Scaling Up with Reducer and Context

Reducers let you consolidate a component's state update logic. Context lets you pass information deep down to other components. You can combine reducers and context together to manage state of a complex screen.

You will learn

- How to combine a reducer with context
- How to avoid passing state and dispatch through props
- How to keep context and state logic in a separate file

Combining a reducer with context

In this example from the introduction to reducers, the state is managed by a reducer. The reducer function contains all of the state update logic and is declared at the bottom of this file:

App.js AddTask.js TaskList.js Reset

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

```
  return (
    <>
      <h1>Day off in Kyoto</h1>
      <AddTask
        onAddTask={handleAddTask}
      />
      <TaskList
        tasks={tasks}
        onChangeTask={handleChangeTask}
        onDeleteTask={handleDeleteTask}
      />
    </>
  );
}

```

```
function tasksReducer(tasks, action) {
  switch (action.type) {
    case 'added': {
      return [...tasks, {
        id: action.id,
        text: action.text,
        done: false
      }];
    }
    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);
    }
  }
}

let nextId = 3;
const initialTasks = [
  { id: 0, text: 'Philosopher’s Path', done: true },
  { id: 1, text: 'Visit the temple', done: false },
  { id: 2, text: 'Drink matcha', done: false }
];

```

Show less

A reducer helps keep the event handlers short and concise. However, as your app grows, you might run into another difficulty. **Currently, the `tasks` state and the `dispatch` function are only available in the top-level `TaskApp` component.** To let other components read the list of tasks or change it, you have to explicitly pass down the current state and the event handlers that change it as props.

For example, `TaskApp` passes a list of tasks and the event handlers to `TaskList`:

```
<TaskList
  tasks={tasks}
  onChangeTask={handleChangeTask}
  onDeleteTask={handleDeleteTask}
/>
```

And `TaskList` passes the event handlers to `Task`:

```
<Task
  task={task}
  onChange={onChangeTask}
```

```
      onDelete={onDeleteTask}
    />
```

In a small example like this, this works well, but if you have tens or hundreds of components in the middle, passing down all state and functions can be quite frustrating!

This is why, as an alternative to passing them through props, you might want to put both the `tasks` state and the `dispatch` function into context. **This way, any component below `TaskApp` in the tree can read the tasks and dispatch actions without the repetitive "prop drilling".**

Here is how you can combine a reducer with context:

1. **Create** the context.
2. **Put** state and dispatch into context.
3. **Use** context anywhere in the tree.

Step 1: Create the context

The `useReducer` Hook returns the current `tasks` and the `dispatch` function that lets you update them:

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

To pass them down the tree, you will create two separate contexts:

- `TasksContext` provides the current list of tasks.
- `TasksDispatchContext` provides the function that lets components dispatch actions.

Export them from a separate file so that you can later import them from other files:

App.js TasksContext.js AddTask.js TaskList.js Reset

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

export const TasksContext = createContext(null);
export const TasksDispatchContext = createContext(null);

```

Here, you're passing `null` as the default value to both contexts. The actual values will be provided by the `TaskApp` component.

Step 2: Put state and dispatch into context

Now you can import both contexts in your `TaskApp` component. Take the

`tasks` and `dispatch` returned by `useReducer()` and provide them to the entire tree below:

```
import { TasksContext, TasksDispatchContext } from './TasksContext.js';

export default function TaskApp() {
  const [tasks, dispatch] = useReducer(tasksReducer, initialTasks);
  // ...
  return (
    <TasksContext.Provider value={tasks}>
      <TasksDispatchContext.Provider value={dispatch}>
        ...
      </TasksDispatchContext.Provider>
    </TasksContext.Provider>
  );
}
```

For now, you pass the information both via props and in context:

App.js TasksContext.js AddTask.js TaskList.js Reset

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

```
      id: taskId
    });
  }

  return (
    <TasksContext.Provider value={tasks}>
      <TasksDispatchContext.Provider value={dispatch}>
        <h1>Day off in Kyoto</h1>
        <AddTask
          onAddTask={handleAddTask}
        />
        <TaskList
          tasks={tasks}
          onChangeTask={handleChangeTask}
          onDeleteTask={handleDeleteTask}
        />
      </TasksDispatchContext.Provider>
    </TasksContext.Provider>
  );
}

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

```
56          text: action.text,
57          done: false
58        }];
59      }
60      case 'changed': {
61        return tasks.map(t => {
62          if (t.id === action.task.id) {
63            return action.task;
64          } else {
65            return t;
66          }
67        });
68      }
69      case 'deleted': {
70        return tasks.filter(t => t.id !== action.id);
71      }
72      default: {
73        throw Error('Unknown action: ' + action.type);
74      }
75    }
76  }
77
78  let nextId = 3;
79  const initialTasks = [
80    { id: 0, text: 'Philosopher’s Path', done: true },
81    { id: 1, text: 'Visit the temple', done: false },
82    { id: 2, text: 'Drink matcha', done: false }
83  ];
84
```

Show less

In the next step, you will remove prop passing.

Step 3: Use context anywhere in the tree

Now you don't need to pass the list of tasks or the event handlers down the tree:

```
<TasksContext.Provider value={tasks}>
  <TasksDispatchContext.Provider value={dispatch}>
    <h1>Day off in Kyoto</h1>
    <AddTask />
    <TaskList />
  </TasksDispatchContext.Provider>
</TasksContext.Provider>
```

Instead, any component that needs the task list can read it from the `TaskContext`:

```
export default function TaskList() {
  const tasks = useContext(TasksContext);
  // ...
```

To update the task list, any component can read the `dispatch` function from context and call it:

```
export default function AddTask() {
  const [text, setText] = useState('');
  const dispatch = useContext(TasksDispatchContext);
  // ...
  return (
    // ...
    <button onClick={() => {
      setText('');
      dispatch({
        type: 'added',
        id: nextId++,
        text: text,
      });
    }}>Add</button>
    // ...
```

The `TaskApp` component does not pass any event handlers down, and the `TaskList` does not pass any event handlers to the `Task` component either. Each component reads the context that it needs:

App.js TasksContext.js AddTask.js TaskList.js Reset

```
import { useState, useContext } from 'react';
```

```
        <li key={task.id}>
          <Task task={task} />
        </li>
      ))}
    </ul>
  );
```

```
  );
}

function Task({ task }) {
  const [isEditing, setIsEditing] = useState(false);
  const dispatch = useContext(TasksDispatchContext);
  let taskContent;
  if (isEditing) {
    taskContent = (
      <>
        <input
          value={task.text}
          onChange={e => {
            dispatch({
              type: 'changed',
              task: {
                ...task,
                text: e.target.value
              }
            });
          }} />
        <button onClick={() => setIsEditing(false)}>
          Save
        </button>
      </>
    );
  } else {
    taskContent = (
      <>
        {task.text}
        <button onClick={() => setIsEditing(true)}>
          Edit
        </button>
      </>
    );
  }
  return (
    <label>
      <input
        type="checkbox"
        checked={task.done}
```

```
        checked={task.done}
        onChange={e => {
          dispatch({
            type: 'changed',
            task: {
              ...task,
              done: e.target.checked
            }
          });
        }}
      />
      {taskContent}
      <button onClick={() => {
        dispatch({
          type: 'deleted',
          id: task.id
        });
      }}>
        Delete
      </button>
    </label>
  );
}

```

Show less

The state still "lives" in the top-level `TaskApp` component, managed with `useReducer`. But its `tasks` and `dispatch` are now available to every component below in the tree by importing and using these contexts.

Moving all wiring into a single file

You don't have to do this, but you could further declutter the components by moving both reducer and context into a single file. Currently, `TasksContext.js` contains only two context declarations:

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

export const TasksContext = createContext(null);
export const TasksDispatchContext = createContext(null);
```

This file is about to get crowded! You'll move the reducer into that same file. Then you'll declare a new `TasksProvider` component in the same file. This component will tie all the pieces together:

1. It will manage the state with a reducer.
2. It will provide both contexts to components below.
3. It will take `children` as a prop so you can pass JSX to it.

```
export function TasksProvider({ children }) {
  const [tasks, dispatch] = useReducer(tasksReducer, initialTasks);

  return (
    <TasksContext.Provider value={tasks}>
      <TasksDispatchContext.Provider value={dispatch}>
        {children}
```

```
      </TasksDispatchContext.Provider>
    </TasksContext.Provider>
  );
}
```

This removes all the complexity and wiring from your `TaskApp` component:

App.js TasksContext.js AddTask.js TaskList.js Reset

```
import AddTask from './AddTask.js';
import TaskList from './TaskList.js';
import { TasksProvider } from './TasksContext.js';

export default function TaskApp() {
  return (
    <TasksProvider>
      <h1>Day off in Kyoto</h1>
      <AddTask />
      <TaskList />
    </TasksProvider>
  );
```

You can also export functions that *use* the context from `TasksContext.js`:

```
export function useTasks() {
  return useContext(TasksContext);
}

export function useTasksDispatch() {
  return useContext(TasksDispatchContext);
}
```

When a component needs to read context, it can do it through these functions:

```
const tasks = useTasks();
const dispatch = useTasksDispatch();
```

This doesn't change the behavior in any way, but it lets you later split these contexts further or add some logic to these functions. **Now all of the context and reducer wiring is in `TasksContext.js`. This keeps the components clean and uncluttered, focused on what they display rather than where they get the data:**

App.js TasksContext.js AddTask.js TaskList.js Reset

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

```
          <li key={task.id}>
            <Task task={task} />
          </li>
        ))}
      </ul>
    );
}

function Task({ task }) {
  const [isEditing, setIsEditing] = useState(false);
  const dispatch = useTasksDispatch();
  let taskContent;
  if (isEditing) {
    taskContent = (
      <>
        <input
          value={task.text}
          onChange={e => {
            dispatch({
              type: 'changed',
              task: {
                ...task,
                text: e.target.value
              }
            });
          }} />
        <button onClick={() => setIsEditing(false)}>
          Save
        </button>
      </>
    );
  } else {
    taskContent = (
      <>
        {task.text}
        <button onClick={() => setIsEditing(true)}>
          Edit
        </button>
      </>
    );
  }
```

```
  }
  return (
    <label>
      <input
        type="checkbox"
        checked={task.done}
        onChange={e => {
          dispatch({
            type: 'changed',
            task: {
              ...task,
              done: e.target.checked
            }
          });
        }}
      />
      {taskContent}
      <button onClick={() => {
        dispatch({
          type: 'deleted',
          id: task.id
        });
      }}>
        Delete
      </button>
    </label>
  );
}

```

Show less

You can think of `TasksProvider` as a part of the screen that knows how to deal with tasks, `useTasks` as a way to read them, and `useTasksDispatch` as a way to update them from any component below in the tree.

Note

Functions like `useTasks` and `useTasksDispatch` are called *Custom Hooks.* Your function is considered a custom Hook if its name starts with `use`. This lets you use other Hooks, like `useContext`, inside it.

As your app grows, you may have many context-reducer pairs like this. This is a powerful way to scale your app and lift state up without too much work whenever you want to access the data deep in the tree.

Recap

- You can combine reducer with context to let any component read and update state above it.
- To provide state and the dispatch function to components below:
  1. Create two contexts (for state and for dispatch functions).
  2. Provide both contexts from the component that uses the reducer.
  3. Use either context from components that need to read them.

- You can further declutter the components by moving all wiring into one file.
  - You can export a component like `TasksProvider` that provides context.
  - You can also export custom Hooks like `useTasks` and `useTasksDispatch` to read it.
- You can have many context-reducer pairs like this in your app.

PREVIOUS
Passing Data Deeply with Context

NEXT
Escape Hatches

Learn React

Quick Start

Installation

Describing the UI

API Reference

React APIs

React DOM APIs

Community

More