\*\*\*\* WHY NOT TO MUTATE STATE IN REACT \*\*\*\*\*\*\*\*\*\*\*\*

To understand state mutations in react, first one need to understand basics of data mutations in javascript.

If something is immutable it means it can not be modified. In javascript there are 5 primitives. numbers, strings, booleans, undefined and null, these are immutable.

Example1:-

let str = ‘abc’

str cannot be modified directly, so if we want to do modifications to the string we have to use functions like replace, toUpperCase etc. which create a new string.

Example2:-

let str1 = ‘abc’

let str2 = ‘abc’

Now str1 === str2 will returns true, because both str1 and str2 have same value and refer to the same primitive.

But this does not work for objects.

let str1 = new String(‘abc’)

let str2 = new String(‘abc’)

str1 === str2 returns false because although the values are equal, two different objects are created in memory, so they both refer to two different objects, so this comparison has failed in reference comparison.

In javascript, if you want to compare the values of arrays or objects there is no direct way to do, each individual elements in array or key value pairs in object need to be compared.

Problems with state mutations in react

In React docs, it’s mentioned that never change this.state directly instead always use this.setState for any state updates, these are two main reasons to do this:

a.) setState works in batches, which means one cannot expect the setState to do the state update immediately, it is an asynchronous operation so the state changes may happen in later point in time which means manually mutating state may get overriden by setState.

b.) Performance. When using pure component or shouldComponentUpdate, they will do a shallow compare using === operator, but if you mutate the state the object reference will still be the same so the comparison would fail.

Avoiding array/object mutations

a.) Use slice

let x = [‘a’, ’b’, ’c’, ’d’, ’e’]

If we want to remove c from the above array and print the array, we can do as below.

x.splice(2,1)

console.log(x) // prints [‘a’, ’b’, ’d’, ’e’]

but splice directly modified x, so it has mutated the array.

This can be achieved immutably as below using slice and concat as these are immutable operations,

let x = [‘a’, ’b’, ’c’, ’d’, ’e’]

let y = x.slice(0,2).concat(x.slice(3))

console.log(x) // prints original array [‘a’, ’b’, ’c’, ’d’, ’e’]

console.log(y) // prints [‘a’, ’b’, ’d’, ’e’]

b.) Use Object.assign

let x = { ‘a’:’Hello’, ‘b’: ‘Hey’ }

Now let’s say if we want to change the value of ‘a’ from ‘Hello’ to ‘Hurray’.

Mutable way:-

x.a = ‘Hurray’, this will directly modify the object which we want to avoid in react if x belongs to state.

Immutable way:-

let y = Object.assign({}, x }// creates a brand new object

y.a = ‘Hurray’, now y can be used to update the react state as this is completely immutable.

c.) Use Spread operator in ES6

The same functionality as in above can be achieved using spread operator.

let x = { ‘a’:’Hello’, ‘b’: ‘Hey’ }

let y = {…x,’a’:’Hurray’}

console.log(x) // prints { a: ‘Hello’, b: ‘Hey’ }

console.log(y)// prints { a: ‘Hurray’, b: ‘Hey’ }

d.) Nested Objects

Let’s say state contains user object which looks as below,

let user = {

profile:{

address:{

city: ‘London’

}

}

}

If we want to modify city from London to Newyork immutably, we need to do it as below

{

…state

user:{

…state.user,

profile:{…state.user.profile,

address:{…state.user.profile.address, city:’Newyork’}}

}

}

\*\*\*\* Immutable Update Patterns \*\*\*\*\*\*\*\*\*\*\*\*

Immutable Update Patterns on reduxjs.org: http://redux.js.org/docs/recipes/reducers/ImmutableUpdatePatterns.html

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:

function updateNestedState(state, action) {

let nestedState = state.nestedState;

// ERROR: this directly modifies the existing object reference - don't do this!

nestedState.nestedField = action.data;

return {

...state,

nestedState

};

}

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:

function updateNestedState(state, action) {

// Problem: this only does a shallow copy!

let newState = {...state};

// ERROR: nestedState is still the same object!

newState.nestedState.nestedField = action.data;

return newState;

}

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:

function updateVeryNestedField(state, action) {

return {

...state,

first : {

...state.first,

second : {

...state.first.second,

[action.someId] : {

...state.first.second[action.someId],

fourth : action.someValue

}

}

}

}

}

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:

function insertItem(array, action) {

return [

...array.slice(0, action.index),

action.item,

...array.slice(action.index)

]

}

function removeItem(array, action) {

return [

...array.slice(0, action.index),

...array.slice(action.index + 1)

];

}

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:

function insertItem(array, action) {

let newArray = array.slice();

newArray.splice(action.index, 0, action.item);

return newArray;

}

function removeItem(array, action) {

let newArray = array.slice();

newArray.splice(action.index, 1);

return newArray;

}

The remove function could also be implemented as:

function removeItem(array, action) {

return array.filter( (item, index) => index !== action.index);

}

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:

function updateObjectInArray(array, action) {

return array.map( (item, index) => {

if(index !== action.index) {

// This isn't the item we care about - keep it as-is

return item;

}

// Otherwise, this is the one we want - return an updated value

return {

...item,

...action.item

};

});

}

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:

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:

var collection = [1, 2, {a: [12, 17, 15]}];

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.