React useReducer with Middleware

DECEMBER 10, 2020 BY ROBIN WIERUCH - EDIT THIS POST

Follow on Twitter 17k Follow on Facebook

This tutorial is part 3 of 3 in this series.

Part 1: What is a reducer in JavaScript?
Part 2: How to useReducer in React

In this React Hooks tutorial, I want to show you how to use a middleware for React's useReducer Hook. This middleware would run either before or after the state transition of the reducer and enables you to opt-in features.

Before we can start, let's establish what we have as a baseline from the previous useReducer tutorial: Our React application looks like the following.

First, we have all of our items -- which serve as our initial state and which will become stateful eventually -- in a list:

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

Second, we have our reducer function, which enables us to transition from one state to another state by using actions:

```
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;
  }
};
```

And last but not least, we have our React component which uses React's useReducer Hook from the previous React Hooks tutorial:

```
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>
  );
};
```

From here, we want extend the application -- to be more specific the reducer -- with a middleware. The simplest middleware would be a logger which would output something before or after the reducer's state transition. Let's get started.

REACT'S USEREDUCER HOOK WITH MIDDLEWARE

The logger middleware we want to establish for our reducer as an example could look like the following function which outputs the reducer's action -- which is in charge of transition our state from one state to another state -- to the developer's console log:

```
const logger = action => {
  console.log('logger:', action);
};
```

In our usage of React's useReducer Hook, we would want to use the middleware the following way:

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

  ...
};
```

What we have right now could be pretty straightforward if React's useReducer Hook would support middleware usage natively. But it doesn't, so we need to come up with a custom hook:

```
const useReducerWithMiddleware = (
  reducer,
  initialState,
  middlewareFn
) => {
  const [state, dispatch] = React.useReducer(reducer, initialState);

  // do something with middlewareFn

  return [state, dispatch];
};

const App = () => {
  const [todos, dispatch] = useReducerWithMiddleware(
    todoReducer,
    initialTodos,
    logger
  );

  ...
};
```

With the middleware function at our hands in the custom hook, we can enhance the useReducer's dispatch function with a higher-order function:

```
const useReducerWithMiddleware = (
  reducer,
  initialState,
  middlewareFn
) => {
  const [state, dispatch] = React.useReducer(reducer, initialState);

  const dispatchWithMiddleware = (action) => {
    middlewareFn(action);
    dispatch(action);
  };

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

What we return from the custom hook is not the dispatch function anymore, but an extended version of it where we pass the action through the middleware before we pass it to the dispatch function.

You could check when this middleware executes, before or after the dispatch function which performs the state transition, if you would insert a logging statement in your reducer function:

```
const todoReducer = (state, action) => {
  console.log(state, action);
  switch (action.type) {
    ...
  }
};
```

That's it for a very basic reducer middleware, however, we are lacking two crucial features: First, we are only able to use one middleware function in this custom hook. And second, the middleware always executes before the state transition with dispatch, so what if we would want to have it executing after the state transition instead. Let's tackle these limitations next.

REACT'S USEREDUCER WITH MULTIPLE MIDDLEWARE

What we maybe want to have is multiple middleware functions that we can pass to the custom hook. In the following scenario, we pass two times the same middleware function as an array:

```
const App = () => {
  const [todos, dispatch] = useReducerWithMiddleware(
    todoReducer,
    initialTodos,
    [logger, logger]
  );

  ...
};
```

The custom hook changes the following way to execute multiple middleware functions:

```
const useReducerWithMiddleware = (
  reducer,
  initialState,
  middlewareFns
) => {
  const [state, dispatch] = React.useReducer(reducer, initialState);

  const dispatchWithMiddleware = (action) => {
    middlewareFns.forEach((middlewareFn) => middlewareFn(action));
    dispatch(action);
  };

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

Because we are able to pass multiple middleware functions to our custom useReducer hook, we solved the first limitation. However, all middleware functions still execute before the state transition with the actual dispatch function. Let's tackle this last limitation.

REACT'S USEREDUCER WITH AFTERWARE

Let's say we have two middleware functions whereas one executes before and the other one executes after the state transition:

```
const loggerBefore = (action) => {
  console.log('logger before:', action);
};

const loggerAfter = (action) => {
  console.log('logger after:', action);
};
```

Event though the logging and the name of the functions are different, the functions are doing the same thing. So we need a way to tell them when (before or after dispatch) to execute. A straigthforward way would be using two arrays that we pass to our custom hook:

```
const App = () => {
  const [todos, dispatch] = useReducerWithMiddleware(
    todoReducer,
    initialTodos,
    [loggerBefore],
    [loggerAfter]
  );

  ...
};
```

Then our custom reducer hook could act upon the middleware functions which run before as we had it before. In a naive approach, we would simply put the afterware functions after the dispatch function:

```
const useReducerWithMiddleware = (
  reducer,
  initialState,
  middlewareFns,
  afterwareFns
) => {
  const [state, dispatch] = React.useReducer(reducer, initialState);

  const dispatchWithMiddleware = (action) => {
    middlewareFns.forEach((middlewareFn) => middlewareFn(action));

    dispatch(action);

    afterwareFns.forEach((afterwareFn) => afterwareFn(action));
  };

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

However, this doesn't work, because dispatch updates the state asynchronously. So instead, we can wait for any state change in a useEffect hook:

```
const useReducerWithMiddleware = (
  reducer,
  initialState,
  middlewareFns,
  afterwareFns
) => {
  const [state, dispatch] = React.useReducer(reducer, initialState);
  const dispatchWithMiddleware = (action) => {
    middlewareFns.forEach((middlewareFn) => middlewareFn(action));

    dispatch(action);
  };

  React.useEffect(() => {
    afterwareFns.forEach(afterwareFn);
  }, [afterwareFns]);

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

For the afterward functions, we don't have the action at our disposal anymore. We can change this by using a ref instance variable -- which will be written before we dispatch the action and which can then be read after we dispatched the action:

```
const useReducerWithMiddleware = (
  reducer,
  initialState,
  middlewareFns,
  afterwareFns
) => {
  const [state, dispatch] = React.useReducer(reducer, initialState);

  const aRef = React.useRef();

  const dispatchWithMiddleware = (action) => {
    middlewareFns.forEach((middlewareFn) => middlewareFn(action));

    aRef.current = action;

    dispatch(action);
  };

  React.useEffect(() => {
    if (!aRef.current) return;

    afterwareFns.forEach((afterwareFn) => afterwareFn(aRef.current));
  }, [afterwareFns]);

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

In addition, this instance variable adds the benefit of not having the side-effect function in our useEffect hook execute on mount for the component. Instead it only executes once the action has been set.

We are done with our middleware and afterware. If you want to pass in more information to your middleware/afterware functions, you can do it like this:

```
const loggerBefore = (action, state) => {
  console.log('logger before:', action, state);
};

const loggerAfter = (action, state) => {
  console.log('logger after:', action, state);
};

const useReducerWithMiddleware = (
  reducer,
  initialState,
  middlewareFns,
  afterwareFns
) => {
  const [state, dispatch] = React.useReducer(reducer, initialState);

  const aRef = React.useRef();

  const dispatchWithMiddleware = (action) => {
    middlewareFns.forEach((middlewareFn) =>
      middlewareFn(action, state)
    );

    aRef.current = action;

    dispatch(action);
  };

  React.useEffect(() => {
    if (!aRef.current) return;

    afterwareFns.forEach((afterwareFn) =>
      afterwareFn(aRef.current, state)
    );
  }, [afterwareFns, state]);

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

That's it. You are now able to run functions prior and after changing the state with React's useReducer Hook by using middlew Show Comments

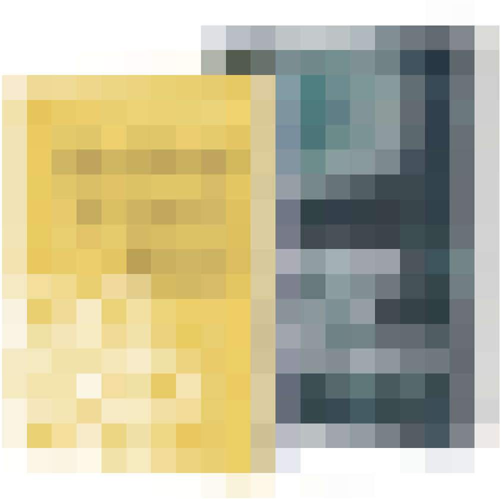
KEEP READING ABOUT REACT >

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 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...



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**.

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

ABOUT

About me

Open Source

Tutorials

What I use

How to work with me

How to support me

Contact Me Privacy & Terms