Use of Asynchronous Methods to Enhance Javascript Code

Siddhant Gupta[1], Sagar D Padiya[2]

[1]B.E. in Information Technology, SSGMCE, Maharashtra, India
[2]Professor, Dept. of Information Technology, SSGMCE, Maharashtra, India

---***---

Abstract - *Since 1996, when JavaScript and its features were first released, there has been no stopping for it. Because of its pace, JavaScript is used to implement some of the language's features in almost all major technologies. Although the development of JavaScript was initially guided solely by implementation, the ECMA standardization process is now gaining traction. The asynchronous function provided by JavaScript is one such feature. Why use Asynchronous JavaScript when you can do whatever you want with standard JavaScript? In this article, we'll look at all of the benefits and enhancements that Asynchronous JavaScript offers over standard JavaScript. We'll look at how to make our web apps load quickly and are easily readable. We will systematically compare the old features such as callbacks, single threading with promises, async await, and error handling.*

***Key Words*: JavaScript, Asynchronous, async, await, promise, Callback**

1. INTRODUCTION

When we talk about making websites, the first thing that comes to mind is probably HTML and CSS and we can build a static website with HTML and CSS. For developers developing cloud, web, or IoT applications, JavaScript code running in the Node.js runtime is a big platform. The use of asynchronous callbacks and event loops to provide highly sensitive applications is a fundamental principle in Node.js programming. This programming model, though conceptually simple, contains several subtleties and behaviors that are described implicitly by the current Node.js implementation. We can include graphics, add colors and static content, make animations (2D/3D) but the websites will not be user responsive. User responsive means that the dynamic nature of the website will be absent. We'll not be able to submit form data, respond with different data every time a user clicks on a button. To make all of this possible, we use JavaScript. JavaScript is a fully fledged programming language capable of making websites, apps and games etc. JavaScript abbreviated as JS uses scripts to interact with the webpage.

Asynchronous JavaScript is a part of JavaScript and is considered fairly advanced bit of JS Language. To get a good grasp of Asynchronous JavaScript, we have to first understand the Synchronous section of JavaScript. JS is single threaded language and this means that only one process at a time can be executed by JavaScript. It starts from line one and executes all the lines of code one by one and all the process are stopped until that line of code is fully executed.

For example:

```
const button = document.querySelector('button');

button.addEventListener("click", () => {

  alert("CLICKED"');

  let paragraph = document.createElement("p");

  paragraph.textContent = "New paragraph added.";

  document.body.append(paragraph);

});
```

When we execute this bit of code, we will find that the alert () will be executed first and until and unless we terminate the alert, the paragraph element below the alert () will not run or we can say that the rendering is paused. Browsers allow us to run such operations asynchronously to solve certain issues. Promises allow you to start an operation (for example, fetching an image from the server) and then wait until the result is returned before continuing with another operation. Since the operation is happening somewhere else, the main thread is not blocked while the async operation is being processed. This was the problem that was faced by programmers all around the world and hence Asynchronous JavaScript was introduced. Asynchronous JavaScript has made life easier for programmers who wanted to run some kind of files from external resources such as fetching files from server, using the database, accessing the webcam, etc. Asynchronous means multiple things can be done at the same time or multithreading. In this paper, we will dive into the world of Asynchronous JavaScript and all the advantages it has to the regular way of writing scripts in JS. We will take a look at the traditional methodology of creating scripts and how we can improve them by using asynchronous methodology and how it improves the quality and readability of the code.

2. PROBLEM STATEMENTS

2.1 The Call Stack

Call stack is the mechanism by which the JS interpreter keeps track of its place in the script that calls multiple functions. This is the way JavaScript knows where it currently is in the code and if there are any functions called from that running function or not.

When we talk about stack, we think of an array. In the same way, when the script calls any function it adds the function to the call stack and then starts carrying out the operations. Any function that is called after the first one automatically gets added to the call stack and this goes on. When one function is finished, the interpreter removes it from the call stack and continues to execute the function after that. This goes on till all the function calls are addressed. When the stack takes up more space rather than the allotted space, then stack overflow error occurs. The Call stack has one major drawback that once there are a lot of functions waiting to be executed in the stack, it becomes difficult to identify which steps are going wrong (if any) at a certain point of time.

For Example:

```
const multiplication = (a, b) => {

return a * b;

};

const squared = (a) => {

return multiplication(a, a);

};

const rightAngleTriangle = (x, y, z) => {

return square(x) + square(y) === square(z);

};

rightAngleTriangle(2, 3, 5);
```

In this above example, when rightAngleTriangle(2, 3, 5) is called, rightAngleTriangle() function is called and the from inside it, square() is called and from it, multiplication() is called. These operations demonstrates call stack. The below Diagram Demonstrates the flow of code in the example stated.

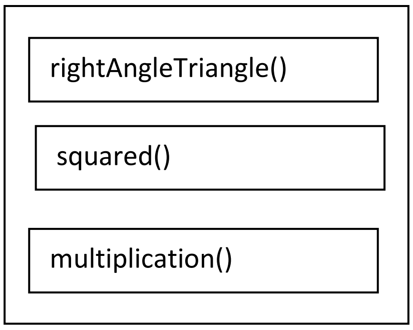

Chart -1: Call Stack flow Chart

This is the flow of the code and it is from top to bottom.

2.2 Single Threaded

JavaScript is a single threaded language. It means that at any point of time, only one particular line of code is running. In other words, multitasking is not possible here. Even if the computer having the JS interpreter has multiple cores, only one thread at a time will be executed and that is called as main thread. The best example of this is the alert () function which was discussed in the introduction. Until and unless the alert () function is executed, all the lines of code following the alert () are halted.

2.3 Callbacks

When there are multiple options for users to click or interact with the elements of webpage, it is hard for us to tell which element will be used by the user so for that in JS, we define callback functions which execute whenever any element in the webpage is interacted with. A callback function is a function that is passed as an argument to another function and then invoked within the outer function to complete a routine or operation. Callbacks are commonly used to create rich interactive Web applications in JavaScript's asynchronous programming. However, the lack of dependencies between callbacks can make it difficult to understand and maintain code, resulting in a mash-up of concerns. Regrettably, current JavaScript solutions do not adequately resolve this issue but this issue can be made more concise with the introduction of promises and that we will look further in this paper.

For example:

```
document.getElementById('button').addEventListener('click
',()=>

  {

    alert("CLICKED!")

  })
```

The second parameter in addEventListener() is the callback function for anything with id button is clicked and when the button is clicked, an alert will pop up stating that it's clicked.

A callback is simply a function which is passed inside another function as a value and it will only be executed when the event happens. This is because JavaScript has a concept of Higher-Order Functions which accepts other functions as values in parameters.

The main problem with callbacks is that it adds a level of nesting to functions. This is good when the nesting is just one or two levels but when the nesting is increased, callbacks substantially increases the amount of time needed to code and increases complexity of code.

For Example:

```
window.addEventListener('load',()=>{

document.getElementById('button').addEventListener('click
',()=>{

setTimeout(()=>{

items.forEach(item=>{

            //Function Body

    })

   }, 2000)

 })

})
```

To handle errors in callback functions, we pass the error object as the first parameter in the arguments of the function. If there is no error, the object will display the null value but if there's an error, the object will display the information about error.

```
fs.readFile('/file.json', (error, data) => {

if (error !== null) {

   // error handling

   console.log(error)

   return

 }

// process data if no errors

 console.log(data)

})
```

So these were some of the problems faced by programmers and to solve these problems, Modern JavaScript (ES6 or ECMAScript 6 and onwards) introduced some asynchronous features which will help us with these problems.

3. PROPOSED METHODOLOGY (SOLUTIONS)

3.1 Promise

A Promise is an entity that represents the successful or unsuccessful completion of an asynchronous process. Essentially, a promise is a returned object to which you attach callbacks, instead of passing callbacks into a function. Promises are a great way of not getting stuck in callbacks every time when we are having multiple function calls dependent on each other. It's a way to deal with asynchronous code. Programmers may prevent common event-driven programming pitfalls including event races and the highly nested counterintuitive control known as "callback hell" by using promises.

They were introduced in JavaScript in ES6 or ECMAScript 6 (also known as Modern JavaScript).

They syntax of creating promises is,

```
new Promise((resolve,reject)=>{

   //Function code

})
```

Let us consider a function that returns the data of a file:

```
function successReading(result) {

 console.log("file ready: " + result);

}

function failureReading(error) {

console.log("Error reading file: " + error);

}

readingFileAsync(readingFile,successReading,
failureReading);
```

If readingFileAsync() was to return a promise, we would follow the syntax:

```
readingFileAsync(readingFile).then(successReading,
failureReading)
```

Unlike callback functions nesting, promises give us some advantages, the callbacks added with then() will never be called until the current run of the JavaScript event loop is concluded and if these callbacks were introduced after the success or failure of the asynchronous process, they will be invoked. Multiple callbacks can be added by calling then many times and all of them will be invoked one after another.

For Example:

```
firstFunction(function(firstResult) {

   secondFunction(result, function(secondResult) {

      thirdFunction(newResult, function(thirdResult) {

         console.log('Got the final result: ' + thirdResult);

      }, failureCallback);

   }, failureCallback);

}, failureCallback);
```

If we try to code the above example using promises then it would look like this:

```
firstFunction()

.then(firstResult => secondFunction(firstResult))

.then(secondResult => thirdFunction(secondResult))

.then(thirdResult => {

   console.log(`Got the final result: ${thirdResult}`);

})

.catch(failureCallback);
```

The catch() function expects a function as an argument which would be executed when the code is rejected and not resolved in the code. Here failureCallback() is a function which will run if the promise is rejected.

In short, Promise accepts two parameters, one is resolve and another is rejecting. Resolve is executed when the request made by promise is successfully dealt with while reject is executed when the operation fails. When a Function uses a promise to resolve or reject a request, it has to use a special keyword called 'then' and 'catch' to execute the next line of code after a resolved or rejected request.

3.2 The async and await keyword

An async function is one that is declared with the async keyword and allows the use of the await keyword inside it. The async and await keywords make it easier to write asynchronous, commitment-based actions without having to specifically configure promise chains. Async keyword is used before a function to make a newer way of writing asynchronous code using Promise. The async function lets us use the await keyword for Promises. Await keyword is used to let the executing function run until the promise is resolved or rejected.

Syntax:

```
async function name([param[, param[, ...param]]]) {

   //statements

}
```

The params (parameters) of the async function are name of the function, Name of argument passed to the function, the function body (await keyword can be used).

A Promise that will either be resolved with the value returned by the async function or refused with an exception thrown by the async function or uncaught within it.

For Example:

```
const newAsyncFunc=()=>{

 return new Promise(resolve=>{

    setTimeout(()=>

       resolve('Promise Resolved')

    ,3000)

 })

}
```

This Function uses promise to resolve a request after a timeout of 3 seconds but when we execute this program, we find out that the code below this line gets executed even before the promise is resolved and the function is executed after 3 seconds. This caused lot of problems for programmers who wanted their promise to be resolved or rejected first.

Await solves this problem.

For Example:

```
const newResolve = async ()=>{

   console.log(await newAsyncFunc())

}
```

This newResolve() async function awaits for newAsyncFunc() to get executed and after 3 seconds when the request is resolved, the execution of code goes on.

There can be zero or more await expressions in async functions. Await expressions simulate synchronous behaviour in promise-returning functions by halting execution before the returned promise is fulfilled or rejected. The settled value of the promise is used as the await expression's return value. The use of async and await allows you to wrap asynchronous code in regular try/catch blocks (Blocks of code used to handle errors).

One important thing to note here is that await keyword can only be used with async functions. If we try to use the await keyword outside the scope of async functions, it will give us syntax error.

Now let us consider a code where we use promise with async functions.

An API returning a promise will result in a promise chain and the function is split into many parts.

```
function getData(url) {
  return downloadData(url) // returns a promise
    .catch(error => {
      return downloadFallbackData(url) // returns promise
    })
    .then(x => {
      return processData(x)  // returns a promise
    })
}
```

This code can be written using async function and await keyword.

```
async function getData(url) {
  let x
  try {
    x = await downloadData(url)
  } catch(error) {
      x = await downloadFallbackData(url)
   }
  return processData(x)
}
```

3.3 Error Handling

Error handling comes into practice when the promise is rejected instead of being resolved. As soon as the Promise is rejected, we see that an error is thrown by our browser. So code after that will never be executed. So the best way to catch errors in async functions is to use the try and catch block. Its syntax is:

```
try{
  //function code
} catch(e){
  // error catching code
}
```

When the function in try block is successfully executed, the catch never runs and hence we will never find an error. In the same way we can put our promises inside the try block with resolve function and if the request is rejected, the catch block will contain the code which will show the error. The main advantage of try and catch block in async a function is that the execution of script never stops. It will stop the execution of the part of code which has an error but all the correct code will continue to run.

4. CONCLUSION

We have concluded that, the Asynchronous JavaScript which was introduced in ES6 or the Modern JavaScript has significantly improved the life of programmers all around the globe with its newer syntax and new way of writing callbacks in form of promises and async functions. Asynchronous programming, which allows programmers to do more than one thing at a time, is becoming more common in modern software design. As you use more efficient APIs, you'll find more cases where asynchronous execution is the only alternative. This reduces the lines of code needed to be written before and thus reducing the time of execution making our web apps fast and more efficient. This becomes useful when script is very big and minor changes like these can make a very big difference in speed of the web app.

REFERENCES

[1] https://nodejs.dev/learn/javascript-asynchronous-programming-and-callbacks - Blog on JavaScript and Node.js
[2] https://developer.mozilla.org/en/US/docs/Learn/JavaScript/Asynchronous - Mozilla Developer Network Documentation on JavaScript.
[3] Gábor Antal, Péter Hegedus, Zoltán Tóth, Rudolf Ferenc, Tibor Gyimóthy in 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM), September 2018.
[4] Saba Alimadadi, Di Zhong, Magnus Madsen, Frank Tip in Proceedings of the ACM on Programming Languages, October 2018, Article 162.
[5] Saba Alimadadi in FSE 2016: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, November 2016, pages 1076-1078.
[6] Martin Bodin, Arthur Chargueraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, Gareth Smith in POPL '14: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 2014, pages 87-100.
[7] Matthew C. Loring, Mark Marron, Daan Leijen in DLS 2017: Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages, October 2017, Pages 51-62.
[8] Paul Leger, Hiroaki Fukada in MODULARITY Companion 2016: Companion Proceedings of the 15th International Conference on Modularity, March 2016, Pages 79-82