Validating a JavaScript Array

If you have a function that takes one more more arrays as arguments, don't assume arrays are what you will get. Validating a JavaScript array can help prevent unexpected errors in your code.

In this article you will learn how about validating a JavaScript array. There are many times when you only want code to execute on an array that is holding one or more values. To prevent any non-arrays from entering your code, we must implement some validation techniques. Many times, people don't consider the need to validate their code a high priority and it can be hard to find useful information concerning ways to validate an array.

Keep reading for several examples of validation techniques you can use in your code to prevent crashes and hangups. Each example builds off the last, and we will discuss what works, what doesn't, and why, to help you get a better understanding of what's going on behind the scenes.

Example 1

```
// example # 1: check if the length property of the passed-in array is 0

function processRecords (records) {
    if (records.length === 0) {
        console.log(`No records to process`);
        return;
    }

    records.forEach(record => {
      console.log(`The record is: ${record}`);
    });
}

processRecords(['foo', 'bar', 'baz']);

// The record is: foo
// The record is: bar
// The record is: baz

processRecords([]);
// No records to process
```

validate-array-example-1.js hosted with ♥ by GitHub view raw

In Example 1, we are checking to see if the length of the array is equal to 0.

In this example, you can see two separate calls to the function named **processRecords**. One call is at line 14, and one is at line 20.
The first time we call it, we include a three-element array for the function to process. The second time we call processRecords, we send no array.

- The first time we call processRecords, we assign three elements (foo, bar, and baz) to the array **records**. This array passes to the function where it is first used by the **if** statement. The if statement uses the **length** property of the records array to make sure that the array is not empty. In this case, the length is equal to three. Since the Length is not equal to zero, we skip the rest of the if statement and move on to the **forEach** statement. The forEach statement then uses **consol.log** to print each element in the array to the screen.
- The second time we call processRecords, we don't assign any values to the records array. This empty array passes to the function where it is met by the if statement. The if statement uses the length property on the records statement, and sees that it is equal to zero. Since the length of the records array is zero, the if statement uses console.log to print the message "No records to process" before ending the processRecords function.

The if statement uses the length array to validate a JavaScript array. If we didn't include this validation step in our code and we passed an empty array, the forEach statement wouldn't fire, and the code would run and end without anything happening. We would have no way of knowing if the code executed at all.

Example 2

```
1  // example # 2: check if the length property of the passed-in array is "truthy"
2
3  function processRecords (records) {
4      if (!records.length) {
5          console.log(`No records to process`);
6          return;
7      }
8
9      records.forEach(record => {
10       console.log(`The record is: ${record}`);
11     });
12 }
13
14 processRecords(['foo', 'bar', 'baz']);
15
16 // The record is: foo
17 // The record is: bar
18 // The record is: baz
19
20 processRecords([]);
21 // No records to process
```

validate-array-example-2.js hosted with ♥ by GitHub view raw

In Example 2, we are checking to make sure the array is not empty by using something called Truthy Falsy.

This example is very similar to Example 1, but it uses a little less code and is more efficient. In the last example, we had to check to see if the length of the array is equal to zero because an array with zero elements is empty.

This time we use the code **!records.length** to validate the array, which means "if the array named records has no length." If we wrote **records.length** instead, we would be saying "if the array named records has any length."

This technique is truthy falsy because other values can mean the array is empty besides zero, like undefined, and NULL.

Example 3

```
// example # 3: If we pass-in an object with a lenght property, the code throws an error

function processRecords (records) {
    if (!records.length) {
        console.log(`No records to process`);
        return;
    }

    records.forEach(record => {
      console.log(`The record is: ${record}`);
    });
}

processRecords({length: 3});

// Uncaught TypeError: records.forEach is not a function
```

validate-array-example-3.js hosted with ♥ by GitHub view raw

In Example 3, we demonstrate how truthy falsy has some drawbacks. The main disadvantage is that we can check to see if the length is truthy, but not if it's an array. In this example, we are going to attempt to pass an object to the processRecords function instead of an array.

If we run this code, you will see if the records object makes it past the if statement that checks to see if records length is truthy, which it is because it contains the length property of the object. When we get to the forEach loop, it causes an error, and the code breaks because there are no array elements to iterate through, only an object. Therefore, you must exercise care when using this technique to ensure that only an array will get passed.

Example 4

```
// example # 4: Make sure that the passed-in object is an instance of the Array constructor

function processRecords (records) {
    if (!(records instanceof Array)) {
        console.log(`Array provided array not valid`);
        return;
    }

    if (!records.length) {
        console.log(`No records to process`);
        return;
    }

    records.forEach(record => {
      console.log(`The record is: ${record}`);
    });
}

processRecords({length: 3});
// Array provided array not valid
```

validate-array-example-4.js hosted with ♥ by GitHub view raw

In Example 4, we are going to look at how we can fix the code in Example 3 to check that the variable named records is an array and not an object.

This time, the first thing we do in the processRecords function is to use an if statement to check to see if records is an array. We use truthy falsy again and the expression instanceof for this task. It reads **!(records instanceof Array)**, and it means "if the value of records is not an instance of an array."

If records turns out to be an object or some other non-array, the if statement will use consol.log to print an error message and exit the function. Otherwise, it will continue with the code from Example 3 to make sure the array is not empty, and then iterate through the elements.

Example 5

```
// example # 5: Demonstrating how the array validation now works better, regardless of the argumer

function processRecords (records) {
    if (!(records instanceof Array)) {
        console.log(`Array provided array not valid`);
        return;
    }

    if (!records.length) {
        console.log(`No records to process`);
        return;
    }

    records.forEach(record => {
      console.log(`The record is: ${record}`);
    });
}

processRecords({length: 3});
// Array provided array not valid

processRecords(true);
// Array provided array not valid

processRecords('hello');
// Array provided array not valid

processRecords(12345);
// Array provided array not valid

processRecords(undefined);
// Array provided array not valid
```

validate-array-example-5.js hosted with ♥ by GitHub view raw

In Example 5, we use the same code as in Example 4, but this time, we'll try passing different things to the function to see how our code reacts. In each case, our first if statement catches the non-arrays before they break the code. Without the first if statement, each of these next examples would pass through our second if statement, and go on to break the code.

- The first thing we pass is the object with the length property from Example 3. This time though, instead of an error message, our code catches the object.
- The next thing we pass is a boolean value.
- Next is a string
- Next is a value
- Finally, we leave the value undefined

Example 6

```
// example # 6: Leverage Lodash to validate the array

function getScript(source, callback) {
    var el = document.createElement('script');
    el.onload = callback;
    el.src = source;

    document.body.appendChild(el);
}


function processRecords (records) {
    if (!_.isArray(records)) {
        console.log(`Array provided array not valid`);
```

```
        return;
    }

    if (!records.length) {
        console.log(`No records to process`);
        return;
    }

    records.forEach(record => {
      console.log(`The record is: ${record}`);
    });
}

function onScriptLoad() {
    processRecords (['foo', 'bar', 'baz']); // The record is: foo, The record is: bar, The record

    processRecords ([]); // No records to process

    processRecords ({length: 3}); // Array provided array not valid
}

getScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.15/lodash.core.js', onScriptLoad)
```

validate-array-example-6.js hosted with ♥ by GitHub view raw

Example 6 is a more sophisticated solution to validating a JavaScript array. This example uses an external JavaScript library called Lodash. Lodash is a powerful library that is too large for these simple examples, but if you are working with large chunks of code, this tool is beneficial and can save you time and typing over the long run.

In Example 6, we see a perfect example of how helpful Lodash can be. With this library loaded, we can use **.isArray** to make sure only arrays get passed to the rest of the function. It's similar to instanceof but more precise.

Example 7

```
// example # 7: Leverage the native isArray method to validate the array

function processRecords (records) {
    if (!Array.isArray(records)) {
        console.log(`Array provided array not valid`);
        return;
    }

    if (!records.length) {
        console.log(`No records to process`);
        return;
    }

    records.forEach(record => {
      console.log(`The record is: ${record}`);
    });
}


processRecords (['foo', 'bar', 'baz']); // The record is: foo, The record is: bar, The record is:

processRecords ([]); // No records to process

processRecords ({length: 3}); // Array provided array not valid
```

validate-array-example-7.js hosted with ♥ by GitHub view raw

Example 7 is the final example on our list. This example uses the JavaScript native expression **isArray**. You can see how this expression cleans up the code compared to the last example, and it's a more natural way to write validation code.

The problem with isArray is that it's relatively new and is only now supported by most browsers. Therefore, many developers shy away from this method for the time being, while there are still plenty of legacy browsers in use.

Conclusion

I hope that you have enjoyed reading over our examples, and we hope they have helped you get a better grasp of validating a JavaScript array. Validation will keep your code free of errors and unexpected crashes. We recommend staying with instanceof Array as seen in Example 4, for any website work and other projects that will see traffic from a lot of different browsers. At least for a few more years. If your projects are on a smaller scale and only run on modern machines, we recommend the native isArray expression to do the job.

If you have any questions or comments, please let us know and please share these validation techniques on Facebook and Twitter.