19. State Management and Immutable Update Patterns

github repo

- Mutability vs. Immutability in JavaScript
- React Review
- Updating State of Varying Complexity
- useReducer

Mutability and Immutability in JavaScript

When it comes to assigning our variables, we should take time consider if the data-type we're working with is mutable or not—that is to ask: is this data-type handled as a primitive or by reference?

Primitives (Mutable Types)

As a reminder, JavaScript's *primitives* include:

- String
- Number (includes `NaN` and `Infinity`)
- Boolean
- `null`
- `undefined`
- Symbol

Let's consider the following examples:

```
let myString1 = 'Hello, World!';
let myString2 = myString1;

myString2 = 'Testing, 1 2 3!';

console.log(myString1, myString2); // 'Hello, World!' 'Testing, 1 2 3!'
```

```
let myNum1 = 3.14;
let myNum2 = myNum1;

myNum2 = 1994;

console.log(myNum1, myNum2); // 3.14 1994
```

```
let myBool1 = true;
let myBool2 = myBool1;

myBool2 = false;

console.log(myBool1, myBool2); // true false
```

Notice how in each of these examples we are able to say `string2 = string1`, and it stores the **VALUE**contained therein. This makes it easy to ensure that both `string1` and `string2` are kept separate and can be changed *individually*. This is in stark contrast to how more immutable, or reference-based types, are handled.

By-Reference (Immutable Types)

For more complex structures, JavaScript must handle things a bit differently. Consider the following types:

- Arrays
- Objects
- Functions

Each of these structures are quite complicated, and require more memory and processing to build, read, or manipulate. Because of this cost, the variable or function name, references a point in memory representing the data for that particular instance.

Let's experience this first hand via a couple of simple examples:

```
const array1 = [1, 2, 3, 4, 5];
const array2 = array1;

array2.push(6);

console.log(array1, array2); // [1, 2, 3, 4, 5, 6] [1, 2, 3, 4, 5, 6]
```

```
const object1 = {language: 'JavaScript', year: 1995};
const object2 = object1;

object2.ext = '.js';

console.log(object1, object2);
// {language: 'JavaScript', year: 1995, ext: '.js'} {language:
'JavaScript', year: 1995, ext: '.js'}
```

Notice how in both examples, `array1` and `array2` as well as `object1` and `object2`, stay in-sync with each other. This is because these names represent *the very same* array or object, respectively.

Is there a way to get around this in a pinch?

Yes! If you do want to create a fresh *copy* of an existing array to handle separately, there are a couple of options. One of the easiest is by use of the spread syntax (not to be confused with the similar-looking rest parameter.)

How do we execute this strategy? Observe the following:

```
const array1 = [1, 2, 3, 4, 5];
const array2 = [...array1]; // Use the spread operator to "spread" the
contents of the array into a new one via square brackets.

array2.push(6);

console.log(array1, array2);
// [1, 2, 3, 4, 5] [1, 2, 3, 4, 5, 6]
```

The `array2` variable, when assigned, is a brand new array. Note that you can spread arrays *and* objects. Be careful to only use this strategy when you need to, as it is more expensive (takes more CPU and RAM) than the alternative. Note this is a shallow copy, any nested arrays or objects will still be stored by reference.

If the array or object itself is many layers deep, and contains many references within, a heavier operation can be used instead. *Don't use this method unless you have to, as it will take up more resources than the above spread syntax*. We can convert the array, or object, into a JSON string—and then back into a JavaScript array or object. This forces a new reference to be created, but you will lose any methods or non-JSON compatible values in this process.

```
const array1 = [1, 2, 3, 4, 5];
const array2 = JSON.parse(JSON.stringify(array1)); // Convert to a JSON
string; convert back to a JS array.

array2.push(6);

console.log(array1, array2);
// [1, 2, 3, 4, 5] [1, 2, 3, 4, 5, 6]
```

As yet another option, you may craft your own, recursive, copier/cloning function.

React Review

Components, Props, and JSX—oh my!

Let's think back to vanilla JavaScript (using JS on its own without libraries.) Functions allow us to store instructions:

```
// Function declaration
// @link https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/
Functions

function HelloWorld () {
  // Logic goes here...
}

// Arrow function expression
// @link https://developer.mozilla.org/en-US/docs/Web/JavaScript/
Reference/Functions/Arrow_functions

const HelloWorld = () => {
  // Logic goes here...
};
```

If we'd like, a function is capable of returning a value. Let's suppose some hello text for our example:

```
const HelloWorld = () => {
  return 'Hello, World!';
};
```

To execute a function, we follow its name with parenthesis like so:

```
const HelloWorld = () => {
  return 'Hello, World!';
};

console.log(
  HelloWorld() // Display the return string in the console.
);
```

If we'd like to make the return content more dynamic, we could add a parameter to our function—essentially a variable or value that we can pass-in:

```
const HelloWorld = (name) => {
  return `Hello, ${name}!`;
};

console.log(
  HelloWorld('Sam') // 'Hello, Sam!'
);
```

This is very similar to the format we see components follow. Consider this React component, and how it is quite analagous to the previous examples:

```
const HelloWorld = (props) => {
  console.log(props); // If you're curious, never be afraid to
console.log()!
  return (
    <p>
      Hello, {props.name}! {/* We use curly braces to run JS expressions
in our JSX. */}
    </p>
  );
}

export default HelloWorld;
```

Instead of directly executing this component via parenthesis, it is common practice to invoke it via JSX syntax like so:

```
import HelloWorld from './HelloWorld';

const App = () => {
  return (
    <div>
      <h1>My App</h1>
      <HelloWorld name="Sam" /> {/* Props are passed in this way. */}
      {/* This would be equivalent to: HelloWorld({name: "Sam"}) */}
    </div>
  );
};

export default App;
```

Note how re-usable and repeatable components can be!

```
import HelloWorld from './HelloWorld';

const App = () => {
  return (
    <div>
      <h1>My App</h1>
      <HelloWorld name="Sam" />
      <HelloWorld name="Alex" />
      <HelloWorld name="Ryan" />
      <HelloWorld name="Quinn" />
    </div>
  );
};

export default HelloWorld;
```

The `useState()` Hook

In order for us to track data that may change over-time, we should always consider the `useState()` hook.

Without a tool like this, assume we make a change to a value. React won't know about that change, and we won't end up seeing the change in the browser! Not only is that boring, but it makes our application unresponsive and unreliable.

When we use this hook, we're letting React know to keep an eye on this value. When we make changes, it will be with intention, and we'll do it the React way; this way React will know to re-render the HTML elements containing that value so the user always sees the latest mutations of that data.

So. We know we want to show the user our latest data, let's have a look at how this hook operates. Let's create a basic counter component to demonstrate:

```
import {useState} from 'react';

const Counter = () => {
  const stateArray = useState(0);
  const getCount = stateArray[0]; // First array position is the GETTER
(read value.)
  const setCount = stateArray[1]; // Second array position is the SETTER
(write value function.)

  const clickHandler = () => {
    setCount(getCount + 1); // Set the count to one more than previous.
  };

  return (
    <section>
      <h2>Counter</h2>
      <p>Current count is: {getCount}</p>
      <button onClick={clickHandler}></button>
    </section>
  );
};

export default Counter;
```

In the above, you can see how `useState()` returns an array. The first item in the array is a way to read the current value. The second, is a function we can use to mutate the value. ***NEVER MUTATE STATE DIRECTLY.***Always use this provided function, as this is the only way that React will know it needs to re-render the HTML.

It is uncommon for developers to have three lines when implementing state, like we did above. We can use array destructuring to do this all at once.

```
import {useState} from 'react';

const Counter = () => {
  const [count, setCount] = useState(0); // Unpack value 1 and 2 from the
array and assign them right away.

  const clickHandler = () => {
    setCount(count + 1); // This is a concern, we are using "stale"
state; there is a more reliable way to implement this.
  };

  return (
    <section>
      <h2>Counter</h2>
      <p>Current count is: {count}</p>
      <button onClick={clickHandler}></button>
    </section>
  );
};

export default Counter;
```

Much cleaner! Another improvement is to use a callback inside of our `setCount()` function. A callback we pass in will receive the most up-to-date state value as an argument.

```
import {useState} from 'react';

const Counter = () => {
  const [count, setCount] = useState(0);

  const clickHandler = () => {
    setCount((previousValue) => {
      return previousValue + 1; // This works better, as it will always
be in-sync with latest changes.
    });
  };

  return (
    <section>
      <h2>Counter</h2>
      <p>Current count is: {count}</p>
      <button onClick={clickHandler}></button>
    </section>
  );
};

export default Counter;
```

React State and Immutable Types

Let's create a component that uses an array in its state. Recall that arrays are handled ***by reference***, meaning we have to be careful with them when making use of `useState()`. We ***NEVER*** want to manipulate the original, as React will not notice our changes and may not re-render our updated data. This means we must create a fresh array each time we want to set a new value. Observe the following example, it even incorporates a web form:

```
import {useState} from 'react';

const Toppings = () => {
  const [toppings, setToppings] = useState(['cheese']); // Keep track of all submitted toppings.
  const [newTopping, setNewTopping] = useState(''); // Keep track of what is in our form field.

  const addNewTopping = () => {
    setToppings((prev) => { // Ensure you create a NEW array! We never want to update the original.
      return [
        ...prev, // Spread old items.
        newTopping // Include new topping.
      ];
    });
    setNewTopping(''); // Clear the form field so it is easy to enter a new one.
  };

  return (
    <section>
      <form onSubmit={addNewTopping}> {/* Run our "addNewTopping" logic when the form is submitted. */}
        <label>
          New pizza topping:
          <input
            onChange={(event) => {setNewTopping(event.target.value); /* Update newTopping state. */}}
            />
        </label>
        <button type="submit">Add to Pizza</button>
      </form>
      {/* Next step? We'll want to show the toppings list! */}
    </section>
  );
};

export default Toppings;
```

Use the developer tools to confirm the above is updating state appropriately! Once you've confirmed, we should also find a way to easily output our array:

```
import {useState} from 'react';

const Toppings = () => {
  const [toppings, setToppings] = useState(['cheese']);
  const [newTopping, setNewTopping] = useState('');

  const addNewTopping = () => {
    setToppings((prev) => {
      return [
        ...prev,
        newTopping
      ];
    });
    setNewTopping('');
  };

  // Create a list of JSX for nicely formatted output.
  const mappedToppings = toppings.map((topping, index) => {
    return (
      <li key={index}>
        {topping}
      </li>
    );
  });

  return (
    <section>
      <form onSubmit={addNewTopping}>
        <label>
          New pizza topping:
          <input
            onChange={(event) => {setNewTopping(event.target.value);}}
            />
        </label>
        <button type="submit">Add to Pizza</button>
      </form>
      <ul>
        {mappedToppings} {/* Output the list here. */}
      </ul>
    </section>
  );
};

export default Toppings;
```

Let's try adding one more form field here... rename the component to `Pizza` so that it is more accurate in describing what our component is responsible for. We can handle related information in a single state. See the following adjustments:

```
import {useState} from 'react';

const Pizza = () => {
  // We'll store all the pizza information in one state object now.
  const [pizza, setPizza] = useState({
    toppings: ['cheese'],
    crust: 'thin' // Let's keep track of crust as well now.
  });

  // The state for our form field can stay put.
  const [newTopping, setNewTopping] = useState('');

  const addNewTopping = () => {
    // We'll have to be careful to update the toppings array properly
given our new structure.
    setToppings((prev) => {
      return { // Ensure we're creating a new object, this time.
        ...prev, // Spread the original contents of the object.
        toppings: [
          ...prev.toppings, // Because it is a shallow copy, we also need
to spread the toppings array.
          newTopping // Ensure we add the new topping to this array,
still.
        ]
      };
    });
    setNewTopping('');
  };

  const updatePizzaCrust = (event) => {
    setPizza((prev) => {
      return {
        ...prev,
        crust: event.target.value
      };
    });
  };

  // Create a list of JSX for nicely formatted output.
  const mappedToppings = toppings.map((topping, index) => {
    return (
      <li key={index}>
        {topping}
      </li>
    );
  });

  return (
    <section>
      <form onSubmit={addNewTopping}>
        <label>
          New pizza topping:
```

useReducer

According to React documentation, state is meant to be simple. It's not wrong to use an array, or an object as state. But the more complex our state gets, the harder it is to manage with setState. If our state becomes complex, we can always move everything into useReducer.

```
const initialState = [
  {key1: "value1"},
  {key2: "value2"},
]

// reducer -> centralize, custom defined logic to update state
const reducer = (state, action) => {
    switch (action.type) {
        case "CASE 1":
            // do something
            //action.data = {key3:"new State 1"}
            return [...state, action.data]
            // Here we are adding a new object to our state
        case "CASE 2":
            // do something else
            return "new State 2"
    }
}

const [state, dispatch] = useReducer(reducer, initalState)

// dispatch is the trigger switch to call reducer
```

With the useReducer, we can utilize custom defined logic by passing in an action type, and data to our dispatcher.

Resources

- Mutability vs Immutability in JavaScript
- Spread Syntax (`...`)
- `JSON.stringify()`
- `JSON.parse()`

- Python Tutor: Visualize code in Javascript
- `useState()` Hook
- `useReducer()` Hook

View on Compass | Leave Feedback