WWW.IJRPR.COM

International Journal of Research Publication and Reviews

Journal homepage: www.ijrpr.com ISSN 2582-7421

Elimination of Callback Hell by Promise and then by Async-Await in JavaScript

Pratik Raj Verma[1], Dr. Vishal Shrivastava[2], Dr. Akhil Pandey[3], Dr. Krishan Kant Lavania[4]

[1]B.TECH. Scholar, [2,3,4]Professor
Information Technology, Arya College of Engineering & I.T. India, Jaipur
[1]erpratikrajverma@gmail.com, [2]vishalshrivastava.cs@aryacollege.in, [3]akhil@aryacollege.in, [4]krishankantlavania@aryacollege.in

ABSTRACT

In this article we will see what is callback function, what is callback hell and how we will get rid of from this Complex topic by modern JavaScript function using async await before moving to Async await we will see what is promise in JavaScript and how it is better than call back Hell and lastly we will see how we can execute our asynchronous code as a synchronous code and we will also see the difference between asynchronous and synchronous code

1. Introduction :

This article explores call back function, address callback hell in javascript , and introduces modern solutions like async/await. It discusses the superiority of promises over callback hell and explore executing asynchronous code synchronously. By examing these concepts, readers gain insights into managing asynchronous operation effectively in javascript

2. Literature Review:

2.1 Callback function:

callback function is a function which pass as a argument to another function what does it mean we will understand it as a real life example let suppose my mom wants to cook mutter paneer so my mother has a task but in kitchen there is no paneer so she tells me to go market and buy paneer and comeback so i have also a task which is necessary for my mother task which means my mother task is totally dependent on my task and we can say that my mother task is a parent task and my task is a callback task it means my mother task can't be completed before completion of my task here we will replace task by a function in JavaScript. like this in JavaScript there parent function and callback function when we pass callback function as a argument of parent function then parent function is totally dependent on completion of callback.

callback function is of two type first is named function and second is anonymous function for name function we will pass named of callback function as a argument of parent function and for anonymous function we will pass whole function as a argument of parent function. in modern JavaScript we we can replace anonymous function by Arrow function

what is Arrow function?

Arrow function is used to sort the code of anonymous function which is used as a callback function eg. let sqr= function (x) { return x*x; } this is anonymous function now we will convert this as a arrow function let sqr = (x)=>x*x; so, by Arrow function we reduce the length of code of anonymous function. note if Arrow function has multiple line of code then we will have to use curly bracket and return with it but we can ignore function keyword.

Eg of callback function let arr=[1,2,3,4,5,6]; console.log(arr.find(num => num === 6));

here find() function is parent function which Is giving each value of array arr to their callback function which is arrow function and inside arrow function value of arrow is checking with 6 if it is equal then it will return true else false and find() function will stop giving value to their callback function if it satisfy the condition.

2.2 Synchronous code:

synchronous code is the execution of code sequently what does it mean there is n line of code then each code will execute one after another code, initially our JavaScript is totally synchronous type language but there is a lots of problem in synchronous code when we deal with event function, server response, input-output response, in this condition our whole application will get freeze and stuck if we use synchronous code so that's why all event listner is a asynchronous type in nature.

Eg. Console.log(‘hello’); Console.log(‘bolo to ’); Console.log(‘kaise o’); Console.log(‘hii’);

This is synchronous code so each code is dependent to their previous code and run one after another. If any code is not running then whole code will not run. So this is disadvantage of this concept if we are dealing with event listner or any server respose task like api call by backend or server it will get freez to whole application. So we will use Asynchronous code for this type of problems.

2.3 Asynchronous code:

asynchronous code is totally opposite of synchronous code in asynchronous code any code will not depend on previous code, it means here is random execution of code so asynchronous code allow JavaScript to parallelly run the code let suppose there is event listener function and they require event by user and user is not performing any event then it will not get stuck the whole application it will send the event listener function to Browser by event loop and when user click to the specified button then that event function get popped and goes to event queue and then it will check our call stack and if call stack is empty then that event listener function will execute, actually this whole process is execute by event loop

what is event loop?

It is a concept used in JavaScript to handle with asynchronous code like event listener in event loop there is basically three component 1 is callstack 2 is browser 3 is event queue

eg. console.log('abcd'); let content=document.querySelector('.wrapper'); content.addEventListener('click',function(event) { console.log('this is para'+event.target.textContent); }); console.log('1234')

in this code first abcd will print and if user has clicked on wrapper then also 1234 will print then after this “‘this is para” it will print so here this is happening console.log ABCD will goes to call stack and it will print and then listener goes to call stack and then to Browser Handler it will wait till user is clicked on specific tag and then third code is console.log 1234 will execute and if user clicked specific tag then it will go to event queue then it will check that call stack is empty or not if call stack is empty then event listener function will execute that is it will print this is para 1 for asynchronous code event listener there is three component call stack Browser Handler event queue every asynchronous code will go through these three component and this will execute by loop and this actually synchronous code there is no need of event loop only event loop is required for asynchronous for their execution it is very important point is that after synchronous code our event listener has executed instead of this it will not execute till call stack is not empty so, for running asynchronous code it is very important that call stack is empty.

3. Callback Hell:

It is basically a situation occur in JavaScript when nested callback function happens one call back is dependent to another call back actually it is a situation when we want to use a synchronous code like a synchronous code.

what does it means?

It means we want to execute asynchronous Code after the completion of previous asynchronous code and so on. so for doing this we have to make asynchronous code and inside that asynchronous code we have to make another asynchronous code and inside that asynchronous code we have to make another asynchronous code and so on and this will become very complex and this is called callback hell.

why it is called call back Hell ? it is call call back Hell because as we know that all asynchronous code is totally based on callback function and for making asynchronous code as a synchronous code we will have to make multiple callback function inside previous callback function and so on and whole code will look like nested callback function and it looks like callback hell. call back hell makes our code very bulky and it become very hard to understand which code is dependent to another code and which will execute first if there is 1000 line of code

setTimeout(() => { console.log('login first page'); setTimeout(() => { console.log('login second page') setTimeout(() => { console.log('login third page'); }, 2000); }, 2000); }, 2000);

This is the example of callback hell here each code is async code and executing like synchronously like a synchronous code it means each async code is dependent to their previous code completion here one logical question should arise in our mind that JavaScript is a synchronous language then what's the requirement is that we have to make asynchronous code as a synchronous code actually what will happen if we make asynchronous code line by line it should run synchronously right one after another.

actually in call stack it will run synchronously but when it will goes to Browser Handler then which code requires less time will execute first here is no matter check that which code has ordering first it will depend on user like if user click on that it will execute there is no dependency of each and every asynchronous code to their previous asynchronous code. that's why we have to make asynchronous code inside another asynchronous then inside another asynchronous code and then inside another asynchronous code and so on. and it become very complex line code.

4. Promise:

It is object to handle asynchronous code in JavaScript it is used to represent resultant statement of asynchronous code. resultant statement means resolve or failed for any asynchronous code.

in JavaScript promise has three phases : pending : in this state promise is waiting for asynchronous operation to complete resolved : this state indicate async code complete their operation rejected : this state indicate async code failed to execute. actually promise happens in our real life also we do promise that is our pending state and second is we fulfil our promise and the third one is we reject our promise.

let promise1= new Promise(function(resolve,reject){ setTimeout(() => { console.log('second'); resolve(); }, 2000); }); promise1.then(()=>{ console.log('promised resolve'); })

basically our async code is run with no boundation or no supervision but when we keep async code inside promise then they have to give resultant that our promise will execute or rejected it means promise is used to find the resultant of a async code resultant means will it complete or not.

there is two predefined function in promise resolve() and reject() if our promise has completed then we call resolve function and if our promise has not completed then we call reject function and outside promise be fetch this statement by . then() and .catch()

when our promise has completed then we call resolve() function and one more thing we can also pass data as a argument to resolve function and we fetch that data by .then() inside .then() function there is callback function which is use to perform operation on resolve data what will happen if our promise has not completed then we call reject() function and we fetch data of reject function by . catch() there is a callback function which will operate on the data of reject function

const promise3= new Promise((resolve,reject)=>{ setTimeout(() => { try{ console.log('promise 3 done'); resolve('promise3 resolve and consumed'); }catch{ reject('Error: somthing went error'); } }, 1000); }); promise3.then((user)=>{ console.log(user); }) .catch((error)=>{ console.log(error); });

what is the requirement of promise?

promise is the way to get eliminate by call back hell but how lets understand promise is nothing but it gives flag of resolve if our code has completed when we are making asynchronous code synchronously means async code is dependent to their previous code and we make call back hell what will happen if one of the async code has not completed then whole code will get freezed and get stuck application will not work when we keep each and every async code inside promises then as we know that promise has to give the flag of reject and resolve suppose one async code has not completed then it will response rejected and we fetch data in catch function.

It is changing of one promise to another by .then function actually we are making asynchronous code to synchronous code we know that in promise async code call resolve function when promise completed any fetch data of resolved in .then function and in .then function we define another promise and after complete it will again call resolve and we can fetch it by .then and so it is changing of promise to another promise by .then function we create first promise independently but after first promise we create dependent promise to first promise by .then function

const promise1 = new Promise((resolve,reject)=>{ setTimeout(()=>{ console.log('first promise'); resolve('promise1 resolve'); },2000); }); promise1.then((resolve)=>{ console.log(resolve); const promise2 = new Promise((resolve, reject) => { setTimeout(() => { console.log('second promise'); resolve('promise2 resolve'); }, 2000); }) return promise2; }).then((resolve)=>{ console.log(resolve); const promise3 = new Promise((resolve, reject) => { setTimeout(() => { console.log('third promise '); resolve('promise3 resolve'); }, 2000); }) return promise3; }).then((resolve)=>{ console.log(resolve); })

here in this code every promise is dependent to their previous promise except first so this is the elimination of call back Hell. our asynchronous code is running synchronously means it is dependent to each other and execute one by one parallelly

5. Async & Await:

async and await is advanced version of promise .then .catch actually it is very useful for handle callback hell as well as promise chaining earlier we wear watching that call back Hill can be resolved by promise chaining but what should happen if promising is also more than 100 promise then on that time it become hard to understand for developer so new concept that is async await come in market async is a function keyword it means how much promise we want to create we have to keep all promise in async function and await is used for waiting the response of promise it means this will wait for completion of promise async function content()

{ const p1= await new Promise((resolve,reject)=>{ setTimeout(()=>{ console.log('hello ji kaise ho'); reject('error: somthing went wrong'); },5000); }).catch((error)=>{ console.log(error); }); console.log(p1); console.log('finished'); } content();

here await is a very important keyword actually it is working like a . then function because when our promise completed then it send signal of resolve and if promise has not completed then also data will fetch by a await received signal of reject

Now we will see Elimination of promise chaining by async await

async function kamkaro()

{ const p1 = new Promise((resolve,reject)=>{ setTimeout(() => { console.log('first promise') resolve('p1 completed'); }, 2000); }) const promise1 = await p1; console.log(promise1); const p2 = new Promise((resolve,reject)=>{ setTimeout(() => { console.log('second promise') resolve('p2 completed'); }, 4000); }) const promise2 = await p2; console.log(promise2); const p3 = new Promise((resolve,reject)=>{ setTimeout(() => { console.log('third promise') resolve('p3 completed'); }, 4000); }) const promise3 = await p3; console.log(promise3); } kamkaro();

It defines an async function named "kamkaro" which sequentially awaits the completion of three promises. Each promise simulates an asynchronous task using setTimeout. The "await" keyword suspends the function execution until the promise is resolved, making the code appear synchronous. This approach enhances code readability and simplifies error handling compared to traditional promise chaining.

The function prints the completion messages of each promise, showcasing the orderly execution facilitated by async/await. Overall, async/await offers a more intuitive and concise way to handle asynchronous operations in JavaScript

7. Conclusion:

To wrap it up, this research has covered the basics of callback functions and the issue of callback hell in JavaScript. It introduced modern techniques like async/await to handle asynchronous tasks more efficiently. By using promises instead of callbacks, we can avoid the complexity of callback hell. Furthermore, we explored how to run asynchronous code synchronously, making it easier to manage. Understanding these concepts helps developers write cleaner and more understandable code, ultimately improving the quality of JavaScript projects..

Through the exploration of promises as an alternative to callbacks, developers can streamline their code and avoid the pitfalls of callback hell. Moreover, the discussion on executing asynchronous code synchronously empowers developers to better manage their codebase. Embracing these concepts not only improves code readability but also enhances the overall efficiency and maintainability of JavaScript projects, paving the way for smoother development experiences.

8. References:

https://blog.risingstack.com/node-js-async-best-practices-avoiding-callback-hell-node-js-at-scale/

https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous.

https://www.freecodecamp.org/news/what-is-a-callback-function-in-javascript/

https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous

https://www.digitalocean.com/community/tutorials/understanding-the-event-loop-callbacks-promises-and-async-await-in-javascript

https://javascript.info/async-await