates the task to the current one that's being edited right now, notice that the change event handler still calls **setTasks** this is how the state are being kept in sync, if you were to display the task out in the webpage it will be updated as you edit the task. Then for delete it just calls the **handleDeleteTask** which just simply update the task to be lists that's not the deleted ones quite clever!

As you can see there are a lot of event handler passing and calling which is hard to keep track of in a larger application. This is where reducer comes into play. Reducer let's you aggregate all those

event handler functions into one place, and the component that needs to make updates to the state will just access those updates directly without having to rely on the parent component passing them the event handler.

Using Reducer

Going off from the previous example if we want to incorporate in reducer there are couple of things that we need to do.

1. We move the state setting to dispatching actions

Let's just extract out all of the functions that handle the updates to the task state. It would just be these three functions.

```
function handleAddTask(text) {
  setTasks([
    ...tasks,
    {
      id: nextId++,
      text: text,
      done: false,
    },
  ]);
}

function handleChangeTask(task) {
  setTasks(
    tasks.map((t) => {
      if (t.id === task.id) {
        return task;
      } else {
        return t;
      }
    })
  );
}

function handleDeleteTask(taskId) {
  setTasks(tasks.filter((t) => t.id !== taskId));
}
```

If we want to move managing states manually to using reducers, we will need to switch the paradigm to dispatching "actions".

```
function handleAddTask(text) {
  dispatch({
    type: 'added',
    id: nextId++,
    text: text,
  });
}

function handleChangeTask(task) {
  dispatch({
    type: 'changed',
    task: task,
  });
}

function handleDeleteTask(taskId) {
  dispatch({
    type: 'deleted',
    id: taskId,
  });
}
```

We update the functions to use the `dispatch` function to handle the state updates. Don't worry the `dispatch` function will be defined a bit later. The thing that you pass into the dispatch function is called an action. It is just a regular JavaScript object that contains what what you want to put in. At it's minimum it should contain information about what happened (what actions did the user took?) This is indicate through the "type" field **by convention**, along with any other information that's necessary to make the update to your state.

2. Writing the reducer function

A reducer function will take in the state and the action as the parameter. It will update the state variable which is `tasks` in this case based off the action that was taken by the user, remember `dispatch`?

This is essentially the logic that was in each of the handler function for handling the state management, and you can see it is now converged into this reducer function.

```
function tasksReducer(tasks, action) {
  if (action.type === 'added') {
```

```
    return [
      ...tasks,
      {
        id: action.id,
        text: action.text,
        done: false,
      },
    ];
  } else if (action.type === 'changed') {
    return tasks.map((t) => {
      if (t.id === action.task.id) {
        return action.task;
      } else {
        return t;
      }
    });
  } else if (action.type === 'deleted') {
    return tasks.filter((t) => t.id !== action.id);
  } else {
    throw Error('Unknown action: ' + action.type);
  }
}
```

> The reducer function returns what the "next state" would be after the user has taken the "action". This is where the name of reducer function comes from, comes from the reducer function in many programming languages, the values are processed and carried on to the next value to be processed and it boils down to one value at the end.

3. Using the reducer function in your component

Finally, to use the reducer you would want to switch up how you initialize your state, rather than going through `useState` you would want to go through `useReducer`.

```
const [tasks, dispatch] = useReducer(tasksReducer, initialTasks);
```

Just like `useState` , `useReducer` returns two values that you can unpack, the first would be the `tasks` state value that you can pass around and extract information and display in the HTML just like a normal state. The second value would be the dispatcher function that you have written. You would be just be calling the `dispatch` function in the place that you would want to make update to the state and passing in the action values.

```
dispatch({type: 'added', id: 5, text: "hello world"})
```

This would invoke the reducer function with the `added` if statement block.

> Notice that the `action` parameter is the object that comes from what you passed in from `dispatch`. The first parameter `tasks` is provided to you automatically by React. `dispatch` only has to worry about the action parameter and not the state value.

Now rerunning the React application it should function the exact same way without any change in functionality. The benefit is obviously easier to manage the logic for updating the state because all of the logic is within the reducer function.

Revision #3
Created 2025-11-22 05:04:19 UTC by Tamarine
Updated 2025-11-23 04:20:59 UTC by Tamarine