Fun With Destructuring

Posted At : May 16, 2019 2:52 PM | Posted By : Cutter
Related Categories: Development, AngularJS, ES2015, React, NodeJS, JavaScript

If you aren't compiling your JavaScript (well, EcmaScript...) code with Babel, you're probably missing out on some of the best features of the evolving language. On the other hand, if you're working in React or Angular every day, you've probably come across some of the most dynamic features. Destructuring is a prime example, but it's also important to understand the "gotchas".

Let's take a simple example to show you some of the power of destructuring.

```
const nodes = [{
 id: '0-1',
 label: 'Led Zeppelin',
 members: [{
  id: '0-1-1',
  label: 'Jimmy Paige'
 }, {
  id: '0-1-2',
  label: 'Robert Plant'
 }, {
  id: '0-1-3',
  label: 'John Paul Jones'
 }, {
  id: '0-1-4',
  label: 'John Bonham'
 }]
}];

const band = nodes[0];
const { label: bandName, members } = band;
const [leadGuitar, leadSinger, bassPlayer, drummer] = members;
```

So, what's it do? Let's break it down. I took the first *node* and assigned it to *band*. I then assigned the *bandName* and *members* variables from the *band*'s *label* and *members* values, respectively. Then, I took the first four items from my *members* array, and assigned each of them to a variable as well. This offers you a lot of power, simplifies your code, and can save some CPU cycles as well.

But, what happens if something doesn't exist? Say you had a *band* with no members? (That's a trick), or *members* but no *drummer*? In those cases the *members* or *drummer* variables would be *undefined*.

Now, let's talk about "gotchas". Here's a neat bit of syntactic sugar for you.

```
drummer = {...drummer, deceased: true};
```

Using the spread operator, with destructuring, we add a new key to the *drummer* object. But, wait...

We also **replaced** the drummer object. This is important. While using destructuring like this can be easy, and very effective, it **can** have consequences. If you needed to update *drummer* **by reference**, you just killed the reference assignment.

And, the above statement would error (as will the array example below). This is because we declared *drummer* (and *members*) using **const**. While we could adjust, add, or remove keys and values, we can't **replace** the variable. We would have to declare using *let* instead of *const*.

The same holds true when using a spread operator and destructuring when attempting to update an array.

```
members = [...members, { id: '0-1-5', label: 'Jason Bonham' }];
```

While the *members* array now has a fifth item, the **reference** to *band.members* is no longer valid, as you **replaced** the variable.

But, this is no big deal, unless you needed to update the *reference* to the original variable. As long as you're aware of this limitation, it's easy to fallback on other methods to update those references. Let's change our variable declarations a little bit, and retool this code to work for us.

```
const band = nodes[0];
const { label: bandName, members } = band;
let [leadGuitar, leadSinger, bassPlayer, drummer] = members;

Object.assign(drummer, {deceased: true});
members.splice(3, 0, {id: '0-1-5', label: 'Jason Bonham'}); // insert Jason in the drummer array position
[,,,drummer] = members; // and update the declaration
```

We switched our member variable declarations to *let*, so they can be replaced, updated the *drummer*, inserted a new member in the correct position, and updated the *drummer* reference to the new member.

This post only briefly touches on the power of destructuring, in modern EcmaScript. For a fantastic overview, **check out the MDN documentation**.