React Lecture - 14

useReducer Hook

Introduction

Components with many state updates spread across many event handlers can get overwhelming.

For these cases, you can consolidate all the state update logic outside your component in a single function, called a reducer.

What a reducer function is ?

As your components grow in complexity, it can get harder to see at a glance all the different ways in which a component's state gets updated.

Ex: Todo List Application: The state in this application will have to go through the following operations.

1. add
2. delete
3. complete

What a reducer function is ?

Reducers are a different way to handle state updates. You can migrate from useState to useReducer in three steps:

1. Move from setting state to dispatching actions.
2. Write a reducer function.
3. Use the reducer from your component.

Step 1: Move from setting state to dispatching actions

We need to move all the state setter logic to a reducer function.

A reducer function is where you will put your state logic. It takes two arguments, the current state and the action object, and it returns the next state:

```
function yourReducer(state, action) {
  // return next state for React to set
}
```

React will set the state to what you return from the reducer.

Step 1: Move from setting state to dispatching actions

```
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);
  }
}
```

Step 2: Use the reducer from your component

Import useReducer in your component

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

Then replace your useState with useReducer

```
const [tasks, setTasks] = useState(initialTasks);
```

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

Step 2: Use the reducer from your component

The useReducer Hook is similar to useState—you must pass it an initial state and it returns a stateful value and a way to set state (in this case, the dispatch function). But it's a little different.

The useReducer Hook takes two arguments:

1. A reducer function
2. An initial state

And it returns:

1. A stateful value
2. A dispatch function (to "dispatch" user actions to the reducer)

Step 3: Use the dispatch function to set your state.

Now we can call the dispatch function.

The dispatch function take one argument which is the action object.

The action object will contain a property of type which will tell what function it needs to perform.

For example we are telling the dispatch function to call the delete action in the reducer function.

We also pass the required information to perform the action. Here we have passed to id to filter out the task.

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

Step 3: Use the dispatch function to set your state.

An action object can have any shape.

By convention, it is common to give it a string type that describes what happened, and pass any additional information in other fields. The type is specific to a component, so in this example either 'added' or 'added_task' would be fine. Choose a name that says what happened!

Summary

- To convert from useState to useReducer:
  - Dispatch actions from event handlers.
  - Write a reducer function that returns the next state for a given state and action.
  - Replace useState with useReducer.
- Reducers require you to write a bit more code, but they help with debugging and testing.
- Reducers must be pure.
- Each action describes a single user interaction.