**Node.js: Promise In Depth -** [Vikas Mishra](https://vikasmishra.medium.com/?source=post_page-----9007ce4a6ca4--------------------------------) Feb 19, 2022

All of us know about promises. They are improvements of lower-level building blocks of Node.js and are widely used by programmers. As a programmer, we should think about the readability of our code. This is the most important feature of promises. They made the code more readable.

In this article, we will learn more features of the Promise library and their advantages. Let’s start with the callbacks-

**Callbacks:**

We have seen many definitions of callbacks like “A callback is a function which is passed to another function “. In Node.js a callback definition is based on its asynchronous nature.

*“The most basic mechanism to notify the completion of asynchronous function is called callback”.*

There are certain disadvantages of callback like-

* Callback hell
* Pyramid Problem
* Closure and Parameters Renaming
* Error Handling

Let’s have a look at the callback hell problem first. At times we want to execute our code after the completion of another task. In programming, this is called the sequential execution (Asynchronous) of tasks.

At this stage, I assume we are familiar with the basic structure of the callback function.

Let’s check the below example -

async1((err,data) =>{  
 async2((err,data)=> {  
 async3((err,data)=> {   
 *//...* })   
 })   
})

Here we are passing the execution result of one function to another function like sequential execution of tasks. This leads our code into an unreadable and unmanageable blob is known as **callback hell.**You can see how code written in this way assumes the shape of a pyramid due to deep nesting, and that’s why it is also colloquially known as the **pyramid of doom**.

Another very important part is to check if we get an error in any of the results, it needs to be passed further in the application. A serial execution flow seems needlessly complicated and error-prone. If we forget to forward an error, then it just gets lost, and if we forget to catch any exception thrown by some synchronous code, then the program crashes. This is called a major **Error Handling** problem with callbacks.

**Promises:**

As you can see the most basic problem here is the readability of code. Now to solve this problem the JavaScript developers come up with a library which they called**Promise A+**.

*Promises are part of the ECMAScript 2015 standard (or ES6, which is why they are also called ES6 promises) and have been natively available in Node.js since version 4. But the history of promises goes back a few years earlier, when there were dozens of implementations around, initially with different features and behavior. Eventually, the majority of those implementations settled on a standard called****Promises/A+****.*

In simple words, we can say “ The first step toward a better asynchronous code experience is the **promise**, an object that “carries” the status and the eventual result of an asynchronous operation ”.

We will go into more details about Promise from here-

To have an idea of how promises can transform our code, let’s consider the following callback-based code:

asyncOperation(arg, (err, result) => {  
 if(err)   
 {   
 *// handle the error*   
 }   
 *// do stuff with the result*   
})

Promises allow us to transform this typical continuation-passing style code into a better structured and more elegant code, such as the following:

asyncOperationPromise(arg)   
.then(result => {   
 *// do stuff with result*   
 },  
 err => {   
 *// handle the error*   
 })

In the code above, asyncOperationPromise() is returning a Promise, which we can then use to receive the fulfillment value or the rejection reason of the eventual result of the function. The most impotent part of the promise is, we can pass the result of one operation to another one like -

asyncOperationPromise(arg)   
 .then(result1 => {  
 *// returns another promise* return asyncOperationPromise(arg2)   
 })  
 .then(result2 => {  
 *// returns a value*   
 return 'done'   
 })  
 .then(undefined, err => {   
 *// any error in the chain is caught here*   
 })

We can see the above code is more readable as compared to the nested Callback. We are doing the same sequential execution of tasks here also. This is the simplest form of writing a Promise in Node.js, but to write it in a better way JavaScript provides a native solution which they called **Promise API.**

**The promise API:**

new Promise((resolve, reject) => {  
 setTimeout(() => {  
 resolve(new Date())  
 }, milliseconds)  
 })

This is just an overview to give you an idea of what we can do with promises.

The promise constructor (new Promise((resolve, reject) => {})) creates a new promise instance that fulfills or rejects based on the behavior of the function provided as an argument. The function provided to the constructor will receive two arguments:

* **resolve(obj)**: This is a function that, when invoked, will fulfill the promise with the provided fulfillment value, which will be obj if obj is a value. It will be the fulfillment value of obj if obj is a promise or a thenable.
* **reject(err)**: This rejects the promise with the reason err. It is a convention for err to be an instance of Error.

**Creating a promise:**

Let’s now see how we can create a Promise using its constructor. Creating a Promise from scratch is a low-level operation and it's usually required when we need to convert an API that uses another asynchronous style (such as a callback-based style). Most of the time we—as developers—are consumers of promises produced by other libraries and most of the promises we create will come from the then() method. Nonetheless, in some advanced scenarios, we need to manually create a Promise using its constructor.

To demonstrate how to use the Promise constructor, let's create a function that returns a Promise that fulfills with the current date after a specified number of milliseconds. Let's take a look at it:

function delay (milliseconds) {  
 return new Promise((resolve, reject) => {  
 setTimeout(() => {  
 resolve(new Date())  
 }, milliseconds)  
 })  
}

As you probably already guessed, we used setTimeout to invoke the resolve function of the Promise constructor. We can notice how the entire body of the function is wrapped by the Promise constructor; this is a frequent code pattern you will see when creating a Promise from scratch.

The delay() function we just created can then be used with some code like the following:

console.log(`Delaying...${new Date().getSeconds()}s`)  
delay(1000)  
 .then(newDate => {  
 console.log(`Done ${newDate.getSeconds()}s`)  
 })

If any error is thrown by code or system we need to catch them but it is more simple than we did it in the callback pattern.

Now comes the best part. If an exception is thrown (using the throw statement) the promise returned by the then() method will automatically reject, with the exception that was thrown provided as the rejection reason. This is a tremendous advantage over callbacks**Error Handling** we saw earlier, as it means that with promises, exceptions will propagate automatically across the chain.

**Promises with Async/await:**

The promises are the best way to solve problems like callback hell and pyramid of doom but still, they are the suboptimal solution when it comes to writing sequential asynchronous code. We need to invoke then() and create a new function for each task in the chain .This is still too much for a control flow that is definitely the most commonly used in everyday programming. JavaScript needed a proper way to deal with the ubiquitous asynchronous sequential execution flow, and the answer arrived with the introduction in the ECMAScript standard of **async functions** and the await **expression** (**async/await** for short)

The async/await dichotomy allows us to write functions that appear to block at each asynchronous operation, waiting for the results before continuing with the following statement. As we will see, any asynchronous code using async/await has a readability comparable to traditional synchronous code.

Today, async/await is the recommended construct for dealing with asynchronous code in both Node.js and JavaScript. However, async/await does not replace all that we have learned so far about asynchronous control flow patterns; on the contrary, as we will see, async/await piggybacks heavily onto promise

Now lets take a example of the async/await-

function delay (milliseconds) {  
 return new Promise((resolve, reject) => {  
 setTimeout(() => {  
 resolve(new Date())  
 }, milliseconds)  
 })  
} async function playingWithDelays () {   
 console.log('Delaying...', new Date())   
 const dateAfterOneSecond = await delay(1000)  
 console.log(dateAfterOneSecond)   
 const dateAfterThreeSeconds = await delay(3000)  
 console.log(dateAfterThreeSeconds) return 'done'   
}

As we can see from the previous function, async/await seems to work like magic. The code doesn’t even look like it contains any asynchronous operation. However, don’t be mistaken; this function does not run synchronously (they are called async functions for a reason!). At each await expression, the execution of the function is put on hold, its state saved, and the control returned to the event loop. Once the Promise that has been *awaited* resolves, the control is given back to the async function, returning the fulfillment value of the Promise.

**Error handling with async/await:**

Async/await doesn’t just improve the readability of asynchronous code under standard conditions, but it also helps when handling errors. In fact, one of the biggest gains of async/await is the ability to normalize the behavior of the try/catch block, to make it work seamlessly with both synchronous *throws* and asynchronous Promise rejections. Let's demonstrate that with an example.

**A unified try…catch experience**

Let’s define a function that returns a Promise that rejects with an error after a given number of milliseconds. This is very similar to the delay() function that we already know very well:

function delayError (milliseconds) {  
 return new Promise((resolve, reject) => {  
 setTimeout(() => {  
 reject(new Error(`Error after ${milliseconds}ms`))  
 }, milliseconds)  
 })  
}

Next, let’s implement an async function that can throw an error synchronously or await a Promise that will reject. This function demonstrates how both the synchronous throw and the Promise rejection are caught by the same catch block:

async function playingWithErrors (throwSyncError) {  
 try {  
 if (throwSyncError) {  
 throw new Error('This is a synchronous error')  
 }  
 await delayError(1000)  
 } catch (err) {  
 console.error(`We have an error: ${err.message}`)  
 } finally {  
 console.log('Done')  
 }  
}

Now, error handling is just as it should be: simple, readable, and most importantly, supporting both synchronous and asynchronous errors.

Above we have covered all the important scenarios related to the Promise. I hope you will find these article useful. Give it some claps to make others find it too!