Hue University Journal of Science: Techniques and Technology
ISSN 2588–1175 | eISSN2615-9732
Vol. 128, No. 2B, 2019, 5–16; DOI: 10.26459/hueuni-jtt.v128i2B.5104

JAVASCRIPT ASYNCHRONOUS PROGRAMMING

Tran Thanh Luong[1]*, Le My Canh[2]

[1] Office for Undergraduate Education, University of Sciences, Hue University
[2] Department of Information Technology, University of Sciences, Hue University

Abstract. JavaScript has become more and more popular in recent years because its rich feature set as being dynamic, interpreted and object-oriented with first-class functions. Furthermore, JavaScript is designed with event-driven and I/O non-blocking model that boosts the performance of overall application especially in the case of Node.js. To take advantage of these characteristics, many design patterns that implement asynchronous programming for JavaScript were proposed. However, choosing a right pattern and implementing a good asynchronous source code is a challenge and thus easily lead into robustless application and low quality source code. Extended from our previous works on exception handling code smells in JavaScript, this research aims at studying the impact of three JavaScript asynchronous programming patterns on quality of source code and application.

Keywords: Keyword: javascript; asynchronous programming; code smell; error handling.

1 Asynchronous error handling by callbacks

1.1 The error-first callback pattern

The very first JavaScript asynchronous error handling pattern is the "error-first callback" pattern, which was introduced in Node.js. This is a standard for callback throughout Node.js. Almost all of I/O operations are supported asynchronous programming by using this pattern. To define an error-first callback, there are two rules to be followed [1]:

- *The first parameter of the callback is reversed for an error object*. If the requested asynchronous operation was failed, it would provide an error object as the first argument of the callback.
- *The successful response data is the second parameter of the callback*. If the requested asynchronous operation did not encounter any error, the first argument will be set to null and the second argument will be the result of the operation.

```
fs.readFile(filePath, function(err, data) {
    if(err) { throw err; }
    data = JSON.parse(data);
});
```

* *Corresponding: ttluong@hueuni.edu.vn*

Received: 14–01–2019; Reviewed: 16–02–2019; Accepted: 25–02–2019

When the listing code above is executed, the V8-engine (the engine to run JavaScript in Node.js) will start a worker thread for file reading asynchronously. If the reading process failed, the callback function will be invoked with the err parameter is not null. This parameter now contains the error of the file reading process. In the opposite case, err will be set to null and the data contains the content of the requested file.

1.2 Callback advantages

Using the error-first callback pattern may bring us some benefits:

- Easily getting notified if an error occurs by checking the first parameter.
- Consistently using the pattern to improve the readability of source code.

As it has been shown in the listing code of previous section, by checking if the first argument is null, we can easily determine whether the asynchronous operation was successfully returned or not. Since most JavaScript programmers already familiar with using callback, this pattern helps them conform easily to Node.js asynchronous programming. It is also clear that, consistently using this pattern throughout the project will significantly improve the readability of source code. Nevertheless, this is the very first asynchronous error handling so it has many disadvantages that will be analyzed in next section.

1.3 The downsides of using callbacks

Although using this pattern is quite easily, in some cases many nested or complex asynchronous operations may be required, and the source code may run into callback hell [2], or become very complex causing hard to read, debug and maintain.

```
fs.readFile("./f_path.txt", "utf8", function(err, data) {
    if(err) { throw err; }
    fs.readFile(data, "utf8", function(err, data) {
        if(err) { throw err; }
        console.log(data);
    });
});
```

With this code listing, we can see that an asynchronous operation is calling in a callback, and we need to submit a callback to this operation, resulting in a nested-callback. In some other cases, this happens with multi-level nested callbacks, which is callback hell. Although we can define an outer function for the inner asynchronous file reading, and then call this function inside that outer callback, in the case of multi-level callbacks, doing so will lead to less readable source code, and cause more troubles in debugging.

2 Asynchronous programming and error handling using promise

2.1 JavaScript asynchronous programming with promise and corresponding error handling mechanism

In recent years, prior to the ECMAScript officially supports promise, many developers have already used some open source libraries that supported promise like q [3] or promisejs [4]. Beginning from ECMAScript 6, JavaScript has already had built-in promise [5], therefore, more and more developers tend to use promise as a better strategy to implement asynchronous programming instead of callback.

A promise as three states:

- *Pending:* the asynchronous operation is not completed yet.
- *Fulfilled:* the asynchronous operation complete successfully.
- *Rejected:* the asynchronous operation may encountered error and failed, or was rejected explicitly.

To create a promise, the promise constructor accepts a function which has the following form:

```
new Promise( /* executor */ function(resolve, reject) { ... } );
```

The two parameters resolve and reject are functions that have one parameter. In executor function, to indicate that the asynchronous operation has completed successful, we invoke the resolve function with argument is the result of the operation. In the other case, if there is any error which has occurred or the operation cannot compete successfully, we call the reject function with the argument as the error. We can chain promises by using function then. This function has two parameters which are the callbacks that corresponding to the cases of successful return or failure of asynchronous operation. These functions are also single-parameter functions. onFulfilled function will be called when the parameter is the data that returned. In case of failure, onRejected will be invoked with the parameter set to the error of asynchronous operation.

```
p.then(onFulfilled[, onRejected]);
```

2.2 Benefits of using promise

The advantages of using promises include:

- Eliminating callback hell,
- Improving readability of source code, and
- Being easier for debugging.

The previous file reading example can be rewritten by using promise as bellow:

```
function readFilePath() {
    return new Promise(function (resolve, reject) {
        fs.readFile("./f_path.txt", "utf8", function (err, data) {
            if (err) reject(err);
            resolve(data);
        })
    });
}
function readFileContent(filePath) {
    return new Promise(function (resolve, reject) {
        fs.readFile(filePath, "utf8", function (err, data) {
            if (err)  reject(err);

            resolve(data);
        });
    });
}
readFilePath()
    .then(readFileContent)
    .then(function(data) {
        console.log(data)
    }, function (err) {
        console.log("An error has occurred!")
    });
```

As it is clearly shown above, we already eliminated all nested callbacks by using independent functions and then chained all promises by then. If we have many sequential asynchronous operations, simply put them into sequential thens. Because all of thens are at the same level, the source code now is similar to synchronous code, thus increase the readability, maintainability significantly.

Error handling with promise now is easier, likely using try catch in synchronous programming because all the asynchronous operation now are at the same source code level by then. With every then, developer can provide corresponding onReject callback for error handling. If an error occurred and this onReject callback is absent, then function will return a promise with Rejected state. Developer can add a onRjected callback at the end of the then chain for error handling for all over the chain.

2.3 Missing global promise rejection to handle code smell

Identification

Although using promise makes error to be handled more easily, some error handling code smells may still happen with even experienced developer. Beside the two errors handling code smells we analyzed in [6]: "*Error swallowing in onRejected handler"* and "*Missing error handling at the end of promise chain"*, in this paper we study a new error handling code smell: "*Missing global rejected promise to handle code smell"*.

At the server side - Node.js, application runs on a single process by default. This process will be terminated on any uncaught exception or unhandled promise rejection (UEUJ for short). This is troublesome because we almost run Node.js application as a web server. Certainly, application will always have exceptions and has a high probability, some of them could become an UEUJ. In addition, any UEUJ will bring down the whole application. After that, the application cannot serve any further incoming requests. Again, this can be seen as having poor software quality. The same can be said about the client side. As analyzed in [7], related to uncaught exception, we already identified the exception handling code smell called *No Global Uncaught Exception Handler*. In case of promise, *Missing Global Promise Rejection Handling* is a similar exception handling code smell.

Refactoring

Implement Global Unhandled Promise Rejection Handling. Two strategies are proposed identical to the server side (in Node.js) and the client side.

a. For server-side Node.js application

Any unhandled promise rejection will lead to identical uncaught result exception: take down the server. Implement a global unhandled promise rejection to make log, notify the administrator and restart the server.

From Node.js v1.4.1, developers can implement a listener for unhandledRejection event of process, as demonstrated in the following example. The first argument of the handler is the rejection reason (the rejection value from the promise, usually an error object), and the second one is the promise that was rejected.

```
process.on(unhandledRejection, function(err, promise) {
  /* log the err and notify administrator */
  /* Code to restart the process */
});
```

b. For client-side JavasScript application

An unhandled promise rejection from a JavaScript program may lead client-side web application to unexpected behaviors. Implement a global error handler to catch all unhandled promise rejections, log them on the server for developers to debug and possibly reload the application [7].

```
window.addEventListener('unhandledrejection', function (event) {
  //log error to server and may reload web application
}); //OR
window.onunhandledrejection = function (event) {
  //log error to server and may reload web application
});
```

Different from Node.js implementation that passes the rejection reason and the rejected promise to the event handler individually, the event handler for browsers will receive a single event object that has the following properties:

- type is the name of the event (unhandledrejection or rejectionhandled, we do not consider the rejectionhandled in this paper but at a glance, this event will be fired when a promise is rejected and a rejection handler is called after one turn of the event loop).
- promise is the promise object that was rejected.
- reason is the rejection value from the promise.

However, the code above only runs on limited number of browsers: Google Chrome and Microsoft Edge. This is because the unhandledrejection event up to now is still in draft version of ECMAScript 9 [8]. Nevertheless, implementing these event handler in JavaScript still is a worth consideration.

Motivation

a. For server-side Node.js application

When an unhandled promise rejection occurs, it is not recommended to keep the process running. unhandledRejection is an event triggered away from the original source of the exception. All you get at this point is the rejection value and the rejected promise. Most likely no reference is available for returning to the context when the promise is rejected to clean up the application state or other resources [7]. As a result, it is best to exit the undergoing process and fork a new one. This would keep the server from going crashing and unexpected behaviors. Logging the rejection reason will help developers for debugging application and the source code now achieve robustness level G1 [9].

b. For client-side JavaScript application

A message of unhandled rejection is logged to browser's console window. However, it is at the user's browser. Developer will not be notified about that failure. A JavaScript application with no global error handler fails to achieve G1 since error information loses. Consequently, a global unhandled promise rejection to deal with unhandled rejection is necessary for error reporting. Since unhandled rejection event is triggered away from the context where the promise is rejected, we cannot process the rejected promise but report it to developers, may reload the application (automatically or manually by giving a recommendation to users).

3 Asynchronous error handling with async function

3.1 The new async/await keyword

Promise is a new method to write JavaScript asynchronous code in a sequential manner. Since ECMAScript 8, async function has been introduced [10] and this allows us to write asynchronous code even more synchronous-looking and corresponding to it is the new async/await keywords. This feature is actually built on top of promise.

To create an async function, we simply put the async keyword before the function definition [11].

```
async function asynfunc() {
  return “Hello Async function”;
}
```

The asyncfunc function now becomes an async function that always returns a promise. In case, the function returns a non-promise value, JavaScript will automatically wrap this value into a resolved promise. Therefore, the above example can be rewritten as the following listing code.

```
async function asynfunc() {
  return Promise.resolve(“Hello Async function”);
}
```

Since the async function always returns a promise, you can use this return value as a promise as usual:

```
asynfunc().then(console.log);
```

ECMAScript 8 also introduced the new keyword await. This keyword is only valid inside async function. Putting await keyword before a promise inside an async function, it will make the async function to return immediately (actually the function will return a promise as said above), thus JavaScript runtime can continue at the next statement right after the call of async function.

```
async function asyncfunc() {
    var promise_inside = new Promise((resolve, reject) => {
        setTimeout(() => resolve("done!"), 1000);
    });
    var result = await promise_inside;
    console.log(result); // "done!"
    return "async function return";
}
asyncfunc().then(console.log);
console.log("outside async function");

/* OUTPUT
outside async function
done!
async function return */
```

3.2 Improved asynchronous programming and error handling experience

The most important benefits of async/await feature can be listing bellow:

- Substantially increase readability, maintainability of asynchronous source code.
- Significantly simplify the error handling of asynchronous code.

Asynchronous error handling in a synchronous manner

As described in Section 2.1, we need to register an onRejected listener to catch any error that happens inside the promise or to handle when the promise is rejected. In addition, we cannot use the try/catch construction to catch these errors if the try/catch is out of promise [12]. With async/await feature, we can use try/catch construction to handle both synchronous and asynchronous handily. This is meaningful for developer since the source codes for error handling is completely synchronous while the task behind is asynchronous.

```
async function asyncfunc() {
    try {
        var promise = new Promise((resolve, reject) => {
            setTimeout(() => resolve("done!"), 1000);
            throw new Error("Error inside promise!"); //or reject(new Error("…"));
        });
        await promise.then(console.log);
    } catch (err) {
        console.log("Error caught " + err);
    }
}

asyncfunc();
//OUTPUT:
//Error caught Error: Error inside promise!
```

Finally, for debugging purpose, async/await with try/catch allow us to go step by step over the source code. From the above example, if we set a breakpoint inside any line inside the try block, when the breakpoint is hit, we can go step by step and can reach the statement inside the catch block. This is completely identical to the debugging process when we debug the synchronous source code. This is impossible for the promise version with onRejected event as listed below, we cannot reach the catch block by following step by step.

```
function asyncfunc() {
    var promise = new Promise((resolve, reject) => {
        setTimeout(() => resolve("done!"), 1000);
        throw new Error("Error inside promise!"); //or reject(new Error("…"));
    });
    promise.then(console.log, function (err) {
        console.log("Error final " + err); //not reachable in step by step debugging
    });
}

asyncfunc();
```

Neat asynchronous source code

Async/await feature makes the source code much cleaner, neater and increases readability significantly especially when it deals with multiple sequential asynchronous tasks. Moreover, asynchronous source code with async/await is completely synchronous-looking thus can also improve readability.

In the bellow example, we simulate a CPU time consuming task by creating a function that adds 10 to its input after 1 second. We need to do this task 4 times sequentially to sum 4 numbers with 10, and finally calculate the sum of these 4 results. In additional, for each step we need to add the step number to the return value of previous step before continue adding 10. The implemented source code is quite complicated, less readable. Besides this, callback hell appears with promise, but this is not always happen since in this example, we need to use closure for accessing the results of 4 steps at last.

```
function addTenAfterTenSecond(a) {
    return new Promise(function(resolve, reject) {
        if(a > 100) throw (new Error("input value must not be greater than 100"));
        setTimeout(() => resolve(a + 10), 1000);
    });
}
function addFourSteps(input) {
    return addTenAfterTenSecond(input).then(a => addTenAfterTenSecond(a + 1)
            .then(b => addTenAfterTenSecond(b + 1).then(c => addTenAfterTenSecond(c + 1)
                    .then(d => a + b + c + d))));
}

addFourSteps(50)
    .then(console.log) // 306
    .catch(console.log);
```

Using async/await features, the code now becomes synchronous-looking and much clearer. Furthermore, this version allows you for step by step debugging that is impossible with the previous one.

```
async function addTenAfterTenSecond(a) {
    return new Promise(function(resolve, reject) {
        if(a > 100) throw (new Error("input value must not be greater than 100"));
        setTimeout(() => resolve(a + 10), 1000);
    });
}
async function addFourSteps(input) {
    var a = await addTenAfterTenSecond(input); var b = await addTenAfterTenSecond(a + 1);
    var c = await addTenAfterTenSecond(b + 1); var d = await addTenAfterTenSecond(c + 1);
    return a + b + c + d;
}
addFourSteps(50)
    .then(console.log) // 306
    .catch(console.log);
```

3.3 The async/await hell code smell

Using async/await completely eliminates callback hell and allows us to write synchronous-looking code for asynchronous tasks. However, careless using this feature may lead into another code smell: async/await hell.

Identification

When using async/await, 2 statements have been await before function calls, however they are responsible for 2 tasks that does not depend on each other. However, in this case, the later statement still needs to wait the previous one to complete thus lead into longer total execution time.

```
async function initBlog() {
    var categoriesData = await getCategoriesData();
    var postsData = await getPostsData();
    settingUpUI(categoriesData, postsData);
}
initBlog();
```

The JavaScript snippet above will load the data of a blog from database then show up the application UI when all data are available. Supposing accessing database for reading categories and posting information are time consuming tasks. Moreover, reading data for posts does not depend on reading data for categories. Although the code above works but it will take more time since we need to wait for retrieval category information to finish then start reading data for post. In fact, we can execute these tasks concurrently.

Another example for async/await hell is shown below. We loop through all categories and get all posts belong to each. Although getting posts of each category can execute concurrently, this source code still runs one after another sequentially.

```
for (var i = 0, len = categoriesData.length; i < len; i++) {
    await getPostsOfCategory(categoriesData[i].id);
}
```

Refactoring.

You have a async/await hell, apply the following strategies to remove the code smell:

- Group dependent statements in async functions then execute those concurrently.
- Start the function call without await and then await the returned promise.
- Use Promise.all.

In case of the initBlog example, we can separate the call of async function and the await statement. When we call the async function without await keyword, the function start execute

and return a promise. However, after start 2 reading data functions, what all we have now is the two corresponding promises. Before showing up the UI, we must wait for those promises to finish. await 2 promise now will not make the reading tasks wait for each other since they already started.

```
async function initBlog() {
    var categoriesData = getCategoriesData();
    var postsData = getPostsData();
    await categoriesData;
    await postsData;
    settingUpUI(categoriesData, postsData);
}
initBlog();
```

Another solution to this scenario is Promise.all.

```
Promise.all(getCategoriesData(), getPostsData())
    .then(blogData => settingUpUI(blogData[0], blogData[1]));
// blogData[0]: resolved value of getCategoriesData
// blogData[1]: resolved value of getPostsData
```

To the case of for loop example, we first start the call without await keyword, then push all promise into an array, finally use Promise.all to wait for all to finish.

```
var promises = [];
for (var i = 0, len = categoriesData.length; i < len; i++) {
    var p = getPostsOfCategory(categoriesData[i].id);
    promises.push(p);
}
await Promise.all(promises);
```

Motivation

Async/await hell makes independent tasks to run consequently, decrease performance of application thus this can be considered as a poor quality software. Applying proposed refactoring method will eliminate this code smell, allowing independent tasks to run concurrently, improve speed of the software and increase the final quality of product.

4 Conclusion

In this paper, we analyzed three methods for asynchronous programming in JavaScript: using callback, promise and async/await feature. Using promise will eliminate callback hell that happens in case of using callback. It also increases the readability and maintainability of the source code. We also identified one error handling code smell: Missing Global Promise Rejection Handling code smell. This code smell make application fail to achieve robustness level G1 and leave no clue for debugging. ECMAScript 8 introduced async/await feature. Using this helps developer write synchronous-looking code while execute asynchronous code behind the scene. Async/await also allows us to step by step debug the application while promise cannot.

Moreover, A code smell related to async/await is also identified namely async/await hell. Corresponding refactoring method also is proposed to remove the code smell.

Our study also can apply for C# language with async/await feature, Java with CompletableFuture. For future work, we planned to research more about error handling and error handling code smell with async/await feature.

References

1. Fred K. S., "The Node.js Way - Understanding Error-First Callbacks", Available: http://fredkschott.com/post/2014/03/understanding-error-first-callbacks-in-node-js/, date accessed: 16/11/2018.
2. C. Hell, "Callback Hell", Available: http://callbackhell.com/, date accessed: 16/11/2018.
3. Kris K., "A Promise Library for JavaScript", Available: https://github.com/ kriskowal/q, date accessed: 22/11/2018.
4. Forbes L., "Promises", Available: https://www.promisejs.org/, date accessed: 22/11/2018.
5. ECMA International, "Standard ECMA-262, 6th Edition/June 2015, ECMAScript® 2015 Language Specification", 2015.
6. L. M. Canh and T. T. Luong, "Exception handling in Javascript asynchronous programming with Promise", Journal of Science and Technology, 2017.
7. C.-Y. Hsieh, L. M. Canh, H. Kim Thoa and Y. C. Cheng, "Identifcation and Refactoring of Exception Handling Code Smells", Journal of Internet Technology, Vol. 18, No. 6, pp. 1461– 1471, 2018.
8. ECMA International, "Standard ECMA-262, 9th Edition/June 2018, ECMAScript® 2018 Language Specification", 2018.
9. C.-T. Chen, Y. C. Cheng, C.-Y. Hsieh and I.-L. Wu, "Exception Handling Refactorings: Directed By Goals and Driven By Bug Fixing", The Journal of Systems and Software, Vol. 82, No. 2, pp. 333–345, 2009.
10. ECMA International, "Standard ECMA-262, 8th Edition/June 2017, ECMAScript® 2017 Language Specification", 2017.
11. Ilya K., "Promises, Async/Await", Available: https://javascript.info/async-await, date accessed: 22/11/2018.
12. Mozilla, "Concurrency Model and Event Loop", Available: https://developer.mozilla. org/en-US/docs/Web/JavaScript/EventLoop, date accessed: 18/11/2018.