React State Hooks: useReducer, useState, useContext

APRIL 06, 2019 BY ROBIN WIERUCH - EDIT THIS POST

Follow on Twitter Follow on Facebook

If you haven't used state management excessively in React Function Components, this tutorial may help you to get a better understanding of how React Hooks -- such as useState, useReducer, and useContext -- can be used in combination for impressive state management in React applications. In this tutorial, we will almost reach the point where these hooks mimic sophisticated state management libraries like Redux for globally managed state. Let's dive into the application which we will implement together step by step.

TABLE OF CONTENTS

REACT USESTATE: SIMPLE STATE

We start with a list of items -- in our scenario a list of todo items -- which are rendered in our function component with a JavaScript Map Method for Arrays. Each todo item rendered as list item receives a key attribute to notify React about its place in the rendered list:

```
import React from 'react';

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

const App = () => (
  <div>
    <ul>
      {initialTodos.map(todo => (
        <li key={todo.id}>
          <label>{todo.task}</label>
        </li>
      ))}
    </ul>
  </div>
);

export default App;
```

In order to add a new todo item to our list of todo items, we need an input field to give a new todo item a potential `task` property. The `id` and `complete` properties will be automatically added to the item. In React, we can use the State Hook called `useState` to manage something like the value of an input field as state within the component:

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

...

const App = () => {
  const [task, setTask] = useState('');

  const handleChangeInput = event => {

  };

  return (
    <div>
      <ul>
        {initialTodos.map(todo => (
          <li key={todo.id}>
            <label>{todo.task}</label>
          </li>
        ))}
      </ul>

      <input type="text" value={task} onChange={handleChangeInput} />
    </div>
  );
};
```

We also had to give our Function Arrow Component a body with an explicit return statement to get the `useState` hook in between. Now, we can change the `task` state with our handler function, because we have the input's value at our disposal in React's synthetic event:

```
const App = () => {
  const [task, setTask] = useState('');

  const handleChangeInput = event => {
    setTask(event.target.value);
  };

  return (
    <div>
      <ul>
        {initialTodos.map(todo => (
          <li key={todo.id}>
            <label>{todo.task}</label>
          </li>
        ))}
      </ul>
      <input type="text" value={task} onChange={handleChangeInput} />
    </div>
  );
};
```

Now the input field has become a controlled input field, because the value comes directly from the React managed state and the handler changes the state. We implemented our first managed state with the State Hook in React. The whole source code can be seen here.

To continue, let's implement a submit button to add the new todo item to the list of items eventually:

```
const App = () => {
  const [task, setTask] = useState('');

  const handleChangeInput = event => {
    setTask(event.target.value);
  };

  const handleSubmit = event => {
    if (task) {
      // add new todo item
    }

    setTask('');

    event.preventDefault();
  };

  return (
    <div>
      <ul>
        {initialTodos.map(todo => (
          <li key={todo.id}>
            <label>{todo.task}</label>
          </li>
        ))}
      </ul>

      <form onSubmit={handleSubmit}>
        <input
          type="text"
          value={task}
          onChange={handleChangeInput}
        />
        <button type="submit">Add Todo</button>
      </form>
    </div>
  );
};
```

The submit handler doesn't add the new todo item yet, but it makes the input field's value empty again after submitting the new todo item. Also it prevents the default behavior of the browser, because otherwise the browser would perform a refresh after the submit button has been clicked.

In order to add the todo item to our list of todo items, we need to make the todo items managed as state within the component as well. We can use again the useState hook:

```
const App = () => {
  const [todos, setTodos] = useState(initialTodos);
  const [task, setTask] = useState('');

  ...

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

      <form onSubmit={handleSubmit}>
        <input
          type="text"
          value={task}
          onChange={handleChangeInput}
        />
        <button type="submit">Add Todo</button>
      </form>
    </div>
  );
};
```

By having the `setTodos` function at our disposal, we can add the new todo item to the list. The built-in array concat method can be used for this kind of scenario. Also the shorthand property name is used to allocate the task property in the object:

```
const App = () => {
  const [todos, setTodos] = useState(initialTodos);
  const [task, setTask] = useState('');

  const handleChangeInput = event => {
    setTask(event.target.value);
  };
  const handleSubmit = event => {
    if (task) {
      setTodos(todos.concat({ id: 'd', task, complete: false }));
    }

    setTask('');

    event.preventDefault();
  };

  ...
};
```

There is one flaw in this implementation. The new todo item has always the same identifier, which shouldn't be this way for a unique identifier. That's why we can use a library to generate a unique identifier for us. First, you can install it on the command line:

```
npm install uuid
```

Second, you can use it to generate a unique identifier:

```
import React, { useState } from 'react';
import uuid from 'uuid/v4';

const initialTodos = [
  {
    id: uuid(),
    task: 'Learn React',
    complete: true,
  },
  {
    id: uuid(),
    task: 'Learn Firebase',
    complete: true,
  },
  {
    id: uuid(),
    task: 'Learn GraphQL',
    complete: false,
  },
];

const App = () => {
  const [todos, setTodos] = useState(initialTodos);
  const [task, setTask] = useState('');

  const handleChangeInput = event => {
    setTask(event.target.value);
  };
  const handleSubmit = event => {
    if (task) {
      setTodos(todos.concat({ id: uuid(), task, complete: false }));
    }

    setTask('');

    event.preventDefault();
  };

  ...
};
```

You have implemented your second use case for managing state in React by appending an item to a list of items. Again it was possible with the useState hook. The whole source code can be seen here and all changes here.

Last but not least, let's implement a checkbox for each todo item in the list to toggle their complete flags from false to true or true to false.

```
const App = () => {
  const [todos, setTodos] = useState(initialTodos);
  const [task, setTask] = useState('');

  const handleChangeCheckbox = event => {

  };

  ...

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

      ...
    </div>
  );
};
```

Since we need the id of the todo item in our handler function, and not the event, we use a wrapping arrow function to pass the identifier of the individual todo item to our handler:

```
const App = () => {
  const [todos, setTodos] = useState(initialTodos);
  ...

  const handleChangeCheckbox = id => {

  };

  ...

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

      ...
    </div>
  );
};
```

Last, by having the id at our disposal, we can only alter the affected todo item in our list -- by negating the complete flag -- and return every other todo item as before. By using the map method, we return a new array made up of the changed todo item and the remaining todo items:

```
const App = () => {
  const [todos, setTodos] = useState(initialTodos);
  ...

  const handleChangeCheckbox = id => {
    setTodos(
      todos.map(todo => {
        if (todo.id === id) {
          return { ...todo, complete: !todo.complete };
        } else {
          return todo;
        }
      })
    );
  };

  ...

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

      ...
    </div>
  );
};
```

That's it. The new todo items are immediately set as state for the list of todo items with the `setTodos` function. The whole source code can be seen here and all changes here. Congratulations, you have implemented a whole todo application with three use cases for state management with the useState hook:

- input field state for tracking task property of new todo item
- adding a todo item to list with a submit button
- checking (and unchecking) a todo item with checkboxes

Exercises:

- Read more about React's useState Hook

REACT USEREDUCER: COMPLEX STATE

The useState hook is great to manage simple state. However, once you run into more complex state objects or state transitions -- which you want to keep maintainable and predictable --, the useReducer hook is a great candidate to manage them. Here you can find a comparison of when to use the useState or useReducer hook. Let's continue implementing our application with the useReducer hook by going through a simpler example first. In our next scenario, we want to add buttons to filter our list of todos for three cases:

- show all todo items
- show only complete todo items
- show only incomplete todo items

Let's see how we can implement these with three buttons:

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

  const handleShowAll = () => {

  };

  const handleShowComplete = () => {

  };

  const handleShowIncomplete = () => {

  };

  ...

  return (
    <div>
      <div>
        <button type="button" onClick={handleShowAll}>
          Show All
        </button>
        <button type="button" onClick={handleShowComplete}>
          Show Complete
        </button>
        <button type="button" onClick={handleShowIncomplete}>
          Show Incomplete
        </button>
      </div>

      ...
    </div>
  );
};
```

We will care later about these. Next, let's see how we can map the three cases in a reducer function:

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

A reducer function always receives the current state and an action as arguments. Depending on the mandatory type of the action, it decides what task to perform in the switch case statement, and returns a new state based on the implementation details. In our case, the implementation details are straightforward:

- In case of action type `SHOW_ALL`, return `ALL` string as state.
- In case of action type `SHOW_COMPLETE`, return `COMPLETE` string as state.
- In case of action type `SHOW_INCOMPLETE`, return `INCOMPLETE` string as state.
- If none of the action types are matched, throw an error to notify us about a bad implementation.

Now we can use the reducer function in a useReducer hook. It takes the reducer function and an initial state and returns the filter state and the dispatch function to change it:

```
import React, { useState, useReducer } from 'react';

...

const App = () => {
  const [filter, dispatchFilter] = useReducer(filterReducer, 'ALL');

  ...
};
```

First, the dispatch function can be used with an action object -- with an action type which is used within the reducer to evaluate the new state:

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

  ...

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

  const handleShowComplete = () => {
    dispatchFilter({ type: 'SHOW_COMPLETE' });
  };

  const handleShowIncomplete = () => {
    dispatchFilter({ type: 'SHOW_INCOMPLETE' });
  };

  ...
};
```

Second, -- after we are able to transition from state to state with the reducer function and the action with action type -- the filter state can be used to show only the matching todo items by using the built-in array filter method:

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

  ...

  const filteredTodos = todos.filter(todo => {
    if (filter === 'ALL') {
      return true;
    }

    if (filter === 'COMPLETE' && todo.complete) {
      return true;
    }

    if (filter === 'INCOMPLETE' && !todo.complete) {
      return true;
    }

    return false;
  });

  ...

  return (
    <div>
      ...

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

      ...
    </div>
  );
};
```

The filter buttons should work now. Every time a button is clicked an action with an action type is dispatched for the reducer function. The reducer function computes then the new state. Often the current state from the reducer function's argument is used to compute the new state with the incoming action. But in this simpler example, we only transition from one JavaScript string to another string as state.

The whole source code can be seen here and all changes here.

Note: The shown use case -- also every other use case with useReducer -- can be implemented with useState as well. However, even though this one is a simpler example for the sake of learning about it, it shows clearly how much its helps for the reasoning for the state transitions by just reading the reducer function.

The useReducer hook is great for predictable state transitions as we have seen in the previous example. Next, we are going to see how it is a good fit for complex state objects too. Therefore, we will start to manage our todo items in a reducer hook and manipulate it with the following transitions:

- Toggle todo item to complete.
- Toggle todo item to incomplete.
- Add todo item to the list of todo items.

The reducer would look like the following:

```
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:
      throw new Error();
  }
};
```

The following transitions are implemented in the reducer:

- DO_TODO: If an action of this kind passes the reducer, the action comes with an additional payload, the todo item's `id`, to identify the todo item that should be changed to **complete** status.
- UNDO_TODO: If an action of this kind passes the reducer, the action comes with an additional payload, the todo item's `id`, to identify the todo item that should be changed to **incomplete** status.
- ADD_TODO: If an action of this kind passes the reducer, the action comes with an additional payload, the new todo item's `task`, to concat the new todo item to the current todo items in the state.

Instead of the useState hook from before, we can manage our todos with this new reducer and the initially given todo items:

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

  ...
};
```

If someone toggles a todo item with the checkbox element, a new handler is used to dispatch an action with the appropriate action type depending on the todo item's complete status:

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

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

  ...

  return (
    <div>
      ...

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

      ...
    </div>
  );
};
```

If someone submits a new todo item with the button, the same handler is used but to dispatch an action with the correct action type and the name of the todo item (`task`) and its identifier (`id`) as payload:

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

  const handleSubmit = event => {
    if (task) {
      dispatchTodos({ type: 'ADD_TODO', task, id: uuid() });
    }

    setTask('');

    event.preventDefault();
  };

  ...

  return (
    <div>
      ...

      <form onSubmit={handleSubmit}>
        <input
          type="text"
          value={task}
          onChange={handleChangeInput}
        />
        <button type="submit">Add Todo</button>
      </form>
    </div>
  );
};
```

Now everything that has been managed by useState for our todo items is managed by useReducer now. The reducer describes what happens for each state transition and how this happens by moving the implementation details in there. The whole source code can be seen here and all changes here.

You have seen how useState and useReducer can be used for simple and complex state management whereas useReducer gives you clear state transitions -- thus improved predictability -- and a better way to manage complex objects.

Exercises:

- Read more about React's useReducer Hook

REACT USECONTEXT: GLOBAL STATE

We can take our state management one step further. At the moment, the state is managed co-located to the component. That's because we only have one component after all. What if we would have a deep component tree? How could we dispatch state changes from anywhere?

Let's dive into React's Context API and the useContext hook to mimic more a Redux's philosophy by making state changes available in the whole component tree. Before we can do this, let's refactor our one component into a component tree. First, the App component renders all its child components and passes the necessary state and dispatch functions to them:

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

  const filteredTodos = todos.filter(todo => {
    if (filter === 'ALL') {
      return true;
    }

    if (filter === 'COMPLETE' && todo.complete) {
      return true;
    }

    if (filter === 'INCOMPLETE' && !todo.complete) {
      return true;
    }

    return false;
  });

  return (
    <div>
      <Filter dispatch={dispatchFilter} />
      <TodoList dispatch={dispatchTodos} todos={filteredTodos} />
      <AddTodo dispatch={dispatchTodos} />
    </div>
  );
};
```

Second, the Filter component with its buttons and handlers which are using the dispatch function:

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

  const handleShowComplete = () => {
    dispatch({ type: 'SHOW_COMPLETE' });
  };

  const handleShowIncomplete = () => {
    dispatch({ type: 'SHOW_INCOMPLETE' });
  };

  return (
    <div>
      <button type="button" onClick={handleShowAll}>
        Show All
      </button>
      <button type="button" onClick={handleShowComplete}>
        Show Complete
      </button>
      <button type="button" onClick={handleShowIncomplete}>
        Show Incomplete
      </button>
    </div>
  );
};
```

Third, the TodoList and TodoItem components. Since the individual TodoItem component defines its own handler, the onChange event handler doesn't need to pass the todo item anymore. The item is already available in the component itself:

```
const TodoList = ({ dispatch, todos }) => (
  <ul>
    {todos.map(todo => (
      <TodoItem key={todo.id} dispatch={dispatch} todo={todo} />
    ))}
  </ul>
);

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

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

Last, the AddTodo component which uses is own local state to manage the value of the input field:

```
const AddTodo = ({ dispatch }) => {
  const [task, setTask] = useState('');

  const handleSubmit = event => {
    if (task) {
      dispatch({ type: 'ADD_TODO', task, id: uuid() });
    }

    setTask('');

    event.preventDefault();
  };

  const handleChange = event => setTask(event.target.value);

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" value={task} onChange={handleChange} />
      <button type="submit">Add Todo</button>
    </form>
  );
};
```

In the end, we have a component tree whereas each component receives state as props and dispatch functions to alter the state. Most of the state is managed by the parent App component. That's it for the refactoring. The whole source code can be seen here and all changes here.

Now, the component tree isn't very deep and it isn't difficult to pass props down. However, in larger applications it can be a burden to pass down everything several levels. That's why React came up with the idea of the context container. Let's see how we can pass the dispatch functions down with React's Context API. First, we create the context:

```
import React, { useState, useReducer, createContext } from 'react';
...

const TodoContext = createContext(null);

...
```

Second, the App can use the context's Provider method to pass implicitly a value down the component tree:

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

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

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

Now, the dispatch function doesn't need to be passed down to the components anymore, because it's available in the context:

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

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

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

The useContext hook helps us to retrieve the value from the context in the AddTodo component:

```
import React, {
  useState,
  useReducer,
  useContext,
  createContext,
} from 'react';

...
```

```
const AddTodo = () => {
  const dispatch = useContext(TodoContext);

  const [task, setTask] = useState('');

  const handleSubmit = event => {
    if (task) {
      dispatch({ type: 'ADD_TODO', task, id: uuid() });
    }

    setTask('');

    event.preventDefault();
  };

  const handleChange = event => setTask(event.target.value);

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" value={task} onChange={handleChange} />
      <button type="submit">Add Todo</button>
    </form>
  );
};
```

Also the TodoItem component makes use of it and the dispatch function doesn't need to be passed through the TodoList component anymore:

```
const TodoList = ({ todos }) => (
  <ul>
    {todos.map(todo => (
      <TodoItem key={todo.id} todo={todo} />
    ))}
  </ul>
);

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

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

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

The application works again, but we are able to dispatch changes for our todo list from anywhere. If you want to continue with this application, experiment with passing down the dispatch function for the filter reducer as well. Moreover, you can pass the state coming from useReducer with React's Context API down as well. Just try it yourself. The whole source code can be seen here and all changes here.

Exercises:

- Read more about React's useContext Hook
- Read more about implementing Redux with React Hooks

. . .

You have learned how modern state management is used in React with useState, useReducer and useContext. Whereas useState is used for simple state (e.g. input field), useReducer is a greater asset for complex objects and complicated state transitions. In larger applications, useContext can be used to pass down dispatch functions (or state) from the useReducer hook.

This tutorial is part 1 of 2 in this series.

Part 2: How to create Redux with React Hooks?

Show Comments

KEEP READING ABOUT REACT ›

MOBX REACT: REFACTOR YOUR APPLICATION FROM REDUX TO MOBX

MobX is a state management solution. It is a standalone pure technical solution without being opinionated about the architectural state management app design. The 4 pillars State, Actions, Reactions...

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...

TAKE PART

NEVER MISS AN ARTICLE ABOUT WEB DEVELOPMENT AND JAVASCRIPT.

✓ Join 50.000+ Developers

✓ Learn Web Development with JavaScript

✓ Tips and Tricks

✓ Access Tutorials, eBooks and Courses

✓ Personal Development as a Software Engineer

Your email address SUBSCRIBE

View our Privacy Policy.

PORTFOLIO

Online Courses

Open Source

Tutorials

ABOUT

About me

What I use

How to work with me

How to support me