A Drip of JavaScript

Making Deep Property Access Safe in JavaScript

Originally published in the A Drip of JavaScript newsletter.

If you've been working with JavaScript for any length of time, you've probably run across the dreaded `TypeError: Cannot read property 'someprop' of undefined` and the similar error for `null`.

```
var rels = {
    Viola: {
        Orsino: {
            Olivia: {
                Cesario: null
            }
        }
    }
};

// Outputs: undefined
console.log(rels.Viola.Harry);

// TypeError: Cannot read property 'Sally' of undefined
console.log(rels.Viola.Harry.Sally);
```

The problem, of course, is that a `TypeError` immediately halts execution of your code. It's simple to deal with when you have predictable inputs, but when you need to access a deep object property that may or may not be there it can be quite problematic.

Sometimes you can solve this by merging with a default object, but at other times that doesn't make sense.

Often, what we really want is to be able to ask for a deep property and just find out whether it has a proper value. If the deep property's parent or grandparent is `undefined`, then for our purposes the property can be considered `undefined` as well.

Let's take a look at a solution:

```
function deepGet (obj, properties) {
    // If we have reached an undefined/null property
    // then stop executing and return undefined.
    if (obj === undefined || obj === null) {
        return;
    }

    // If the path array has no more elements, we've reached
    // the intended property and return its value.
    if (properties.length === 0) {
        return obj;
    }

    // Prepare our found property and path array for recursion
    var foundSoFar = obj[properties[0]];
    var remainingProperties = properties.slice(1);

    return deepGet(foundSoFar, remainingProperties);
}
```

The `deepGet` function will recursively search a given object until it reaches an `undefined` or `null` property, or until it reaches the final property specified in the `properties` array.

Let's try it out.

```
// Outputs: { Cesario: null }
console.log(deepGet(rels, ["Viola", "Orsino", "Olivia"]));

// Outputs: undefined
console.log(deepGet(rels, ["Viola", "Harry"]));

// Outputs: undefined
console.log(deepGet(rels, ["Viola", "Harry", "Sally"]));
```

Excellent!

Of course, we probably want to use this value in some way. And it's unlikely that `undefined` in itself will be all that useful.

```
var oliviaRel = deepGet(rels, ["Viola", "Orsino", "Olivia"]);
var sallyRel = deepGet(rels, ["Viola", "Harry", "Sally"]);

// Produces a pretty graph of Olivia's love interest
graph(oliviaRel);

// Tries to produce a graph of Sally's love interest
graph(sallyRel);
```

The problem here is that we have to explicitly handle `undefined` in our `graph` function. But what if we are using a third party library that doesn't check for `undefined`? We could use the "or" trick, like so:

```
graph(sallyRel || {});
```

But that's not very explicit about our intentions, and will also fail if `sallyRel` happens to be `false` or another falsy value like `0` or `""`.

Alternately, we could explicitly check for `null` and `undefined`.

```
if (sallyRel === undefined || sallyRel === null) {
    sallyRel = {};
}

graph(sallyRel);
```

But that seems unnecessarily verbose.

It would be much nicer if we could just specify a default value to return instead of `undefined`. So how would we do that?

```
function deepGet (obj, props, defaultValue) {
    // If we have reached an undefined/null property
    // then stop executing and return the default value.
    // If no default was provided it will be undefined.
    if (obj === undefined || obj === null) {
        return defaultValue;
    }

    // If the path array has no more elements, we've reached
    // the intended property and return its value
    if (props.length === 0) {
        return obj;
    }

    // Prepare our found property and path array for recursion
    var foundSoFar = obj[props[0]];
    var remainingProps = props.slice(1);

    return deepGet(foundSoFar, remainingProps, defaultValue);
}

sallyRel = deepGet(rels, ["Viola", "Harry", "Sally"], {});

// Will output a graph based on the empty object
graph(sallyRel);
```

Now we have a nice safe way to do deep property access and even get back a useful value when the property doesn't have one.

If you find this utility useful or interesting, I have open-sourced it on GitHub. I've even added some syntactic sugar so you can use a string-based property list, like `Viola.Harry.Sally`.

Have ideas for future drips? Is there some part of JavaScript that consistently gives you trouble? Drop me a topic suggestion.

Thanks for reading!

Joshua Clanton
