Immutable Update Patterns

Immutable Update Patterns on reduxjs.org:

Updating Nested Objects

The key to updating nested data is **that *every* level of nesting must be copied and updated appropriately**. This is often a difficult concept for those learning Redux, and there are some specific problems that frequently occur when trying to update nested objects. These lead to accidental direct mutation, and should be avoided.

Common Mistake #1: New variables that point to the same objects

Defining a new variable does *not* create a new actual object - it only creates another reference to the same object. An example of this error would be:

```
1.   function updateNestedState(state, action) {
2.      let nestedState = state.nestedState;
3.      // ERROR: this directly modifies the existing object reference - don't do this!
4.      nestedState.nestedField = action.data;
5.
6.      return {
7.         ...state,
8.         nestedState
9.      };
10.  }
```

This function does correctly return a shallow copy of the top-level state object, but because the `nestedState` variable was still pointing at the existing object, the state was directly mutated.

Common Mistake #2: Only making a shallow copy of one level

Another common version of this error looks like this:

```
1.   function updateNestedState(state, action) {
```

```
2.        // Problem: this only does a shallow copy!
3.        let newState = {...state};
4.
5.        // ERROR: nestedState is still the same object!
6.        newState.nestedState.nestedField = action.data;
7.
8.        return newState;
9.    }
```

Doing a shallow copy of the top level is *not* sufficient - the `nestedState` object should be copied as well.

Correct Approach: Copying All Levels of Nested Data

Unfortunately, the process of correctly applying immutable updates to deeply nested state can easily become verbose and hard to read. Here's what an example of updating `state.first.second[someId].fourth` might look like:

```
1.    function updateVeryNestedField(state, action) {
2.        return {
3.            ...state,
4.            first : {
5.                ...state.first,
6.                second : {
7.                    ...state.first.second,
8.                    [action.someId] : {
9.                        ...state.first.second[action.someId],
10.                        fourth : action.someValue
11.                    }
12.                }
13.            }
14.        }
15.    }
```

Obviously, each layer of nesting makes this harder to read, and gives more chances to make mistakes. This is one of several reasons why you are encouraged to keep your state flattened, and compose reducers as much as possible.

Inserting and Removing Items in Arrays

Normally, a Javascript array's contents are modified using mutative functions like `push`, `unshift`, and `splice`. Since we don't want to mutate state directly in reducers, those should normally be avoided. Because of that, you might see "insert" or "remove" behavior written like this:

```
1.  function insertItem(array, action) {
2.      return [
3.          ...array.slice(0, action.index),
4.          action.item,
5.          ...array.slice(action.index)
6.      ]
7.  }
8.
9.  function removeItem(array, action) {
10.     return [
11.         ...array.slice(0, action.index),
12.         ...array.slice(action.index + 1)
13.     ];
14. }
```

However, remember that the key is that the *original in-memory reference* is not modified. **As long as we make a copy first, we can safely mutate the copy**. Note that this is true for both arrays and objects, but nested values still must be updated using the same rules.

This means that we could also write the insert and remove functions like this:

```
1.  function insertItem(array, action) {
2.      let newArray = array.slice();
3.      newArray.splice(action.index, 0, action.item);
4.      return newArray;
5.  }
6.
7.  function removeItem(array, action) {
8.      let newArray = array.slice();
9.      newArray.splice(action.index, 1);
10.     return newArray;
11. }
```

The remove function could also be implemented as:

```
1.  function removeItem(array, action) {
2.      return array.filter( (item, index) => index !== action.index);
3.  }
```

Updating an Item in an Array

Updating one item in an array can be accomplished by using `Array.map`, returning a new value for the item we want to update, and returning the existing values for all other items:

```
1.  function updateObjectInArray(array, action) {
2.      return array.map( (item, index) => {
3.          if(index !== action.index) {
```

```
4.          // This isn't the item we care about - keep it as-is
5.            return item;
6.        }
7.
8.          // Otherwise, this is the one we want - return an updated value
9.          return {
10.           ...item,
11.           ...action.item
12.        };
13.    });
14.  }
```

Immutable Update Utility Libraries

Because writing immutable update code can become tedious, there are a number of utility libraries that try to abstract out the process. These libraries vary in APIs and usage, but all try to provide a shorter and more succinct way of writing these updates. Some, like dot-prop-immutable, take string paths for commands:

```
1.    state = dotProp.set(state, `todos.${index}.complete`, true)
```

Others, like immutability-helper (a fork of the now-deprecated React Immutability Helpers addon), use nested values and helper functions:

```
1.    var collection = [1, 2, {a: [12, 17, 15]}];
2.    var newCollection = update(collection, {2: {a: {$splice: [[1, 1, 13, 14]]}}});
```

They can provide a useful alternative to writing manual immutable update logic.

Immutable Data#Immutable Update Utilities section of the Redux Addons Catalog.