Extracting State Logic into a Reducer

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.”

You will learn

- What a reducer function is
- How to refactor `useState` to `useReducer`
- When to use a reducer
- How to write one well

Consolidate state logic with a reducer

As your components grow in complexity, it can get harder to see all the different ways that a component’s state gets updated at a glance. For example, the `TaskBoard` component below holds an array of `tasks` in state and uses three different event handlers to add, remove, and edit tasks:

App.js Reset Fork

```
import { useState } from 'react';
import AddTask from './AddTask.js';
import TaskList from './TaskList.js';


```

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


  function handleChangeTask(task) {
```

Show more

Prague itinerary

Add task Add

Visit Kafka Museum Edit Delete

Watch a puppet show Edit Delete

Lennon Wall pic Edit Delete

Each of its event handlers calls `setTasks` in order to update the state. As this component grows, so does the amount of state logic sprinkled throughout it. To reduce this complexity and keep all your logic in one easy-to-access place, you can move that state logic into a single function outside your component, **called a "reducer."**

Reducers are a different way to handle state. 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

Your event handlers currently specify *what to do* by setting state:

```
      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)
  );
}
```

Remove all the state setting logic. What you are left with are three event handlers:

- `handleAddTask(text)` is called when the user presses “Add”.
- `handleChangeTask(task)` is called when the user toggles a task or presses “Save”.
- `handleDeleteTask(taskId)` is called when the user presses “Delete”.

Managing state with reducers is slightly different from directly setting state. Instead of telling React “what to do” by setting state, you specify “what the user just did” by dispatching “actions” from your event handlers. (The state update logic will live elsewhere!) So instead of “setting `tasks`” via event handler, you’re dispatching an “added/removed/deleted a task” action. This is more descriptive of the user’s intent.

```
function handleAddTask(text) {
  dispatch({
```

```
  });
}

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

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

The object you pass to `dispatch` is called an "action:"

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

It is a regular JavaScript object. You decide what to put in it, but generally it should contain the minimal information about *what happened*. (You will add the `dispatch` function itself in a later step.)

Conventions

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!

```
dispatch({
  // specific to component
  type: 'what_happened',
  // other fields go here
});
```

Step 2: Write 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.

To move your state setting logic from your event handlers to a reducer function in this example, you will:

1. Declare the current state (`tasks`) as the first argument.
2. Declare the `action` object as the second argument.
3. Return the *next* state from the reducer (which React will set the state to).

Here is all the state setting logic migrated to a reducer function:

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

```
      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);
  }
}
```

Because the reducer function takes state (`tasks`) as an argument, you can **declare it outside of your component.** This decreases the indentation level and can make your code easier to read.

Conventions

The code above uses if/else statements, but it's a convention to use switch statements inside reducers. The result is the same, but it can be easier to read switch statements at a glance. We'll be using them throughout the rest of this documentation like so:

```
function tasksReducer(tasks, action) {
  switch (action.type) {
    case 'added': {
      return [...tasks, {
```

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

We recommend to wrap each `case` block into the `{` and `}` curly braces so that variables declared inside of different `case` s don't clash with each other. Also, a `case` should usually end with a `return` . If you forget to `return` , the code will "fall through" to the next `case` , which can lead to mistakes!

If you're not yet comfortable with switch statements, using if/else is completely fine.

DEEP DIVE

Why are reducers called this way?

Show Details

Finally, you need to hook up the `tasksReducer` to your component. Make sure to import the `useReducer` Hook from React:

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

Then you can replace `useState`:

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

with `useReducer` like so:

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

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)

Now it's fully wired up! Here, the reducer is declared at the bottom of the component file:

```
import AddTask from './AddTask.js';
import TaskList from './TaskList.js';

export default function TaskBoard() {
  const [tasks, dispatch] = useReducer(
    tasksReducer,
    initialTasks
  );

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

Show more

If you want, you can even move the reducer to a different file:

App.js tasksReducer.js Reset Fork

```
import { useReducer } from 'react';
import AddTask from './AddTask.js';
import TaskList from './TaskList.js';
import tasksReducer from './tasksReducer.js';

export default function TaskBoard() {
  const [tasks, dispatch] = useReducer(
    tasksReducer,
    initialTasks
  );

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

Show more

event handlers only specify *what happened* by dispatching actions, and the reducer function determines *how the state updates* in response to them.

Comparing useState and useReducer

Reducers are not without downsides! Here's a few ways you can compare them:

- **Code size:** Generally, with `useState` you have to write less code upfront. With `useReducer`, you have to write both a reducer function *and* dispatch actions. However, `useReducer` can help cut down on the code if many event handlers modify state in a similar way.
- **Readability:** `useState` is very easy to read when the state updates are simple. When they get more complex, they can bloat your component's code and make it difficult to scan. In this case, `useReducer` lets you cleanly separate the *how* of update logic from the *what happened* of event handlers.
- **Debugging:** When you have a bug with `useState`, it can be difficult to tell *where* the state was set incorrectly, and *why*. With `useReducer`, you can add a console log into your reducer to see every state update, and *why* it happened (due to which `action`). If each `action` is correct, you'll know that the mistake is in the reducer logic itself. However, you have to step through more code than with `useState`.
- **Testing:** A reducer is a pure function that doesn't depend on your component. This means that you can export and test it separately in isolation. While generally it's best to test components in a more realistic environment, for complex state update logic it can be useful to assert that your reducer returns a particular state for a particular initial state and action.
- **Personal preference:** Some people like reducers, others don't. That's okay. It's a matter of preference. You can always convert between `useState` and `useReducer` back and forth: they are equivalent!

We recommend using a reducer if you often encounter bugs due to incorrect state updates in some component, and want to introduce more structure to its code. You don't have to use reducers for everything: feel free to mix and match! You can even `useState` and `useReducer` in the same component.

Keep these two tips in mind when writing reducers:

- **Reducers must be pure.** Similar to state updater functions, reducers run during rendering! (Actions are queued until the next render.) This means that reducers must be pure—same inputs always result in the same output. They should not send requests, schedule timeouts, or perform any side effects (operations that impact things outside the component). They should update objects and arrays without mutations.
- **Actions describe "what happened," not "what to do."** For example, if a user presses "Reset" on a form with five fields managed by a reducer, it makes more sense to dispatch one `reset_form` action rather than five separate `set_field` actions. If you log every action in a reducer, that log should be clear enough for you to reconstruct what interactions or responses happened in what order. This helps with debugging!

Writing concise reducers with Immer

Just like with updating objects and arrays in regular state, you can use the Immer library to make reducers more concise. Here, `useImmerReducer` lets you mutate the state with `push` or `arr[i] =` assignment:

App.js package.json Reset Fork

```
import { useImmerReducer } from 'use-immer';
import AddTask from './AddTask.js';
import TaskList from './TaskList.js';

function tasksReducer(draft, action) {
  switch (action.type) {
    case 'added': {
      draft.push({
        id: action.id,
        text: action.text,
        done: false
      });
      break;
```

Show more

Reducers must be pure, so they shouldn't mutate state. But Immer provides you with a special `draft` object which is safe to mutate. Under the hood, Immer will create a copy of your state with the changes you made to the `draft`. This is why reducers managed by `useImmerReducer` can mutate their first argument and don't need to return state.

Recap

- To convert from `useState` to `useReducer`:
  1. Dispatch actions from event handlers.
  2. Write a reducer function that returns the next state for a given state and action.
  3. Replace `useState` with `useReducer`.
- Reducers require you to write a bit more code, but they help with debugging and testing.
- Reducers must be pure.
- Actions describe "what happened," not "what to do."
- Use Immer if you want to write reducers in a mutating style.

Try out some challenges

1. Dispatch actions from event handlers | 2. Clear the input on sending a message | 3.

Challenge 1 of 4:
Dispatch actions from event handlers

Currently, the event handlers in `ContactList.js` and `Chat.js` have `// TODO` comments. This is why typing into the input doesn't work, and clicking on the buttons doesn't change the selected recipient.

`messengerReducer.js`. The reducer is already written so you won't need to change it. You only need to dispatch the actions in `ContactList.js` and `Chat.js`.

App.js messengerReducer.js ContactList.js Chat.js Reset Fork

```
import { useReducer } from 'react';
import Chat from './Chat.js';
import ContactList from './ContactList.js';
import {
  initialState,
  messengerReducer
} from './messengerReducer';

export default function Messenger() {
  const [state, dispatch] = useReducer(
    messengerReducer,
    initialState
  );
  const message = state.message;
  const contact = contacts.find(c =>
    c.id === state.selectedId
```

Show more

Show hint Show solution Next Challenge

PREVIOUS
Preserving and Resetting State

NEXT
Passing Data Deeply with Context

FACEBOOK

Open Source

©2021

Learn React

Quick Start

Installation

Describing the UI

Adding Interactivity

Managing State

Escape Hatches

API Reference

React APIs

React DOM APIs

Community

Code of Conduct

Acknowledgements

Meet the Team

More

React Native

Privacy

Terms