



Understanding the Event Loop, Callbacks, Promises, and Async/Await in JavaScript

This article was originally written for [DigitalOcean](#).

Introduction

In the early days of the internet, websites often consisted of static data in an [HTML page](#). But now that web applications have become more interactive and dynamic, it has become increasingly necessary to do intensive operations like make external network requests to retrieve [API](#) data. To handle these operations in JavaScript, a developer must use *asynchronous programming* techniques.

Since JavaScript is a *single-threaded* programming language with a *synchronous* execution model that processes one operation after another, it can only process one statement at a time. However, an action like requesting data from an API can take an indeterminate amount of time, depending on the size of data being requested, the speed of the network connection, and other factors. If API calls were performed in a synchronous manner, the browser would not be able to handle any user input, like scrolling or clicking a button, until that operation completes. This is known as *blocking*.

In order to prevent blocking behavior, the browser environment has many Web APIs that JavaScript can access that are *asynchronous*, meaning they can run in parallel with other operations instead of sequentially. This is useful because it allows the user to continue using the browser normally while the asynchronous operations are being processed.

As a JavaScript developer, you need to know how to work with asynchronous Web APIs and handle the response or error of those operations. In this article, you will learn about the event loop, the original way of dealing with asynchronous behavior through callbacks, the updated [ECMAScript 2015](#) addition of promises, and the modern practice of using `async / await`.

Note: This article is focused on client-side JavaScript in the browser environment. The same concepts are generally true in the [Node.js](#) environment, however Node.js uses its own [C++ APIs](#) as opposed to the browser's [Web APIs](#). For more information on asynchronous programming in Node.js, check out [How To Write Asynchronous Code in Node.js](#).

[About me](#) [Writing](#) [Projects](#)

- [Callback Functions](#)
 - [Nested Callbacks and the Pyramid of Doom](#)
- [Promises](#)
- [Async Functions with `async` / `await`](#)

The Event Loop

This section will explain how JavaScript handles asynchronous code with the event loop. It will first run through a demonstration of the event loop at work, and will then explain the two elements of the event loop: the stack and the queue.

JavaScript code that does not use any asynchronous Web APIs will execute in a synchronous manner—one at a time, sequentially. This is demonstrated by this example code that calls three functions that each print a number to the [console](#):

```
// Define three example functions
function first() {
  console.log(1)
}

function second() {
  console.log(2)
}

function third() {
  console.log(3)
}
```

In this code, you define three functions that print numbers with `console.log()`.

Next, write calls to the functions:

```
// Execute the functions
first()
second()
third()
```

The output will be based on the order the functions were called: `first()`, `second()`, then `third()`.

[About me](#) [Writing](#) [Projects](#)

```
1  
2  
3
```

When an asynchronous Web API is used, the rules become more complicated. A built-in API that you can test this with is `setTimeout`, which sets a timer and performs an action after a specified amount of time. `setTimeout` needs to be asynchronous, otherwise the entire browser would remain frozen during the waiting, which would result in a poor user experience.

Add `setTimeout` to the `second` function to simulate an asynchronous request:

```
// Define three example functions, but one of them contains asynchronous code  
function first() {  
  console.log(1)  
}  
  
function second() {  
  setTimeout(() => {  
    console.log(2)  
  }, 0)  
}  
  
function third() {  
  console.log(3)  
}
```

`setTimeout` takes two arguments: the function it will run asynchronously, and the amount of time it will wait before calling that function. In this code you wrapped `console.log` in an anonymous function and passed it to `setTimeout`, then set the function to run after `0` milliseconds.

Now call the functions, as you did before:

```
// Execute the functions  
first()  
second()  
third()
```

You might expect with a `setTimeout` set to `0` that running these three functions would still result in the numbers being printed in sequential order. But because it is asynchronous, the function with the timeout will be printed last:

[About me](#) [Writing](#) [Projects](#)

```
1
3
2
```

Whether you set the timeout to zero seconds or five minutes will make no difference—the `console.log` called by asynchronous code will execute after the synchronous top-level functions. This happens because the JavaScript host environment, in this case the browser, uses a concept called the *event loop* to handle concurrency, or parallel events. Since JavaScript can only execute one statement at a time, it needs the event loop to be informed of when to execute which specific statement. The event loop handles this with the concepts of a *stack* and a *queue*.

Stack

The *stack*, or call stack, holds the state of what function is currently running. If you're unfamiliar with the concept of a stack, you can imagine it as an [array](#) with "Last in, first out" (LIFO) properties, meaning you can only add or remove items from the end of the stack. JavaScript will run the current *frame* (or function call in a specific environment) in the stack, then remove it and move on to the next one.

For the example only containing synchronous code, the browser handles the execution in the following order:

- Add `first()` to the stack, run `first()` which logs 1 to the console, remove `first()` from the stack.
- Add `second()` to the stack, run `second()` which logs 2 to the console, remove `second()` from the stack.
- Add `third()` to the stack, run `third()` which logs 3 to the console, remove `third()` from the stack.

The second example with `setTimeout` looks like this:

- Add `first()` to the stack, run `first()` which logs 1 to the console, remove `first()` from the stack.
- Add `second()` to the stack, run `second()`.
 - Add `setTimeout()` to the stack, run the `setTimeout()` Web API which starts a timer and adds the anonymous function to the *queue*, remove `setTimeout()` from the stack.
- Remove `second()` from the stack.
- Add `third()` to the stack, run `third()` which logs 3 to the console, remove `third()` from the stack.

[About me](#) [Writing](#) [Projects](#)

the stack.

Using `setTimeout`, an asynchronous Web API, introduces the concept of the *queue*, which this tutorial will cover next.

Queue

The *queue*, also referred to as message queue or task queue, is a waiting area for functions. Whenever the call stack is empty, the event loop will check the queue for any waiting messages, starting from the oldest message. Once it finds one, it will add it to the stack, which will execute the function in the message.

In the `setTimeout` example, the anonymous function runs immediately after the rest of the top-level execution, since the timer was set to `0` seconds. It's important to remember that the timer does not mean that the code will execute in exactly `0` seconds or whatever the specified time is, but that it will add the anonymous function to the queue in that amount of time. This queue system exists because if the timer were to add the anonymous function directly to the stack when the timer finishes, it would interrupt whatever function is currently running, which could have unintended and unpredictable effects.

Note: There is also another queue called the *job queue* or *microtask queue* that handles promises. Microtasks like promises are handled at a higher priority than macrotasks like `setTimeout`.

Now you know how the event loop uses the stack and queue to handle the execution order of code. The next task is to figure out how to control the order of execution in your code. To do this, you will first learn about the original way to ensure asynchronous code is handled correctly by the event loop: callback functions.

Callback Functions

In the `setTimeout` example, the function with the timeout ran after everything in the main top-level execution context. But if you wanted to ensure one of the functions, like the `third` function, ran after the timeout, then you would have to use asynchronous coding methods. The timeout here can represent an asynchronous API call that contains data. You want to work with the data from the API call, but you have to make sure the data is returned first.

[About me](#) [Writing](#) [Projects](#)

The function that takes another function as an argument is called a *higher-order function*. According to this definition, any function can become a callback function if it is passed as an argument. Callbacks are not asynchronous by nature, but can be used for asynchronous purposes.

Here is a syntactic code example of a higher-order function and a callback:

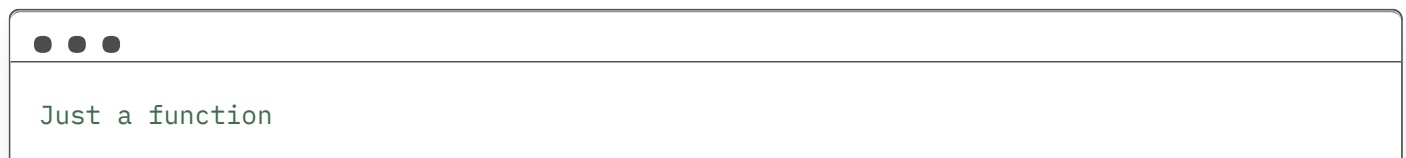
```
// A function
function fn() {
  console.log('Just a function')
}

// A function that takes another function as an argument
function higherOrderFunction(callback) {
  // When you call a function that is passed as an argument, it is referred to as a call
  callback()
}

// Passing a function
higherOrderFunction(fn)
```

In this code, you define a function `fn`, define a function `higherOrderFunction` that takes a function `callback` as an argument, and pass `fn` as a callback to `higherOrderFunction`.

Running this code will give the following:



Let's go back to the `first`, `second`, and `third` functions with `setTimeout`. This is what you have so far:

```
function first() {
  console.log(1)
}

function second() {
  setTimeout(() => {
    console.log(2)
  }, 0)
```

[About me](#) [Writing](#) [Projects](#)

```
console.log(3)
}
```

The task is to get the `third` function to always delay execution until after the asynchronous action in the `second` function has completed. This is where callbacks come in. Instead of executing `first`, `second`, and `third` at the top-level of execution, you will pass the `third` function as an argument to `second`. The `second` function will execute the callback after the asynchronous action has completed.

Here are the three functions with a callback applied:

```
// Define three functions
function first() {
  console.log(1)
}

function second(callback) {
  setTimeout(() => {
    console.log(2)

    // Execute the callback function
    callback()
  }, 0)
}

function third() {
  console.log(3)
}
```

Now, execute `first` and `second`, then pass `third` as an argument to `second`:

```
first()
second(third)
```

After running this code block, you will receive the following output:

```
1
2
3
```



execution of the function until the asynchronous Web API (`setTimeout`) completes.

The key takeaway here is that callback functions are not asynchronous— `setTimeout` is the asynchronous Web API responsible for handling asynchronous tasks. The callback just allows you to be informed of when an asynchronous task has completed and handles the success or failure of the task.

Now that you have learned how to use callbacks to handle asynchronous tasks, the next section explains the problems of nesting too many callbacks and creating a "pyramid of doom."

Nested Callbacks and the Pyramid of Doom

Callback functions are an effective way to ensure delayed execution of a function until another one completes and returns with data. However, due to the nested nature of callbacks, code can end up getting messy if you have a lot of consecutive asynchronous requests that rely on each other. This was a big frustration for JavaScript developers early on, and as a result code containing nested callbacks is often called the "pyramid of doom" or "callback hell."

Here is a demonstration of nested callbacks:

```
function pyramidOfDoom() {  
  setTimeout(() => {  
    console.log(1)  
    setTimeout(() => {  
      console.log(2)  
      setTimeout(() => {  
        console.log(3)  
      }, 500)  
    }, 2000)  
  }, 1000)  
}
```

In this code, each new `setTimeout` is nested inside a higher order function, creating a pyramid shape of deeper and deeper callbacks. Running this code would give the following:



[About me](#) [Writing](#) [Projects](#)

next request. Doing this with callbacks will make your code difficult to read and maintain.

Here is a runnable example of a more realistic "pyramid of doom" that you can play around with:

```
// Example asynchronous function
function asynchronousRequest(args, callback) {
  // Throw an error if no arguments are passed
  if (!args) {
    return callback(new Error('Whoa! Something went wrong.'))
  } else {
    return setTimeout(
      // Just adding in a random number so it seems like the contrived asynchronous func
      // returned different data
      () => callback(null, { body: args + ' ' + Math.floor(Math.random() * 10) }),
      500
    )
  }
}

// Nested asynchronous requests
function callbackHell() {
  asynchronousRequest('First', function first(error, response) {
    if (error) {
      console.log(error)
      return
    }
    console.log(response.body)
    asynchronousRequest('Second', function second(error, response) {
      if (error) {
        console.log(error)
        return
      }
      console.log(response.body)
      asynchronousRequest(null, function third(error, response) {
        if (error) {
          console.log(error)
          return
        }
        console.log(response.body)
      })
    })
  })
}

// Execute
callbackHell()
```

[About me](#) [Writing](#) [Projects](#)

in the code, you must make every function account for a possible response and a possible error, making the function `callbackHell` visually confusing.

Running this code will give you the following:

```
First 9
Second 3
Error: Whoa! Something went wrong.
    at asynchronousRequest (<anonymous>:4:21)
    at second (<anonymous>:29:7)
    at <anonymous>:9:13
```

This way of handling asynchronous code is difficult to follow. As a result, the concept of *promises* was introduced in ES6. This is the focus of the next section.

Promises

A *promise* represents the completion of an asynchronous function. It is an object that might return a value in the future. It accomplishes the same basic goal as a callback function, but with many additional features and a more readable syntax. As a JavaScript developer, you will likely spend more time consuming promises than creating them, as it is usually asynchronous Web APIs that return a promise for the developer to consume. This tutorial will show you how to do both.

Creating a Promise

You can initialize a promise with the `new Promise` syntax, and you must initialize it with a function. The function that gets passed to a promise has `resolve` and `reject` parameters. The `resolve` and `reject` functions handle the success and failure of an operation, respectively.

Write the following line to declare a promise:

```
// Initialize a promise
const promise = new Promise((resolve, reject) => {})
```

If you inspect the initialized promise in this state with your web browser's console, you will find it has a pending status and undefined value:

[About me](#) [Writing](#) [Projects](#)

```
__proto__: Promise
[[PromiseStatus]]: "pending"
[[PromiseValue]]: undefined
```

So far, nothing has been set up for the promise, so it's going to sit there in a `pending` state forever. The first thing you can do to test out a promise is fulfill the promise by resolving it with a value:

```
const promise = new Promise((resolve, reject) => {
  resolve('We did it!')
})
```

Now, upon inspecting the promise, you'll find that it has a status of `fulfilled`, and a `value` set to the value you passed to `resolve`:

```
__proto__: Promise
[[PromiseStatus]]: "fulfilled"
[[PromiseValue]]: "We did it!"
```

As stated in the beginning of this section, a promise is an object that may return a value. After being successfully fulfilled, the `value` goes from `undefined` to being populated with data.

A promise can have three possible states: `pending`, `fulfilled`, and `rejected`.

- **Pending** - Initial state before being resolved or rejected
- **Fulfilled** - Successful operation, promise has resolved
- **Rejected** - Failed operation, promise has rejected

After being fulfilled or rejected, a promise is settled.

Now that you have an idea of how promises are created, let's look at how a developer may consume these promises.

Consuming a Promise

The promise in the last section has fulfilled with a value, but you also want to be able to access the value. Promises have a method called `then` that will run after a promise reaches `resolve` in the code.

[About me](#) [Writing](#) [Projects](#)

This is how you would return and log the `value` of the example promise:

```
promise.then((response) => {  
  console.log(response)  
})
```

The promise you created had a `[[PromiseValue]]` of `We did it!`. This value is what will be passed into the anonymous function as `response`:



```
We did it!
```

So far, the example you created did not involve an asynchronous Web API—it only explained how to create, resolve, and consume a native JavaScript promise. Using `setTimeout`, you can test out an asynchronous request.

The following code simulates data returned from an asynchronous request as a promise:

```
const promise = new Promise((resolve, reject) => {  
  setTimeout(() => resolve('Resolving an asynchronous request!'), 2000)  
})  
  
// Log the result  
promise.then((response) => {  
  console.log(response)  
})
```

Using the `then` syntax ensures that the `response` will be logged only when the `setTimeout` operation is completed after `2000` milliseconds. All this is done without nesting callbacks.

Now after two seconds, it will resolve the promise value and it will get logged in `then`:




```
Resolving an asynchronous request!
```

Promises can also be chained to pass along data to more than one asynchronous operation. If a value is returned in `then`, another `then` can be added that will fulfill with the return value of the previous

[About me](#) [Writing](#) [Projects](#)

```
// Chain a promise
promise
  .then((firstResponse) => {
    // Return a new value for the next then
    return firstResponse + ' And chaining!'
  })
  .then((secondResponse) => {
    console.log(secondResponse)
  })
```

The fulfilled response in the second `then` will log the return value:



```
Resolving an asynchronous request! And chaining!
```

Since `then` can be chained, it allows the consumption of promises to appear more synchronous than callbacks, as they do not need to be nested. This will allow for more readable code that can be maintained and verified easier.

Error Handling

So far, you have only handled a promise with a successful `resolve`, which puts the promise in a `fulfilled` state. But frequently with an asynchronous request you also have to handle an error—if the API is down, or a malformed or unauthorized request is sent. A promise should be able to handle both cases. In this section, you will create a function to test out both the success and error case of creating and consuming a promise.

This `getUsers` function will pass a flag to a promise, and return the promise.

```
function getUsers(onSuccess) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      // Handle resolve and reject in the asynchronous API
    }, 1000)
  })
}
```



```
function getUsers(onSuccess) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      // Handle resolve and reject in the asynchronous API
      if (onSuccess) {
        resolve([
          { id: 1, name: 'Jerry' },
          { id: 2, name: 'Elaine' },
          { id: 3, name: 'George' },
        ])
      } else {
        reject('Failed to fetch data!')
      }
    }, 1000)
  })
}
```

For the successful result, you return [JavaScript objects](#) that represent sample user data.

In order to handle the error, you will use the `catch` instance method. This will give you a failure callback with the `error` as a parameter.

Run the `getUsers` command with `onSuccess` set to `false`, using the `then` method for the success case and the `catch` method for the error:

```
// Run the getUsers function with the false flag to trigger an error
getUsers(false)
  .then((response) => {
    console.log(response)
  })
  .catch((error) => {
    console.error(error)
  })
```

Since the error was triggered, the `then` will be skipped and the `catch` will handle the error:





```
// Run the getUsers function with the true flag to resolve successfully
getUsers(true)
  .then((response) => {
    console.log(response)
  })
  .catch((error) => {
    console.error(error)
  })
```

This will yield the user data:

```
(3) [{...}, {...}, {...}]
0: {id: 1, name: "Jerry"}
1: {id: 2, name: "Elaine"}
3: {id: 3, name: "George"}
```

For reference, here is a table with the handler methods on `Promise` objects:

Method	Description
<code>then()</code>	Handles a <code>resolve</code> . Returns a promise, and calls <code>onFulfilled</code> function asynchronously
<code>catch()</code>	Handles a <code>reject</code> . Returns a promise, and calls <code>onRejected</code> function asynchronously
<code>finally()</code>	Called when a promise is settled. Returns a promise, and calls <code>onFinally</code> function asynchronously

Promises can be confusing, both for new developers and experienced programmers that have never worked in an asynchronous environment before. However as mentioned, it is much more common to consume promises than create them. Usually, a browser's Web API or third party library will be providing the promise, and you only need to consume it.

In the final promise section, this tutorial will cite a common use case of a Web API that returns promises: [the Fetch API](#).

Using the Fetch API with Promises


One of the most useful and frequently used Web APIs that returns a promise is the Fetch API, which allows you to make an asynchronous resource request over a network. `fetch` is a two-part process,



```
// Fetch a user from the GitHub API
fetch('https://api.github.com/users/octocat')
  .then((response) => {
    return response.json()
  })
  .then((data) => {
    console.log(data)
  })
  .catch((error) => {
    console.error(error)
  })
```

The `fetch` request is sent to the `https://api.github.com/users/octocat` URL, which asynchronously waits for a response. The first `then` passes the response to an anonymous function that formats the response as [JSON data](#), then passes the JSON to a second `then` that logs the data to the console. The `catch` statement logs any error to the console.

Running this code will yield the following:



```
login: "octocat",
id: 583231,
avatar_url: "https://avatars3.githubusercontent.com/u/583231?v=4"
blog: "https://github.blog"
company: "@github"
followers: 3203
...
```

This is the data requested from `https://api.github.com/users/octocat`, rendered in JSON format.

This section of the tutorial showed that promises incorporate a lot of improvements for dealing with asynchronous code. But, while using `then` to handle asynchronous actions is easier to follow than the pyramid of callbacks, some developers still prefer a synchronous format of writing asynchronous code. To address this need, [ECMAScript 2016 \(ES7\)](#) introduced `async` functions and the `await` keyword to make working with promises easier.

Async Functions with `async/await`

[About me](#) [Writing](#) [Projects](#)

section, you will try out examples of this syntax.

You can create an `async` function by adding the `async` keyword before a function:


```
// Create an async function
async function getUser() {
  return {}
}
```

Although this function is not handling anything asynchronous yet, it behaves differently than a traditional function. If you execute the function, you'll find that it returns a promise with a `[[PromiseStatus]]` and `[[PromiseValue]]` instead of a return value.

Try this out by logging a call to the `getUser` function:

```
console.log(getUser())
```

This will give the following:



```
__proto__: Promise
[[PromiseStatus]]: "fulfilled"
[[PromiseValue]]: Object
```

This means you can handle an `async` function with `then` in the same way you could handle a promise.

Try this out with the following code:

```
getUser().then((response) => console.log(response))
```

This call to `getUser` passes the return value to an anonymous function that logs the value to the console.

You will receive the following when you run this program:

[About me](#) [Writing](#) [Projects](#)

25

An `async` function can handle a promise called within it using the `await` operator. `await` can be used within an `async` function and will wait until a promise settles before executing the designated code.

With this knowledge, you can rewrite the Fetch request from the last section using `async / await` as follows:

```
// Handle fetch with async/await
async function getUser() {
  const response = await fetch('https://api.github.com/users/octocat')
  const data = await response.json()

  console.log(data)
}

// Execute async function
getUser()
```

The `await` operators here ensure that the `data` is not logged before the request has populated it with data.

Now the final `data` can be handled inside the `getUser` function, without any need for using `then`. This is the output of logging `data`:

```
login: "octocat",
id: 583231,
avatar_url: "https://avatars3.githubusercontent.com/u/583231?v=4"
blog: "https://github.blog"
company: "@github"
followers: 3203
...
```

Note: In many environments, `async` is necessary to use `await`—however, some new versions of browsers and Node allow using top-level `await`, which allows you to bypass creating an `async` function to wrap the `await` in.

[About me](#) [Writing](#) [Projects](#)

[try / catch](#) pattern to handle the exception.

Add the following highlighted code:

```
// Handling success and errors with async/await
async function getUser() {
  try {
    // Handle success in try
    const response = await fetch('https://api.github.com/users/octocat')
    const data = await response.json()

    console.log(data)
  } catch (error) {
    // Handle error in catch
    console.error(error)
  }
}
```

The program will now skip to the `catch` block if it receives an error and log that error to the console.

Modern asynchronous JavaScript code is most often handled with `async / await` syntax, but it is important to have a working knowledge of how promises work, especially as promises are capable of additional features that cannot be handled with `async / await`, like combining promises with [Promise.all\(\)](#).

Note: `async / await` can be reproduced by using [generators combined with promises](#) to add more flexibility to your code. To learn more, check out our [Understanding Generators in JavaScript](#) tutorial.

Conclusion

Because Web APIs often provide data asynchronously, learning how to handle the result of asynchronous actions is an essential part of being a JavaScript developer. In this article, you learned how the host environment uses the event loop to handle the order of execution of code with the *stack* and *queue*. You also tried out examples of three ways to handle the success or failure of an asynchronous event, with callbacks, promises, and `async / await` syntax. Finally, you used the Fetch Web API to handle asynchronous actions.



[How To Code in JavaScript](#) series.

Comments

22 Comments - powered by [utteranc.es](#)

markomeje commented on Sep 10, 2020

Very explanatory. Thank you and keep it up.



6



1

dephraiim commented on Sep 10, 2020

Thank You.

kemalgencay commented on Sep 10, 2020

Wonderful article, thank you,

mshams999 commented on Sep 11, 2020

Nice

Lissettelbnz commented on Sep 11, 2020

Great!

VikR0001 commented on Sep 11, 2020

I'm reading your article and learning a lot! I have a quick question. A lot of my code has this:

```
return Promise.resolve()
  .then(() => {
    let objectFromDatabase = connectors.myTable.findOne({
      where: {id: objectId}
    });
    return objectFromDatabase;
  })
  .catch((err) => {
    console.log(err);
  });
```

[About me](#) [Writing](#) [Projects](#)

```
return new Promise.resolve()
```

...instead of:

```
return Promise.resolve()
```

philsav commented on Sep 12, 2020

Very good article, just need to know is which scenario to use promise and which to use async/await, Is promises used when multiple promises need to be checked using promise all, is there also a await all

astrit commented on Sep 12, 2020

What a detailed explanation, learned a-lot from it. thank you

KVNEZE commented on Sep 13, 2020

This is a really well written article, I connected dots that I didn't know I needed to connect.

Thank you.

andrebonfimdev commented on Sep 22, 2020

Great article. Congratulations!

moofoo commented on Sep 25, 2020

Concise and informative.

yonycalsin commented on Sep 26, 2020

Nice 👍

👍 1

pb03 commented on Sep 26, 2020

Nicely explained.

[About me](#) [Writing](#) [Projects](#)

Web API will add this function to the queue (not directly from the stack)

DrWongKC commented on Sep 27, 2020

Thank you so much for writing this. You're an incredibly teacher. :) I hope you keep writing about programming, even after you decide to retire. It's really helping me.

liwei766 commented on Sep 29, 2020

Good personal blog! Static site and commenting system works.

kusaljay commented on Oct 11, 2020

Thanks for this article.
Can you please write a one for Closures?

sddania commented on Oct 16, 2020

@VikR0001

My question is, do I need to be doing: `return new Promise.resolve()` instead of: `return Promise.resolve()` as [Promise doc](#) says, the `resolve` method is static and you don't need to instantiate a new Promise to call it. if you want to use Promise as object you can wrap functions that do not already support promises on Promise constructor:

```
const promise = new Promise( (res ,rej) => {  
  res("hello!")  
} )  
promise.then(val => console.log(val))
```

the console will show "hello!"

pedegago commented on Oct 24, 2020

Great article. I was reading it for some days in order to understand everything!

hunzaboy commented on Oct 28, 2020

[About me](#) [Writing](#) [Projects](#)

immaithil commented on Oct 7, 2021

It was a great article finally I am able to clear confusion about .then and .catch.

alex-j-garcia commented on Nov 5, 2021

Thanks for the great article. Some things didn't make sense right away, but I kept rereading and going through your examples until it clicked!

aakinlalu commented on Nov 13, 2021

Thank you

Write

Preview

Sign in to comment



Styling with Markdown is supported

Sign in with GitHub



ABOUT ME



[About me](#) [Writing](#) [Projects](#)

I'm a software developer who writes articles and tutorials about things that interest me. This site is and has always been free of ads, trackers, social media, affiliates, and sponsored posts.

I hope you enjoy the post and have a nice day.

DETAILS

- Published September 10, 2020

CATEGORY

- JavaScript

TAGS

[javascript](#) [fundamentals](#) [asynchronous](#) [es6](#)[Newsletter](#) [Ko-Fi](#) [Patreon](#) [RSS](#)[Gatsby](#)  [GitHub](#)  [Netlify](#) 