What is a Reducer in JavaScript/React/Redux?

APRIL 19, 2019 BY ROBIN WIERUCH - EDIT THIS POST

The concept of a Reducer became popular in JavaScript with the rise of Redux as state management solution for React. But no worries, you don't need to learn Redux to understand Reducers. Basically reducers are there to manage state in an application. For instance, if a user writes something in an HTML input field, the application has to manage this UI state (e.g. controlled components).

Let's dive into the implementation details: In essence, a reducer is a function which takes two arguments -- the current state and an action -- and returns based on both arguments a new state. In a pseudo function it could be expressed as:

```
(state, action) => newState
```

As example, it would look like the following in JavaScript for the scenario of increasing a number by one:

```
function counterReducer(state, action) {
  return state + 1;
}
```

Or defined as JavaScript arrow function, it would look the following way for the same logic:

```
const counterReducer = (state, action) => {
  return state + 1;
};
```

In this case, the current state is an integer (e.g. count) and the reducer function increases the count by one. If we would rename the argument `state` to `count`, it may be more readable and approachable by newcomers to this concept. However, keep in mind that the `count` is still the state:

```
const counterReducer = (count, action) => {
  return count + 1;
};
```

The reducer function is a pure function without any side-effects, which means that given the same input (e.g. `state` and `action`), the expected output (e.g. `newState`) will always be the same. This makes reducer functions the perfect fit for reasoning about state changes and testing them in isolation. You can repeat the same test with the same input as arguments and always expect the same output:

```
expect(counterReducer(0)).to.equal(1); // successful test
expect(counterReducer(0)).to.equal(1); // successful test
```

That's the essence of a reducer function. However, we didn't touch the second argument of a reducer yet: the action. The `action` is normally defined as an object with a `type` property. Based on the type of the action, the reducer can perform conditional state transitions:

```
const counterReducer = (count, action) => {
  if (action.type === 'INCREASE') {
    return count + 1;
  }
```

```
  if (action.type === 'DECREASE') {
    return count - 1;
  }

  return count;
};
```

If the action `type` doesn't match any condition, we return the unchanged state. Testing a reducer function with multiple state transitions -- given the same input, it will always return the same expected output -- still holds true as mentioned before which is demonstrated in the following test cases:

```
// successful tests
// because given the same input we can always expect the same output
expect(counterReducer(0, { type: 'INCREASE' })).to.equal(1);
expect(counterReducer(0, { type: 'INCREASE' })).to.equal(1);

// other state transition
expect(counterReducer(0, { type: 'DECREASE' })).to.equal(-1);

// if an unmatching action type is defined the current state is returned
expect(counterReducer(0, { type: 'UNMATCHING_ACTION' })).to.equal(0);
```

However, more likely you will see a switch case statement in favor of if else statements in order to map multiple state transitions for a reducer function. The following reducer performs the same logic as before but expressed with a switch case statement:

```
const counterReducer = (count, action) => {
  switch (action.type) {
    case 'INCREASE':
      return count + 1;
    case 'DECREASE':
      return count - 1;
    default:
      return count;
  }
};
```

In this scenario, the `count` itself is the state on which we are applying our state changes upon by increasing or decreasing the count. However, often you will not have a JavaScript primitive (e.g. integer for count) as state, but a complex JavaScript object. For instance, the count could be one property of our `state` object:

```
const counterReducer = (state, action) => {
  switch (action.type) {
```

```
    case 'INCREASE':
      return { ...state, count: state.count + 1 };
    case 'DECREASE':
      return { ...state, count: state.count - 1 };
    default:
      return state;
  }
};
```

Don't worry if you don't understand immediately what's happening in the code here. Foremost, there are two important things to understand in general:

- **The state processed by a reducer function is immutable.** That means the incoming state -- coming in as argument -- is never directly changed. Therefore the reducer function always has to return a new state object. If you haven't heard about immutability, you may want to check out the topic immutable data structures.
- Since we know about the state being a immutable data structure, we can use the JavaScript spread operator to **create a new state object from the incoming state and the part we want to change** (e.g. `count` property). This way we ensure that the other properties that aren't touch from the incoming state object are still kept intact for the new state object.

Let's see these two important points in code with another example where we want to change the last name of a person object with the following reducer function:

```
const personReducer = (person, action) => {
  switch (action.type) {
    case 'INCREASE_AGE':
      return { ...person, age: person.age + 1 };
    case 'CHANGE_LASTNAME':
      return { ...person, lastname: action.lastname };
    default:
      return person;
  }
};
```

We could change the last name of a user the following way in a test environment:

```
const initialState = {
  firstname: 'Liesa',
  lastname: 'Huppertz',
  age: 30,
};

const action = {
  type: 'CHANGE_LASTNAME',
```

```
  lastname: 'Wieruch',
};

const result = personReducer(initialState, action);

expect(result).to.equal({
  firstname: 'Liesa',
  lastname: 'Wieruch',
  age: 30,
});
```

You have seen that by using the JavaScript spread operator in our reducer function, we use all the properties from the current state object for the new state object but override specific properties (e.g. `lastname`) for this new object. That's why you will often see the spread operator for keeping state operation immutable (= state is not changed directly).

Also you have seen another aspect of a reducer function: **An action provided for a reducer function can have an optional payload** (e.g. `lastname`) next to the mandatory action type property. The payload is additional information to perform the state transition. For instance, in our example the reducer wouldn't know the new last name of our person without the extra information.

Often the optional payload of an action is put into another generic `payload` property to keep the top-level of properties of an action object more general (.e.g `{ type, payload }`). That's useful for having type and payload always separated side by side. For our previous code example, it would change the action into the following:

```
const action = {
  type: 'CHANGE_LASTNAME',
  payload: {
    lastname: 'Wieruch',
  },
};
```

The reducer function would have to change too, because it has to dive one level deeper into the action:

```
const personReducer = (person, action) => {
  switch (action.type) {
    case 'INCREASE_AGE':
      return { ...person, age: person.age + 1 };
    case 'CHANGE_LASTNAME':
      return { ...person, lastname: action.payload.lastname };
    default:
      return person;
```

```
    }
};
```

Basically you have learned everything you need to know for reducers. They are used to perform state transitions from A to B with the help of actions that provide additional information. You can find reducer examples from this tutorial in this GitHub repository including tests. Here again everything in a nutshell:

- **Syntax:** In essence a reducer function is expressed as `(state, action) => newState`.
- **Immutability:** State is never changed directly. Instead the reducer always creates a new state.
- **State Transitions:** A reducer can have conditional state transitions.
- **Action:** A common action object comes with a mandatory type property and an optional payload:
  - The type property chooses the conditional state transition.
  - The action payload provides information for the state transition.

Also check out this tutorial if you want to know how to use reducers in React with the useReducer hook.

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...

REDUX DUCKS: RESTRUCTURE YOUR REDUX APP WITH DUCKS

The Redux Ducks: Restructure your Redux App with Ducks tutorial will teach you how to bundle action creators, action types and reducers side by side in your

Redux app. Usually in the beginning of...