Published in Better Programming · Follow

Zeng Hou Lim · Follow

Sep 8, 2019 · 4 min read ★

4 Ways to Safely Access Nested Objects in Vanilla Javascript

How to access nested objects and write safer code

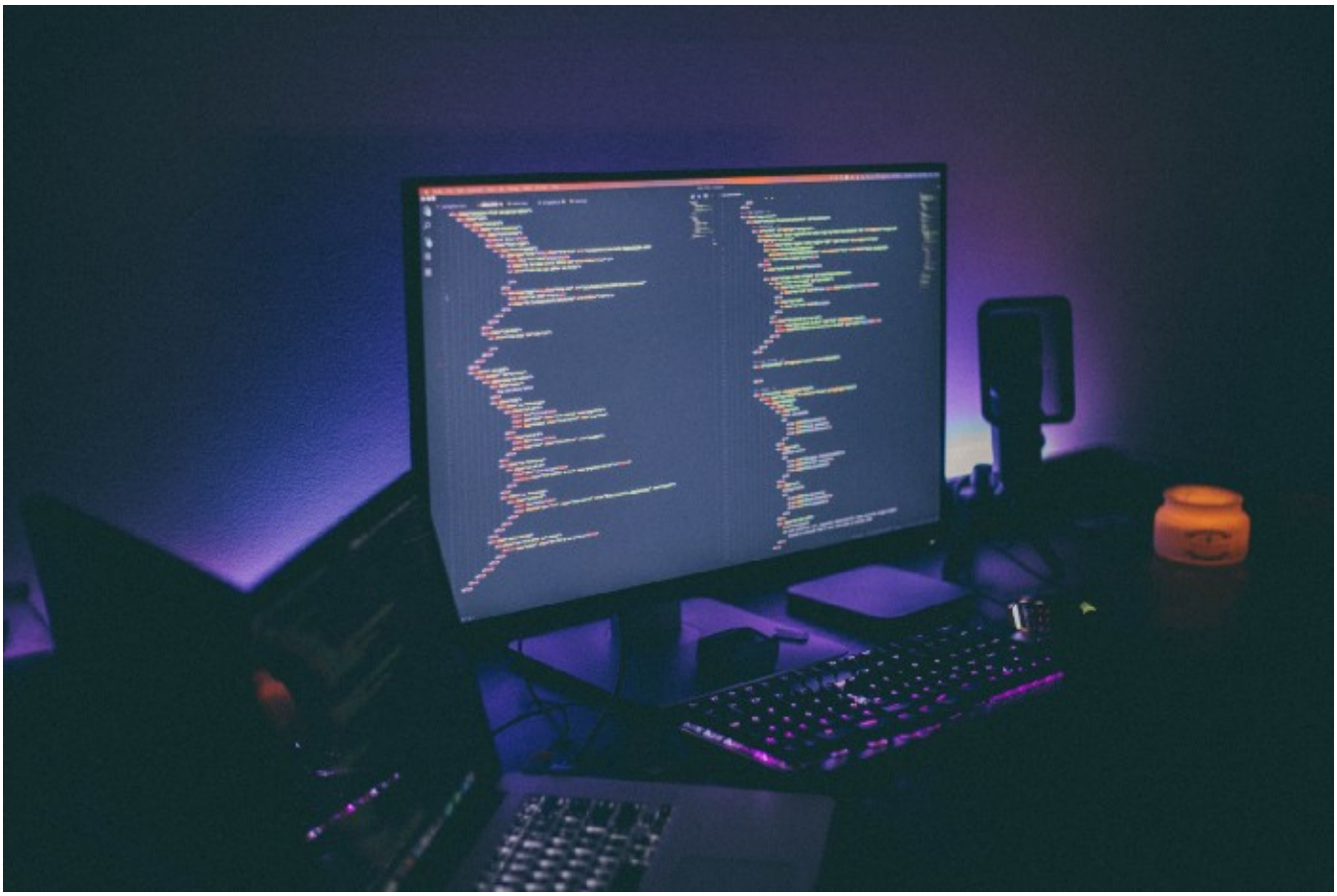

Photo by Blake Connally on Unsplash

If you're working with Javascript, chances are you might have encountered a situation

thinking: “How could that value not be populated?”

Thankfully, all we need to prevent these pesky errors is an additional check for `undefined` values.

Example of the Error

For those folks who are fortunate to not have seen this before, here’s an example to get you acquainted. Let’s say I have a JavaScript object, representing a music artist that I am interested in.

```
const macAyres = {
  tours: {
    nearMe: {
      sanFrancisco: {
        date: 'Sun Oct 27',
        location: 'The Regency Ballroom',
        cost: '30.00',
      },
    },
  }
}
```

If I had mistakenly assumed that this concert was in San Jose and wanted to retrieve the location of the (imaginary) concert, I might have typed something like this:

```
const concertLocation = macAyres.tours.nearMe.sanJose.location;
```

As the property `sanJose` does not exist, the expression above will be akin to something like `undefined.location`. Of course, we’ll be greeted with an all too familiar error; uncaught typeerror: cannot read property ‘type’ of undefined.

1. Ternary Operator to Check for null/undefined

```
const concertCity = macAyres.tours.nearMe.sanJose
const concertLocation = concertCity ? concertCity.location :
undefined;
```

This option is possibly the easiest to implement and check. When the object is not deeply nested, this could be a great way to check.

However, when we have to traverse several layers deeper, the checks can get pretty gnarly and repetitive.

```
const concertLocation = (macAyres.tours &&
macAyres.tours.nearMe &&
macAyres.tours.nearMe.sanJose) ?
macAyres.tours.nearMe.sanJose.location : undefined;
```

If you have a deeply nested object, the last two ways provide a more elegant way.

2. Oliver Steele's Nested Object Access Pattern

This method of accessing values is similar to what was described in the above point.

Instead of using the `&&` operator to check for `null/undefined` values, we use the `||` operator and an empty object literal. The example below will provide more clarity.

```
const concertLocation = (macAyres.tours.nearMe.sanJose ||
{}).location;
```

As the `||` operator breaks return the truthy value encountered, the above expression would return `macAyres.tours.nearMe.sanJose` if it is not `null/undefined`, otherwise `{}`.



Do also note that, as the `||` operator looks for the **first** encountered truthy value, we *should not* write an expression like this:

```
const concertLocation = ({} ||
macAyres.tours.nearMe.sanJose).location;
```

In this case, we will always fall back onto `{}` as it is considered a truthy value. If you would like to learn more about logical operators, this article provides an excellent rundown.

Similar to the method described above, even Oliver Steele's pattern can get increasingly unreadable with all the parenthesis if we needed to use `||` operators at every step of the way.

Fortunately, the next method shows how you can leverage a powerful JavaScript function to keep things clean.

3. Array Reduce

This method requires that you are familiar with JavaScript's built-in `reduce` function. If you do not already know about this, this is a great tool to add to your arsenal, and you can learn more about it at Mozilla.

In short, `reduce` executes a reducer function on an array and returns a single output. You can imagine that the array we'd be working with would be an array that contains, in sequence, the path that we'd traverse through the object.

We will use the above example data structure but, to demonstrate the capability of the `reduce` function, we will assume that each level of data is not guaranteed.



```
        location: 'The Regency Ballroom',
        cost: '30.00',
      },
    },
  }
}

const paths = ['tours', 'nearMe', 'sanJose', 'location'];
const location = paths.reduce((object, path) => {
    return (object || {})[path]; // Oliver Steele's pattern

}, macAyres)
```

Notice how we incorporated what we have learned earlier! If you prefer not to use the Oliver Steele pattern, the ternary operator check works fine too.

This way of accessing values is one of my personal favorites as it remains highly readable and DRY, even if the object is deeply nested.

4. Try/Catch Helper Function With ES6 Arrow Function

I was doing some research and stumbled across this Stack Overflow post which I thought was really sleek!

I have not used this before but this neat way of doing this can be implemented as a utility method and be more reusable. After writing this point, I suspect that I might really like this the most.

```
// Code Snippet taken from the post
function getSafe(fn, defaultVal) {
    try {
        return fn();
    } catch (e) {
        return defaultVal;
    }
}

// use it like this
```

  

Conclusion

These are the four methods that you can use to write safer code and avoid getting that pesky error that breaks our code.

Although there are libraries that can help you overcome that, you might want to consider these plain JavaScript methods if you want a simple and fast way.

On a final note, I hope you picked up something useful from this article. This is my first time writing a technical piece and I hope people find value in it.