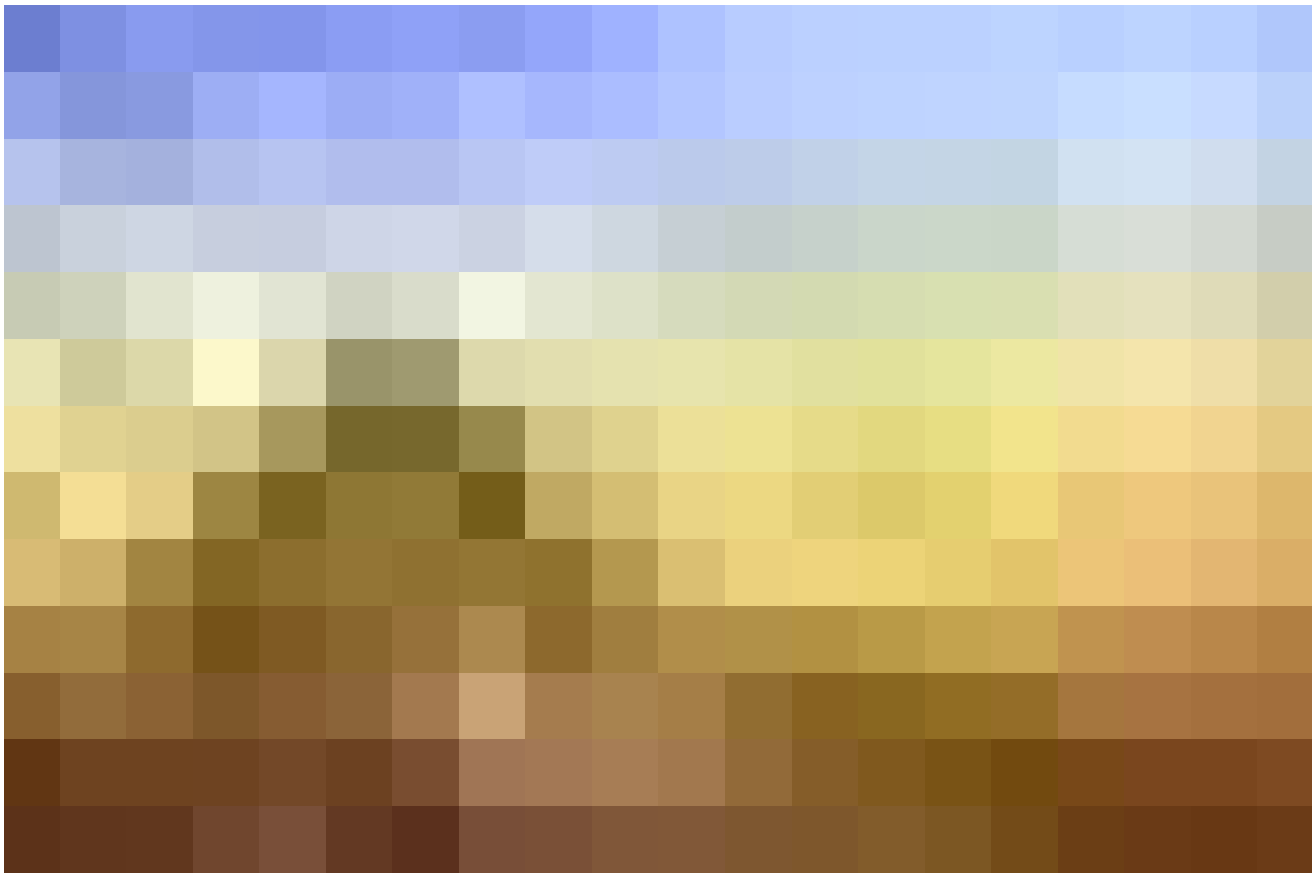
How to create Redux with React Hooks?

MAY 20, 2019 BY ROBIN WIERUCH - EDIT THIS POST

Follow on Twitter Follow on Facebook



This tutorial is part 2 of 2 in this series.

Part 1: React State with Hooks: useReducer, useState, useContext

There are several React Hooks that make state management in React Components possible. Whereas the last tutorial has shown you how to use these hooks -- useState, useReducer, and useContext -- for modern state management in React, this tutorial pushes it to the next level by implementing one global state container with useReducer and useContext.

There are two caveats with useReducer why it cannot be used as **one global state** container: First, every reducer function operates on one independent state. There is not one state container. And second, every dispatch function operates only on one reducer function. There is

no global dispatch function which pushes actions through every reducer. If you are interested about the details, read more about it here: useReducer vs Redux. Keep also in mind that Redux comes with much more than the global state container like the Redux Dev Tools.

GLOBAL DISPATCH WITH REACT HOOKS

So far, we have an application that uses useReducer (and useState) to manage state and React's Context API to pass information such as the dispatch function and state down the component tree. State and state update function (dispatch) could be made available in all components by using useContext.

Since we have two useReducer functions, both dispatch functions are independent. Now we could either pass both dispatch functions down the component tree with React's Context API or implement one global dispatch function which dispatches actions to all reducer functions. It would be one universal dispatch function which calls all the independent dispatch functions given by our useReducer hooks:



```
const App = () => {
  const [filter, dispatchFilter] = useReducer(filterReducer, 'ALL');
  const [todos, dispatchTodos] = useReducer(todoReducer, initialTodos);

  // Global Dispatch Function
  const dispatch = action =>
    [dispatchTodos, dispatchFilter].forEach(fn => fn(action));

  ...
};
```

Now, instead of having a React Context for each dispatch function, let's have one universal context for our new global dispatch function:

```
const DispatchContext = createContext(null);
```

Note: If you continued with the application from the previous tutorial, rename all `TodoContext` simply to `DispatchContext` in the entire application.

In our App component, we merged all dispatch functions from our reducers into one dispatch function and pass it down via our new context provider:

```
const App = () => {
```

```
  const [filter, dispatchFilter] = useReducer(filterReducer, 'ALL');
  const [todos, dispatchTodos] = useReducer(todoReducer, initialTodos);

  // Global Dispatch Function
  const dispatch = action =>
    [dispatchTodos, dispatchFilter].forEach(fn => fn(action));

  const filteredTodos = todos.filter(todo => {
    ...
  });

  return (
    <DispatchContext.Provider value={dispatch}>
      <Filter />
      <TodoList todos={filteredTodos} />
      <AddTodo />
    </DispatchContext.Provider>
  );
};
```

The global dispatch function iterates through all dispatch functions and executes everyone of them by passing the incoming action object to it. Now the dispatch function from the context can be used everywhere the same; in the TodoItem and AddTodo components, but also in the Filter component:

```
const Filter = () => {
  const dispatch = useContext(DispatchContext);

  const handleShowAll = () => {
    dispatch({ type: 'SHOW_ALL' });
  };

  ...
};

const TodoItem = ({ todo }) => {
  const dispatch = useContext(DispatchContext);

  ...
};

const AddTodo = () => {
  const dispatch = useContext(DispatchContext);

  ...
};
```

In the end, we only need to adjust our reducers, so that they don't throw an error anymore in case of an incoming action type that isn't matching one of the cases, because it can happen

that not all reducers are interested in the incoming action now:

```
const filterReducer = (state, action) => {
  switch (action.type) {
    case 'SHOW_ALL':
      return 'ALL';
    case 'SHOW_COMPLETE':
      return 'COMPLETE';
    case 'SHOW_INCOMPLETE':
      return 'INCOMPLETE';
    default:
      return state;
  }
};

const todoReducer = (state, action) => {
  switch (action.type) {
    case 'DO_TODO':
      return state.map(todo => {
        if (todo.id === action.id) {
          return { ...todo, complete: true };
        } else {
          return todo;
        }
      });
    case 'UNDO_TODO':
      return state.map(todo => {
        if (todo.id === action.id) {
          return { ...todo, complete: false };
        } else {
          return todo;
        }
      });
    case 'ADD_TODO':
      return state.concat({
        task: action.task,
        id: action.id,
        complete: false,
      });
    default:
      return state;
  }
};
```

Now all reducers receive the incoming actions when actions are dispatched, but not all care about them. However, the dispatch function is one global function, accessible anywhere via React's context, to alter the state in different reducers. The whole source code can be seen here and all changes here.

GLOBAL STATE WITH REACT HOOKS

Basically we already have all our state from useReducer "globally" accessible, because it is located in our top-level component and *can* be passed down via React's Context API. In order to have *one* global state container (here object) though, we can put all our state coming from the useReducer hooks in one object:

```
const App = () => {
  const [filter, dispatchFilter] = useReducer(filterReducer, 'ALL');
  const [todos, dispatchTodos] = useReducer(todoReducer, initialTodos);

  // Global Dispatch Function
  const dispatch = action =>
    [dispatchTodos, dispatchFilter].forEach(fn => fn(action));

  // Global State
  const state = {
    filter,
    todos,
  };

  ...
};
```

At the moment, all state is passed down via React props. However, now it is up to you to pass it down as one unified state container via React's Context API. The whole source code can be seen here and all changes here.

USECOMBINEDREDUCERS HOOK

The last two sections gave us **one global state** container. Our state is located at our top-level component, can be altered with one dispatch function from anywhere, and comes out as one state. In the last step, we want to hide everything behind one custom React hook called useCombinedReducers:

```
const App = () => {
  const [state, dispatch] = useCombinedReducers({
    filter: useReducer(filterReducer, 'ALL'),
    todos: useReducer(todoReducer, initialTodos),
  });

  ...
```

```
};
```

As before, we want to have access to one global state container (`state`) and one universal dispatch function (`dispatch`). That's what our custom hook returns. As parameters our custom hook receives each returned array from our useReducer calls allocated by object keys. These keys will define our so called substates of our state container so that `const { filter, todos } = state;` will be possible later on. Also note that this custom hook looks very similar to Redux's combineReducers function. Now let's implement the new hook:

```
const useCombinedReducer = combinedReducers => {
  // Global State
  const state =

  // Global Dispatch Function
  const dispatch =

  return [state, dispatch];
};
```

In the previous sections, we already have seen how to create a global state and global dispatch function. However, this time we need to work with a generic object `combinedReducers`.

```
const useCombinedReducer = combinedReducers => {
  // Global State
  const state = Object.keys(combinedReducers).reduce(
    (acc, key) => ({ ...acc, [key]: combinedReducers[key][0] }),
    {}
  );

  // Global Dispatch Function
  const dispatch = action =>
    Object.keys(combinedReducers)
      .map(key => combinedReducers[key][1])
      .forEach(fn => fn(action));

  return [state, dispatch];
};
```

In case of the global state object, we iterate through all values from `combinedReducers` to retrieve from every entry the first item (state) from the array to allocate each by the key given from the outside.

In case of the global dispatch function, we iterate through all values from `combinedReducers` to retrieve from every entry the second item (dispatch function) from the array to call each

dispatch function with the given action from the global dispatch function.

Basically that's it. You have one custom hook which takes in the return values from all your useReducer hooks at a top-level component of your application. The new hook returns the global state object in addition to the global dispatch function. Both can be passed down by React's Context API to be consumed from anywhere in your application. The whole source code can be seen here and all changes here.

You can find the custom hook open sourced over here: useCombinedReducers. If you want to install it, just type `npm install use-combined-reducers` and then import it in your application:

```
import useCombinedReducers from 'use-combined-reducers';
```

. . .

You have seen how multiple useReducer hooks can be used in a custom hook to return one state container and one universal dispatch function. Basically it's **useReducer for global state**. By using React's Context API you can pass state and dispatch function down the component tree to make it everywhere accessible. The shown implementation comes close to Redux's global state container implementation, but it comes with its caveats as explained in the beginning of this tutorial.

KEEP READING ABOUT REACT›

REACT GLOBAL STATE WITHOUT REDUX

The article is a short tutorial on how to achieve global state in React without Redux. Creating a global state in React is one of the first signs that you may need Redux (or another state management...

REACT'S USEREDUCER HOOK VS REDUX

Since React Hooks have been released, function components can use state and side-effects. There are two hooks that are used for modern state management in React (useState and useReducer) and one...