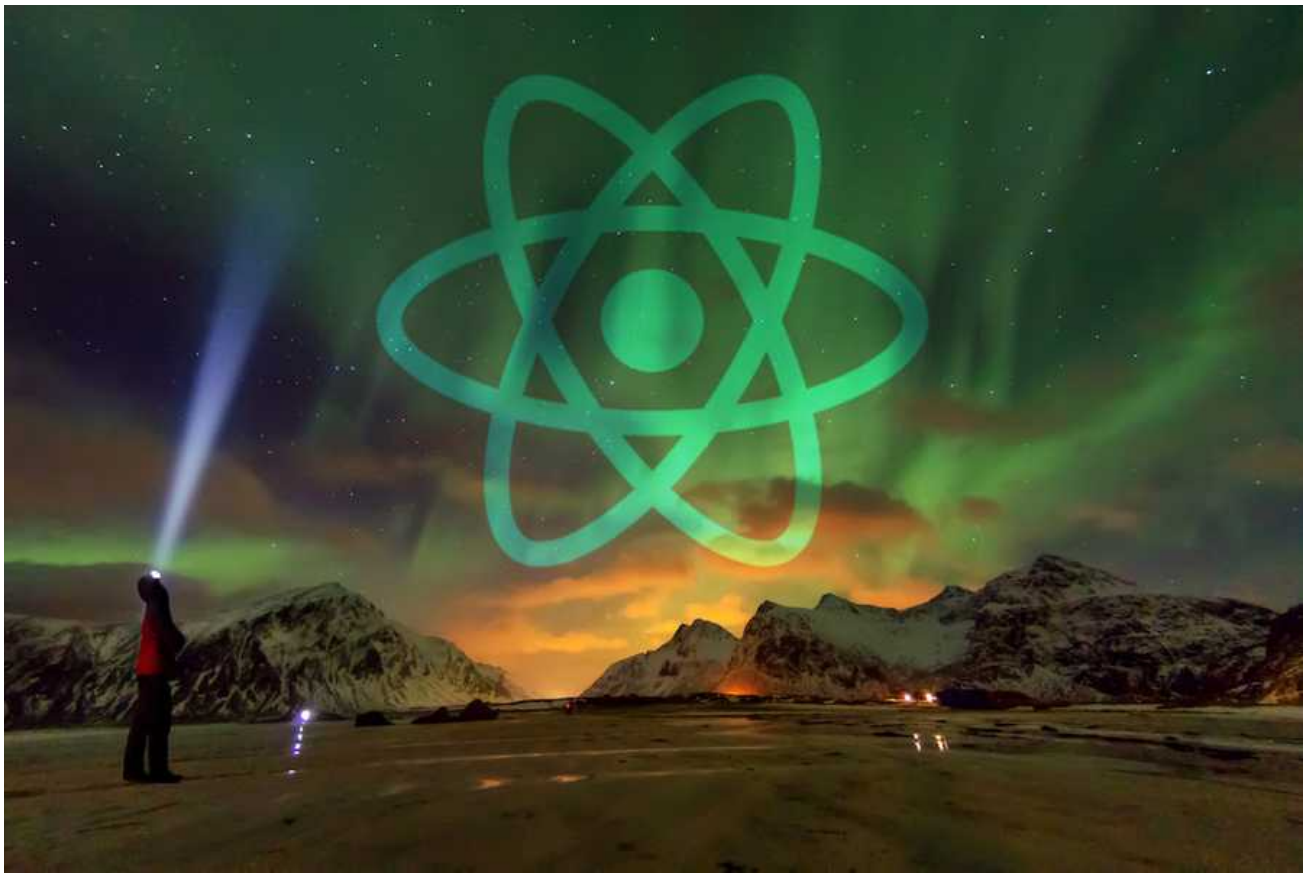
How to useReducer in React

APRIL 28, 2019 BY ROBIN WIERUCH - EDIT THIS POST



This tutorial is part 2 of 2 in this series.

Part 1: What is a reducer in JavaScript?

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. This tutorial goes step by step through a useReducer example in React for getting you started with this React Hook for state management.

REDUCER IN REACT

If you haven't heard about reducers as concept or as implementation in JavaScript, you should read more about them over here: Reducers in JavaScript. This tutorial builds up on this knowledge, so be prepared what's coming. The following function is a reducer function for managing state transitions for a list of items:

```
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;
        }
      });
    default:
      return state;
  }
};
```

There are two types of actions for an equivalent of two state transitions. They are used to toggle the `complete` boolean to true or false of a todo item. As additional payload an identifier is needed which coming from the incoming action's payload.

The state which is managed in this reducer is an array of items:

```
const todos = [
  {
    id: 'a',
    task: 'Learn React',
    complete: false,
  },
  {
    id: 'b',
    task: 'Learn Firebase',
    complete: false,
  },
];
```

In code, the reducer function could be used the following way with an initial state and action:

```
const todos = [
  {
    id: 'a',
    task: 'Learn React',
    complete: false,
  },
  {
    id: 'b',
    task: 'Learn Firebase',
    complete: false,
  },
];

const action = {
  type: 'DO_TODO',
  id: 'a',
};

const newTodos = todoReducer(todos, action);

console.log(newTodos);
// [
//   {
//     id: 'a',
//     task: 'Learn React',
//     complete: true,
//   },
//   {
//     id: 'b',
//     task: 'Learn Firebase',
//     complete: false,
//   },
// ]
```

So far, everything demonstrated here is not related to React. If you have any difficulties to understand the reducer concept, please revisit the referenced tutorial from the beginning for Reducers in JavaScript. Now, let's dive into React's useReducer hook to integrate reducers in React step by step.

REACT'S USEREDUCER HOOK

The useReducer hook is used for complex state and state transitions. It takes a reducer function and an initial state as input and returns the current state and a dispatch function as output with array destructuring:

```
const initialTodos = [
  {
    id: 'a',
    task: 'Learn React',
    complete: false,
  },
  {
    id: 'b',
    task: 'Learn Firebase',
    complete: false,
  },
];

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;
        }
      });
    default:
      return state;
  }
};

const [todos, dispatch] = useReducer(todoReducer, initialTodos);
```

The dispatch function can be used to send an action to the reducer which would implicitly change the current state:

```
const [todos, dispatch] = React.useReducer(todoReducer, initialTodos);

dispatch({ type: 'DO_TODO', id: 'a' });
```

The previous example wouldn't work without being executed in a React component, but it demonstrates how the state can be changed by dispatching an action. Let's see how this

would look like in a React component. We will start with a React component rendering a list of items. Each item has a checkbox as controlled component:

```
import React from 'react';

const initialTodos = [
  {
    id: 'a',
    task: 'Learn React',
    complete: false,
  },
  {
    id: 'b',
    task: 'Learn Firebase',
    complete: false,
  },
];

const App = () => {
  const handleChange = () => {};

  return (
    <ul>
      {initialTodos.map(todo => (
        <li key={todo.id}>
          <label>
            <input
              type="checkbox"
              checked={todo.complete}
              onChange={handleChange}
            />
            {todo.task}
          </label>
        </li>
      ))}
    </ul>
  );
};

export default App;
```

It's not possible to change the state of an item with the handler function yet. However, before we can do so, we need to make the list of items stateful by using them as initial state for our useReducer hook with the previously defined reducer function:

```
import React from 'react';

const initialTodos = [...];

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;
        }
      });
    default:
      return state;
  }
};

const App = () => {
  const [todos, dispatch] = React.useReducer(
    todoReducer,
    initialTodos
  );

  const handleChange = () => {};

  return (
    <ul>
      {todos.map(todo => (
        <li key={todo.id}>
          ...
        </li>
      ))}
    </ul>
  );
};

export default App;
```

Now we can use the handler to dispatch an action for our reducer function. Since we need the `id` as the identifier of a todo item in order to toggle its `complete` flag, we can pass the item within the handler function by using a encapsulating arrow function:

```
const App = () => {
  const [todos, dispatch] = React.useReducer(
    todoReducer,
    initialTodos
```

```
  );

  const handleChange = todo => {
    dispatch({ type: 'DO_TODO', id: todo.id });
  };

  return (
    <ul>
      {todos.map(todo => (
        <li key={todo.id}>
          <label>
            <input
              type="checkbox"
              checked={todo.complete}
              onChange={() => handleChange(todo)}
            />
            {todo.task}
          </label>
        </li>
      ))}
    </ul>
  );
};
```

This implementation works only one way though: Todo items can be completed, but the operation cannot be reversed by using our reducer's second state transition. Let's implement this behavior in our handler by checking whether a todo item is completed or not:

```
const App = () => {
  const [todos, dispatch] = React.useReducer(
    todoReducer,
    initialTodos
  );

  const handleChange = todo => {
    dispatch({
      type: todo.complete ? 'UNDO_TODO' : 'DO_TODO',
      id: todo.id,
    });
  };

  return (
    <ul>
      {todos.map(todo => (
        <li key={todo.id}>
          <label>
            <input
              type="checkbox"
              checked={todo.complete}
              onChange={() => handleChange(todo)}
            />
```

```
            {todo.task}
          </label>
        </li>
      ))}
    </ul>
  );
};
```

Depending on the state of our todo item, the correct action is dispatched for our reducer function. Afterward, the React component is rendered again but using the new state from the useReducer hook. The demonstrated useReducer example can be found in this GitHub repository.

. . .

React's useReducer hook is a powerful way to manage state in React. It can be used with useState and useContext for modern state management in React. Also, it is often used in favor of useState for complex state and state transitions. After all, the useReducer hook hits the sweet spot for middle sized applications that don't need Redux for React yet.

KEEP READING ABOUT REACT ›

HOW TO CREATE REDUX WITH REACT HOOKS?

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...

HOW TO USESTATE IN REACT

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. This...

THE ROAD TO REACT

Learn React by building real world applications. No setup configuration. No tooling. Plain React in 200+ pages of learning material. Learn React like **50.000+ readers**.

GET THE BOOK >

Get it on Amazon.

TAKE PART

NEVER MISS AN ARTICLE ABOUT WEB DEVELOPMENT AND JAVASCRIPT.

✔ Join 50.000+ Developers

✔ Learn Web Development with JavaScript