

# Handling Responses to Actions in the Reducer

In Redux, all actions pass through the reducer. Surely, the reducer must have a way to differentiate between different actions. Let's find out.

Now that you successfully understand what an action is, it is important to see how they become useful i.e in a practical sense.

Earlier, I did say that a reducer takes in two arguments. One, **state**, the other, **action**. Here's what a simple Reducer looks like:

```
function reducer(state, action) {  
  //return new state  
}
```

The action is passed in as the second parameter to the Reducer, however, we've done nothing with it within the function itself.

To handle the actions passed into the reducer, you typically write a **switch** statement within your reducer, like this:

```
function reducer (state, action) {  
  switch (action.type) {  
  
    case "withdraw_money":  
      //do something  
      break;  
    case "deposit-money":  
      //do something  
      break;  
    default:  
      return state;  
  }  
}
```



Some people seem not to like the switch statement, but it's basically an **if/else** for possible values on a single field.

The code above will switch over the action type and do something based on the type of action passed in. Technically, the *do something* bit is required to

the type of action passed in. Technically, the `isClicked` field is required to return a new state.

Let me explain further. Assume that you had 2 hypothetical buttons, button #1 and button #2 on a certain webpage, and your state object looked something like this:

```
{
  isOpen: true,
  isClicked: false,
}
```



When button #1 is clicked, you want to toggle the `isOpen` field. In the context of a React app, the solution is simple. As soon as the button is clicked, you would do this:

```
this.setState({isOpen: !this.state.isOpen})
```

Also, let's assume that when #2 is clicked, you want to update the `isClicked` field. Again, the solution is simple, and along the lines of this:

```
this.setState({isClicked: !this.state.isClicked})
```

Good. With a Redux app, you can't use `setState()` to update the state object managed by Redux.

You have to dispatch an action first. Let's assume the actions are as below:

#1 :

```
{
  type: "is_open"
}
```

#2 :

```
{
  type: "is_clicked"
}
```

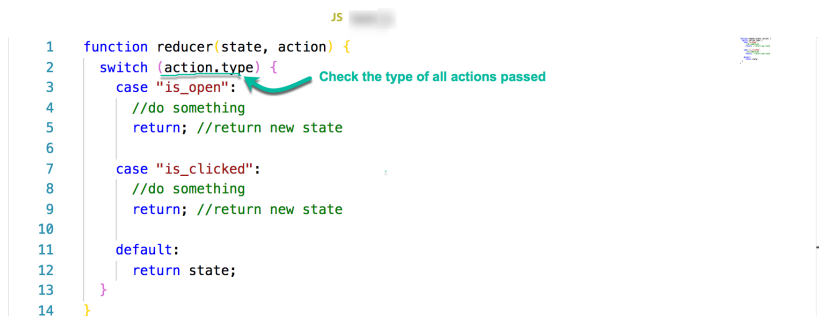
In a Redux app, **every action flows through the reducer**. All of them. So, in this example, both action #1 and action #2 will pass through the same reducer.

In this case, how does the reducer differentiate each of them?

Yeah, you guessed right.

By switching over the **action.type** we can handle both actions without hassles. Here is what I mean:

```
function reducer (state, action) {  
  switch (action.type) {  
    case "is_open":  
      return; //return new state  
    case "is_clicked":  
      return; //return new state  
    default:  
      return state;  
  }  
}
```



```
1 function reducer(state, action) {  
2   switch (action.type) {  
3     case "is_open":  
4       //do something  
5       return; //return new state  
6  
7     case "is_clicked":  
8       //do something  
9       return; //return new state  
10  
11    default:  
12      return state;  
13  }  
14 }
```

Now you see why the switch statement is useful. It is important to handle each action type separately.

Here's an important question to ask yourself

Q

For a single component of an application, we must:

In the next section, we will continue with the task of building the mini app below:

