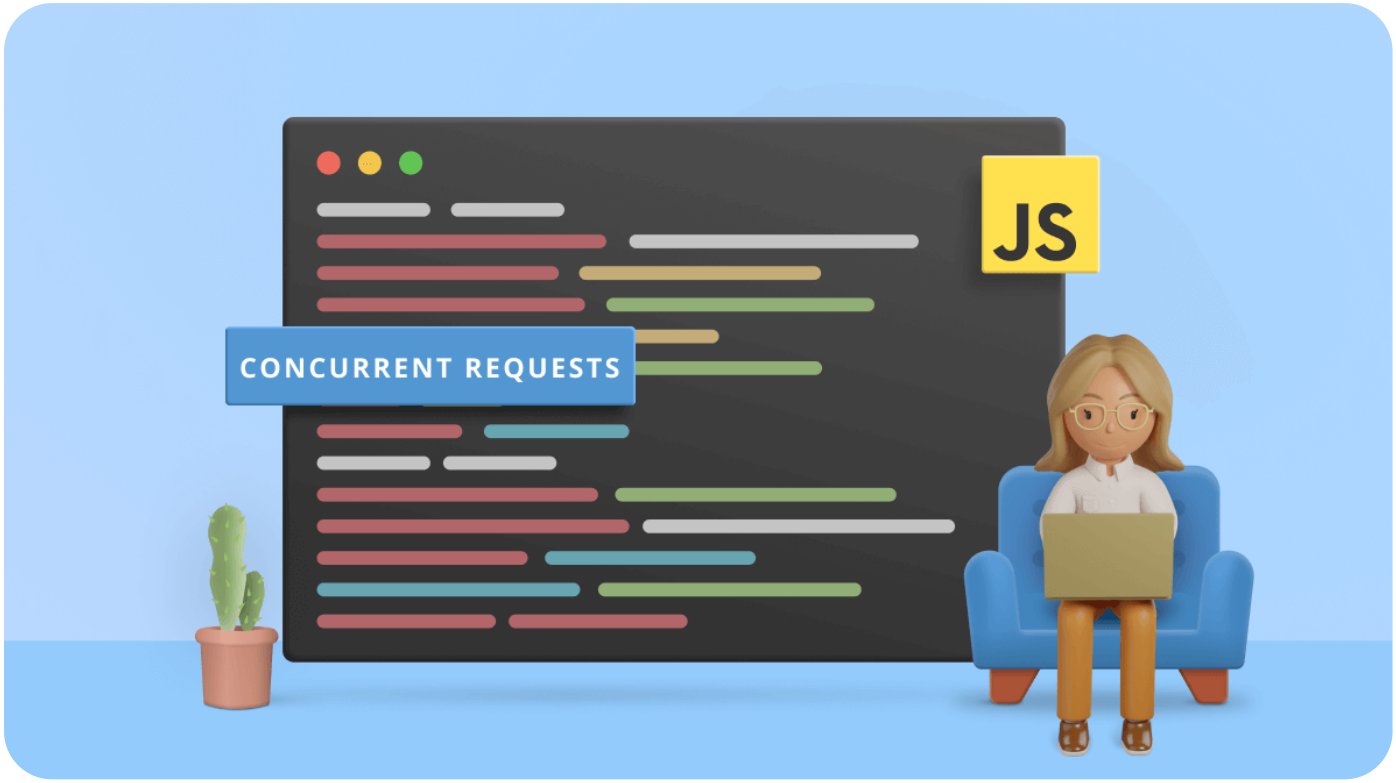
Handling Concurrent Requests with JavaScript Callbacks

Piumi Liyana Gunawardhana • 7 min read • Nov 19, 2024 • Updated



Concurrency is a common concept in modern web application development. It is the ability to execute multiple tasks simultaneously. Concurrent programming may appear complicated at first, but it helps us to enhance user experience, and make our web apps more dynamic and interactive.

Many programming languages provide flexibility and performance with concurrency. Despite being one of the most popular programming languages in the world, though, JavaScript was never designed for it. But with time, while the JavaScript event loop laid the groundwork, Node.js made JavaScript a proper server-side concurrency solution. The event loop, callbacks, promises, and async/await support in JavaScript make it possible.

This article will go through how we can handle concurrent requests with the use of plain JavaScript callbacks, along with their advantages, disadvantages, applicability, and best practices.

What is a callback?

The basic architectural building blocks for writing concurrent code in JavaScript are callback functions and AJAX service calls.

A function taken as an argument to another function and called within the outer function to complete a task or an event afterward is known as a callback function.

Therefore, a callback function is a piece of code that must be executed only after another piece of code with an indefinite duration has finished.

According to this definition, any function without a specific syntax passed as an argument can be a callback function, and these are not inherently concurrent. However, we can apply callback functions to handle concurrency in JavaScript.

Following is the way you can implement callback functions.

Copy

```
// A function.
function fnc() {
 console.log('A function')
}
// Higher order function: a function that takes another function as an argument.
function higherOrderFunction(callback) {
 // When you call a function that is passed as an argument to another function,
 callback()
}
// Passing a function.
higherOrderFunction(fnc)
```

Executing the previous code will give you the following output.

```
//Output
A function
```

Now, let’s see how we can use callbacks to handle concurrent requests.

JavaScript concurrency with callbacks

You can see the demonstration for handling concurrent requests with callbacks in the following example.

You know that **setTimeout()** is a built-in JavaScript asynchronous function that runs a function or analyzes an expression after a specified amount of time. It takes two parameters: a callback function and a milliseconds-long delay.

Consider the **setTimeout()** behavior given in the following example. It uses a callback function to call **setTimeout(),** which attaches **printOutput()** to the result object. We set the runtime delay for this as 500 milliseconds.

```
var delay = 500;

function printOutput(result) {

var runtime = Date.now() - result.start;

   console.log('execution', result.n, 'completed in', runtime, 'ms after a delay

   return runtime;

}

var result = { n:0, start:Date.now(), delay:delay };

var out = setTimeout( () => printOutput(result), delay );
```

In the previous code, it's worth noting that the **setTimeout()** function returns instantly, while the **printOutput()** function runs 500 milliseconds later. The value returned by **printOutput()** is entirely unrelated to the value returned by **setTimeout()** because that function will not be invoked until **setTimeout()** has generated its value for **out.** Though **setTimeout()** returns a valid result for **out,** since it is irrelevant to our discussion, the code will falsely state that the value of **out** is undefined.

When the previous code is executed, the **printOutput()function** will return a message including the exact time spent between the initiation of the code and its termination, along with the latency set by the **setTimeout()** function. Like in the following example output, these two durations are unlikely to be equal. Furthermore, the **printOutput()** method returns the exact runtime (in milliseconds), but this value is unnecessary.

```
execution 0 completed in 520 ms after a delay of 505
```

Now, create custom concurrent functions by encapsulating **setTimeout()** calls and with the use of callbacks. The concurrent behavior is based on the inclusion of unnecessary and random latency.

The **asyncFn()** in the following code is an asynchronous function that takes an integer **n**. When called, the function creates a random latency and publishes a message to the terminal once that time has passed. The number **n** is a label that notes how many times the **asyncFn()** has been called. Of course, **asyncFn()** immediately returns the value as undefined. Therefore, the value of the result will generally be undefined.

Copy

```
function asyncFn( n ){
   var delay = Math.floor( 100 + Math.random() * 900 );

function printOutput( result ){
   var runtime = Date.now() - result.start;
   console.log( 'execution', n, 'completed in', runtime, 'ms after a delay of',
   return runtime;
}
var result = { n:n, start:Date.now(), delay:delay }
  setTimeout( () => printOutput( result ), delay );
}
for( var rank = 0; rank < 5; rank++) {
   var out = asyncFn( rank );
}
```

When the loop executes, the asynchronous nature of this code is visible. The output provided by these five sequential calls does not match the sequence in which they are executed, despite the fact that the **asyncFn()** is called five times with rankings of 0 through 5 in order. Furthermore, the gap between delay time and actual completion time fluctuates among invocations.

The output will look as follows for one execution of the previous code.

```
execution 2 finished in 170 ms after a delay of 166
execution 0 finished in 185 ms after a delay of 184
execution 1 finished in 481 ms after a delay of 476
execution 3 finished in 519 ms after a delay of 518
execution 4 finished in 794 ms after a delay of 792
```

Although it shows concurrency, the code is inefficient because the **asyncFn():**

- Fails to convey the calculated output back to the caller.
- Assumes that the calculated output should be written to the console.

Functions should disclose their outputs to the caller and let the caller determine what to do with the calculated values. The callback functions offer a solution for resolving both of these design problems.

Let's now rewrite **asyncFn()** as a function that takes two arguments: an integer value **n** and a callback function **cb**. The callback function takes one argument, the **asyncFn**'s calculated output, and executes after a random delay of 100 to 999 milliseconds. The client code is therefore allowed to design its own callback function that:

- Receives the calculated output via the provided parameter.
- Handles that in any way the client desires.

In the following example, the callback function is declared in the client's context, allowing the client to access the calculated output through the callback's explicit parameter result.

```
function asyncFn( n, cb ) {
var delay = Math.floor( 100 + Math.random() * 900 );
var result = { n:n, start:Date.now(), delay:delay };
setTimeout( () => cb( result ), delay );
}
for( var rank = 0; rank < 5 ; rank++){
  function printOutput( result ) {
     var runtime = Date.now() - result.start;

     console.log( 'execution', result.n, 'completed in', runtime,
                  'ms after a delay of', result.delay);
     return runtime;
}
var out= asyncFn( rank, printOutput);
}
```

So, now you should understand how we can achieve concurrency via callbacks in JavaScript.

Even though the callbacks in the previous example are instantly executed, most JavaScript callbacks are associated with an event, such as a timer, an API call, or reading a file. If you utilize callbacks correctly, they might assist in making your code more manageable.

Advantages of callbacks

Callbacks will help you to:

- Make your code as simple as possible (less repetition).
- Improve abstraction so that you may use more generic functions that can perform a wide range of functionalities (e.g., sum, product).
- Enhance your code's readability and maintainability.

Disadvantages of callbacks

Callback functions have certain limitations, despite the fact that they provide a simple approach for dealing with asynchrony in JavaScript.

The drawbacks of callbacks are:

- The possibility of creating callback hells if callbacks are not used properly.
- Error handling is a challenge.
- You can’t use the **throw** keyword or return values with the return statement.

Conclusion

Concurrent code is better than sequential code because it is nonblocking and can accommodate several users or requests concurrently with minimal issues. This article walked through a demonstration of how to deal with concurrency using JavaScript callbacks. It’s up to you to decide where callbacks should be used to handle concurrent requests without ending up in callback hell.

I hope you found this article helpful and do share your experiences with callbacks in the comments section.

Thank you for reading!

Syncfusion Essential JS 2 is the only suite you will ever need to build an app. It contains over 65 high-performance, lightweight, modular, and responsive UI components in a single package. Download a free trial to evaluate the controls today.

If you have any questions or comments, you can also contact us through our support forums, support portal, or feedback portal. We are always happy to assist you!

Related blogs

- Top 7 JavaScript Object Destructuring Techniques
- 5 Different Ways to Deep Compare JavaScript Objects
- Easily Configure Syncfusion JavaScript UI Controls in Salesforce
- 10 JavaScript Naming Conventions Every Developer Should Know

MEET THE AUTHOR

Piumi Liyana Gunawardhana

Software Engineer | Technical Writer since 2020