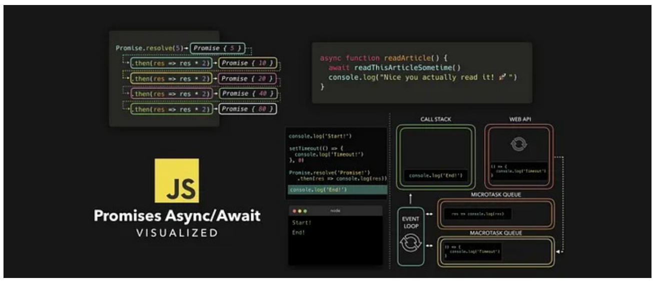
JavaScript Visualized: Promises & Async/Await

Lydia Hallie · Follow
14 min read · Apr 15, 2020

> 🔥🪄 If you don't want to deal with the paywall, here is the original article!


Promise.resolve(5) → Promise { 5 }
.then(res => res * 2) → Promise { 10 }
.then(res => res * 2) → Promise { 20 }
.then(res => res * 2) → Promise { 40 }
.then(res => res * 2) → Promise { 80 }
async function readArticle() {
await readThisArticleSometime()
console.log("Nice you actually read it! ")
}
JS
Promises Async/Await
VISUALIZED
CALL STACK
WEB API
MICROTASK QUEUE
MACROTASK QUEUE
EVENT LOOP
Start!
End!

Ever had to deal with JS code that just… didn't run the way you expected it to? Maybe it seemed like functions got executed at random, unpredictable times, or the execution got delayed. There's a chance you were dealing with a cool new feature that ES6 introduced: **Promises**!

My curiosity from many years ago has paid off and my sleepless nights have once again given me the time to make some animations. Time to talk about Promises: **why** would you use them, **how** do they work "under the hood", and

The very first thing we need to do, is *get* the image that we want to edit. A `getImage` function can take care of this! Only once that image has been loaded successfully, we can pass that value to a `resizeImage` function. When the image has been resized successfully, we want to apply a filter to the image in the `applyFilter` function. After the image has been compressed and we've added a filter, we want to save the image and let the user know that everything worked correctly! 🥳

In the end, we'll end up with something like this:

```
getImage('./image.png', (image, err) => {
  if (err) throw new Error(err)
  compressImage(image, (compressedImage, err) => {
    if (err) throw new Error(err)
    applyFilter(compressedImage, (filteredImage, err) => {
      if (err) throw new Error(err)
      saveImage(compressedImage, (res, err) => {
        if (err) throw new Error(err)
        console.log("Successfully saved image!")
      })
    })
  })
})
```

Hmm... Notice anything here? Although it's... *fine*, it's not great. We end up with many nested callback functions that are dependent on the previous callback function. This is often referred to as a *callback hell*, as we end up with tons of nested callback functions that make the code quite difficult to read!

Luckily, we now got something called **promises** to help us out! Let's take a look at what promises are, and how they can help us in situations like these! 😃

. . .

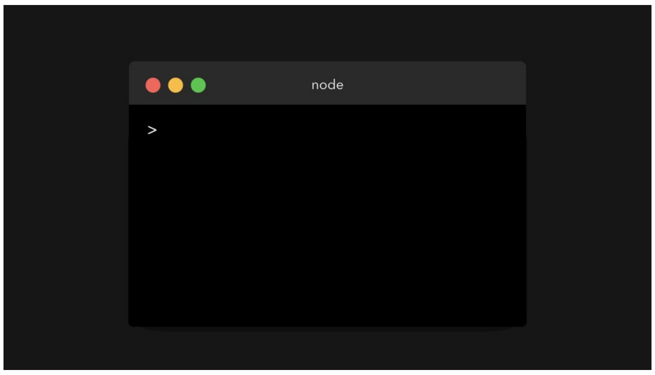
Promise Syntax

ES6 introduced **Promises.** In many tutorials, you'll read something like:

> "A promise is a placeholder for a value that can either resolve or reject at some time in the future"

Yeah... That explanation never made things clearer for me. In fact it only made me feel like a Promise was a weird, vague, unpredictable piece of magic. So let's look at what promises *really* are.

We can create a promise, using a `Promise` constructor that receives a callback. Okay cool, let's try it out!



```
>
```

Wait woah, what just got returned? 🤯

A `Promise` is an object that contains a **status**, (`[[PromiseStatus]]`) and a **value** (`[[PromiseValue]]`). In the above example, you can see that the value of `[[PromiseStatus]]` is `"pending"`, and the value of the promise is `undefined`.

Don't worry — you'll never have to interact with this object, you can't even access the `[[PromiseStatus]]` and `[[PromiseValue]]` properties! However, the values of these properties are important when working with promises.

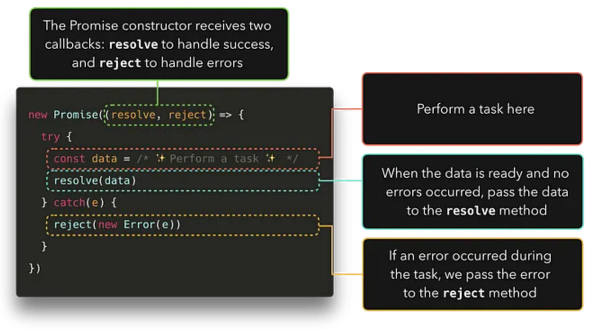
. . .

The value of the `PromiseStatus`, the **state**, can be one of three values:

- ✅ `fulfilled`: The promise has been `resolved`. Everything went fine, no errors occurred within the promise 🥳
- ❌ `rejected` : The promise has been `rejected`. Argh, something went wrong..
- ⏳ `pending`: The promise has neither resolved nor rejected (yet), the promise is still `pending`.

Alright this all sounds great, but *when* is a promise status `"pending"`, `"fulfilled"` or `"rejected"`? And why does that status even matter?

In the above example, we just passed the simple callback function `() => {}` to the `Promise` constructor. However, this callback function actually receives two arguments. The value of the first argument, often called `resolve` or `res`, is the method to be called when the Promise should **resolve**. The value of the second argument, often called `reject` or `rej`, is the value method to be called when the Promise should **reject**, something went wrong.



```
new Promise((resolve, reject) => {
  try {
    const data = /* ✨ Perform a task ✨ */
    resolve(data)
  } catch(e) {
    reject(new Error(e))
  }
})
```

The Promise constructor receives two callbacks: **resolve** to handle success, and **reject** to handle errors
Perform a task here
When the data is ready and no errors occurred, pass the data to the **resolve** method
If an error occurred during the task, we pass the error to the **reject** method

Let's try and see that gets logged when we invoke either the `resolve` or `reject` method! In my example, I called the `resolve` method `res`, and the `reject` method `rej`.

Awesome! We finally know how to get rid of the `"pending"` status and the `undefined` value! The **status** of a promise is `"fulfilled"` if we invoked the `resolve` method, and the status of the promise is `"rejected"` if we invoked the `rejected` method.

The **value** of a promise, the value of `[[PromiseValue]]`, is the value that we pass to the either the `resolved` or `rejected` method as their argument.

> Fun fact, I let Jake Archibald proofread this article and he actually pointed out there's a bug in Chrome that currently shows the status as `"resolved"` instead of `"fulfilled"`. Thanks to Mathias Bynens it's now fixed in Canary! 🥳🕺

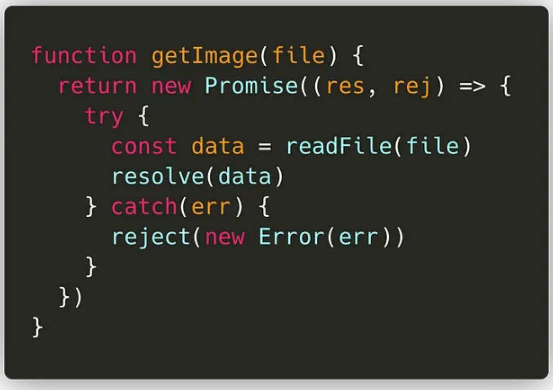
. . .

Okay so, now we know a little better how to control that vague `Promise` object. But what is it used for?

In the introductory section, I showed an example in which we get an image, compress it, apply a filer, and save it! Eventually, this ended up being a nested callback mess.

Luckily, Promises can help us fix this! First, let's rewrite the entire code block, so that each function returns a `Promise` instead.

If the image is loaded and everything went fine, let's **resolve** the promise with the loaded image! Else, if there was an error somewhere while loading the file, let's **reject** the promise with the error that occurred.



```
function getImage(file) {
  return new Promise((res, rej) => {
    try {
      const data = readFile(file)
      resolve(data)
    } catch(err) {
      reject(new Error(err))
    }
  })
}
```

Let's see what happens when we run this in the terminal!

Cool! A promise got returned with the value of the parsed data, just like we expected.

But... what now? We don't care about that entire promise object, we only care about the value of the data! Luckily, there are built-in methods to get a promise's value. To a promise, we can attach 3 methods:

- `.then()`: Gets called after a promise *resolved*.
- `.catch()`: Gets called after a promise *rejected*.
- `.finally()`: *Always* gets called, whether the promise resolved or rejected.

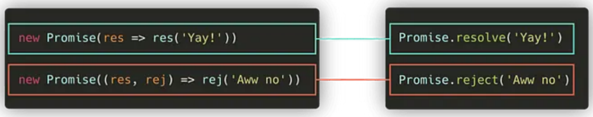
The `.then` method receives the value passed to the `resolve` method.

The `.catch` method receives the value passed to the `rejected` method

Finally, we have the value that got resolved by the promise without having that entire promise object! We can now do whatever we want with this value.

. . .

FYI, when you know that a promise will always resolve or always reject, you can write `Promise.resolve` or `Promise.reject` , with the value you want to reject or resolve the promise with!



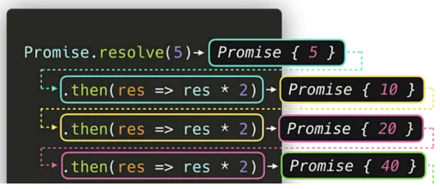
```
new Promise(res => res('Yay!'))                Promise.resolve('Yay!')
new Promise((res, rej) => rej('Aww no'))       Promise.reject('Aww no')
```

You'll often see this syntax in the following examples 😄

. . .

In the `getImage` example, we ended up having to nest multiple callbacks in order to run them. Luckily, the `.then` handlers can help us with that! 🥳

The result of the `.then` itself is a promise value. This means that we can chain as many `.then`s as we want: the result of the previous `then` callback will be passed as an argument to the next `then` callback!



```
Promise.resolve(5)→ Promise { 5 }
  .then(res => res * 2)→ Promise { 10 }
  .then(res => res * 2)→ Promise { 20 }
  .then(res => res * 2)→ Promise { 40 }
```

```
.then(res => res * 2) → Promise { 80 }
```

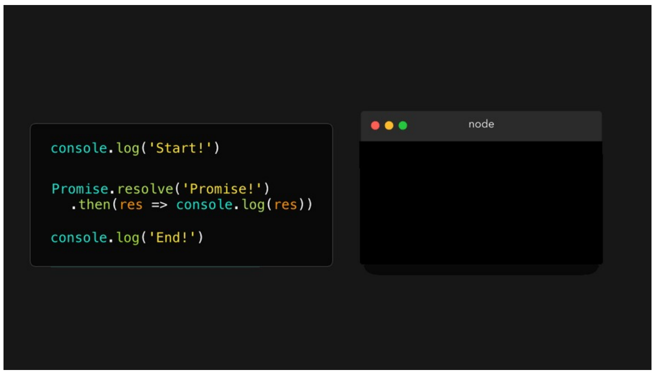
In the case of the `getImage` example, we can chain multiple `then` callbacks in order to pass the processed image onto the next function! Instead of ending up with many nested callbacks, we get a clean `then` chain.

Perfect! This syntax already looks way better than the nested callbacks.

. . .

Microtasks and (Macro)tasks

Okay so we know a little better how to create a promise and how to extract values out of a promise. Let's add some more code to the script, and run it again:



```
console.log('Start!')

Promise.resolve('Promise!')
  .then(res => console.log(res))

console.log('End!')
```

Wait what?! 😲

First, `Start!` got logged. Okay we could've seen that one coming: `console.log('Start!')` is on the very first line! However, the second value that got logged was `End!`, and *not* the value of the resolved promise! Only after `End!` was logged, the value of the promise got logged. What's going on here?

We've finally seen the true power of promises! 🚀 Although JavaScript is single-threaded, we can add asynchronous behavior using a `Promise`!

. . .

But wait, haven't we seen that before? 🤔 In the JavaScript event loop, can't we also use methods native to the browser such as `setTimeout` to create some sort of asynchronous behavior?

Yes! Hoewver, within the Event Loop, there are actually two types of queues: the **(macro)task queue** (or just called the **task queue**), and the **microtask queue**. The (macro)task queue is for **(macro)tasks** and the microtask queue is for **microtasks**.

So what's a *(macro)task* and what's a *microtask*? Although there are a few more than I'll cover here, the most common are shown in the table below!

(Macro)task: `setTimeout` | `setInterval`
Microtask: `process.nextTick` | `Promise callback` | `queueMicrotask`

Ahh, we see `Promise` in the microtask list! 😃 When a `Promise` resolves and calls its `then()`, `catch()` or `finally()`, method, the callback within the method gets added to the **microtask queue**! This means that the callback within the `then()`, `catch()` or `finally()` method isn't executed immediately, essentially adding some async behavior to our JavaScript code!

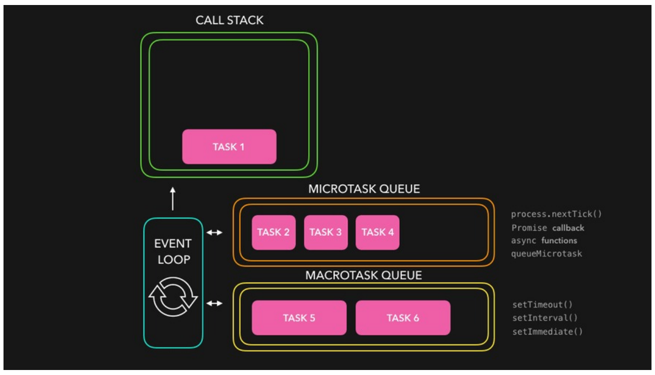
So when *is* a `then()`, `catch()` or `finally()` callback executed? The event loop gives a different priority to the tasks:

1. All functions in that are currently in the **call stack** get executed. When they returned a value, they get popped off the stack.
2. When the call stack is empty, *all* queued up **microtasks** are popped onto the callstack one by one, and get executed! (Microtasks themselves can also return new microtasks, effectively creating an infinite microtask loop 😬)
3. If both the call stack and microtask queue are empty, the event loop checks if there are tasks left on the (macro)task queue. The tasks get popped onto the callstack, executed, and popped off!

. . .

Let's take a look at a quick example, simply using:

- `Task1`: a function that's added to the call stack immediately, for example by invoking it instantly in our code.
- `Task2`, `Task3`, `Task4`: microtasks, for example a promise `then` callback, or a task added with `queueMicrotask`.
- `Task5`, `Task6`: a (macro)task, for example a `setTimeout` or `setImmediate` callback



First, `Task1` returned a value and got popped off the call stack. Then, the engine checked for tasks queued in the microtask queue. Once all the tasks were put on the call stack and eventually popped off, the engine checked for tasks on the (macro)task queue, which got popped onto the call stack, and popped off when they returned a value.

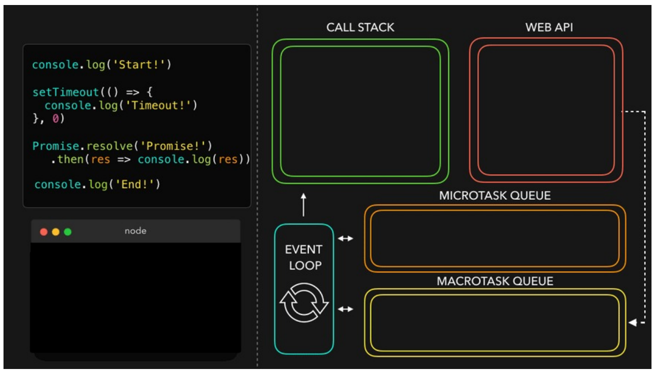
Okay okay enough pink boxes. Let's use it with some real code!

In this code, we have the macro task `setTimeout`, and the microtask promise `then()` callback. Once the engine reaches the line of the `setTimeout` function. Let's run this code step-by-step, and see what gets logged!

. . .

> Quick FYI — in the following examples I'm showing methods like `console.log`, `setTimeout` and `Promise.resolve` being added to the call stack. They're internal methods and actually don't appear in stack traces - so don't worry if you're using the debugger and you don't see them anywhere! It just makes explaining this concept easier without adding a bunch of boilerplate code 🙂

On the first line, the engine encounters the `console.log()` method. It gets added to the call stack, after which it logs the value `Start!` to the console. The method gets popped off the call stack, and the engine continues.

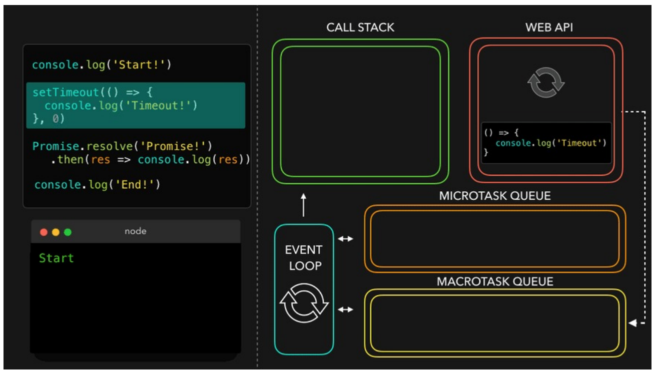


```
console.log('Start!')

setTimeout(() => {
  console.log('Timeout!')
}, 0)

Promise.resolve('Promise!')
   .then(res => console.log(res))

 console.log('End!')
```

The engine encounters the `setTimeout` method, which gets popped on to the call stack. The `setTimeout` method is native to the browser: its callback function (`() => console.log('In timeout')`) will get added to the Web API, until the timer is done. Although we provided the value `0` for the timer, the call back still gets pushed to the Web API first, after which it gets added to the **(macro)task queue**: `setTimeout` is a macro task!

. . .

The engine encounters the `Promise.resolve()` method. The `Promise.resolve()` method gets added to the call stack, after which is resolves with the value `Promise!`. Its callback function, the `then()` method in this case, gets added to the **microtask queue**.



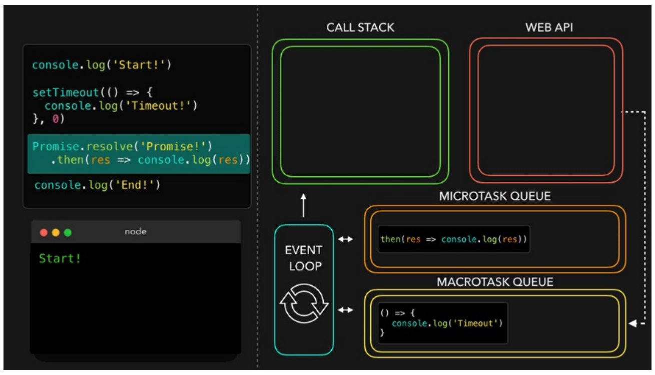
```
console.log('Start!')

setTimeout(() => {
  console.log('Timeout!')
}, 0)

Promise.resolve('Promise!')
   .then(res => console.log(res))

console.log('End!')
```

```
Start
```

```
() => {
  console.log('Timeout')
}
```

. . .

The engine encounters the `console.log()` method. It gets added to the call stack immediately, after which it logs the value `End!` to the console, gets popped off the call stack, and the engine continues.



```
console.log('Start!')

setTimeout(() => {
  console.log('Timeout!')
}, 0)

Promise.resolve('Promise!')
   .then(res => console.log(res))

console.log('End!')
```

```
Start!
```

```
then(res => console.log(res))
```

```
() => {
  console.log('Timeout')
}
```

The engine sees the callstack is empty now. Since the call stack is empty, it's going to check whether there are queued tasks in the **microtask queue**! And yes there are, the promise `then` callback is waiting for its turn! It gets popped onto the call stack, after which it logs the resolved value of the promise: the string `Promise!` in this case.

The engine sees the call stack is empty, so it's going to check the microtask queue once again to see if tasks are queued. Nope, the microqueue is all empty.

It's time to check the **(macro)task queue**: the `setTimeout` callback is still waiting there! The `setTimeout` callback gets popped on to the callstack. The callback function returns the `console.log` method, which logs the string `"In timeout!"`. The `setTimeout` callback get popped off the callstack.

Finally, all done! 🥳 It seems like the output we saw earlier wasn't so unexpected after all.

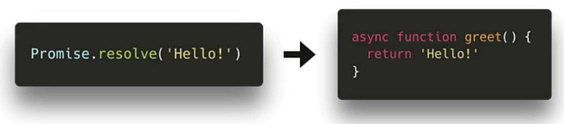
. . .

Async/Await

ES7 introduced a new way to add async behavior in JavaScript and make working with promises easier! With the introduction of the `async` and `await` keywords, we can create **async** functions that implicitly return a promise. But.. how can we do that? 😮

Previously, we saw that we can explicitly create promises using the `Promise` object, whether it was by typing `new Promise(() => {})`, `Promise.resolve`, or `Promise.reject`.

Instead of explicitly using the `Promise` object, we can now create asynchronous functions that *implicitly* return an object! This means that we no longer have to write any `Promise` object ourselves.



```
Promise.resolve('Hello!')
```

```
async function greet() {
  return 'Hello!'
}
```

Although the fact that **async** functions implicitly return promises is pretty great, the real power of `async` functions can be seen when using the `await` keyword! With the `await` keyword, we can *suspend* the asynchronous function while we wait for the `await`ed value return a resolved promise. If we want to get the value of this resolved promise, like we previously did with the `then()` callback, we can assign variables to the `await`ed promise value!

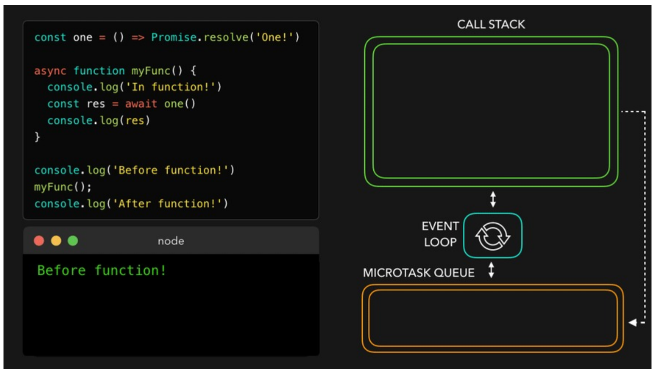
So, we can *suspend* an async function? Okay great but.. what does that even mean?

Let's see what happens when we run the following block of code:

Hmm.. What's happening here?

. . .

First, the engine encounters a `console.log`. It gets popped onto the call stack, after which `Before function!` gets logged.

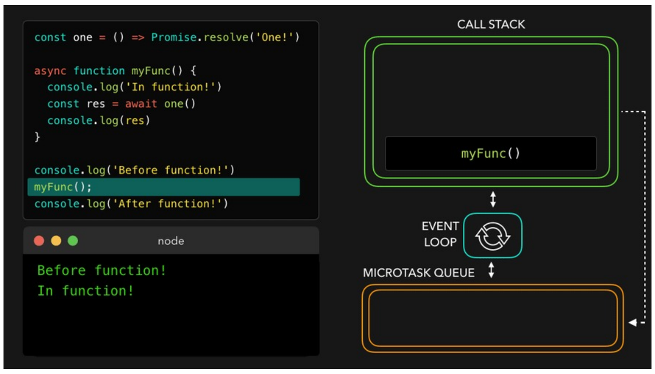


```
const one = () => Promise.resolve('One!')

async function myFunc() {
  console.log('In function!')
  const res = await one()
  console.log(res)
}

console.log('Before function!')
myFunc();
console.log('After function!')
```

```
Before function!
```

. . .

Then, we invoke the async function `myFunc()`, after which the function body of `myFunc` runs. On the very first line within the function body, we call another `console.log`, this time with the string `In function!`. The `console.log` gets added to the call stack, logs the value, and gets popped off.



```
const one = () => Promise.resolve('One!')

async function myFunc() {
  console.log('In function!')
  const res = await one()
  console.log(res)
}

console.log('Before function!')
myFunc();
console.log('After function!')
```

```
Before function!
In function!
```

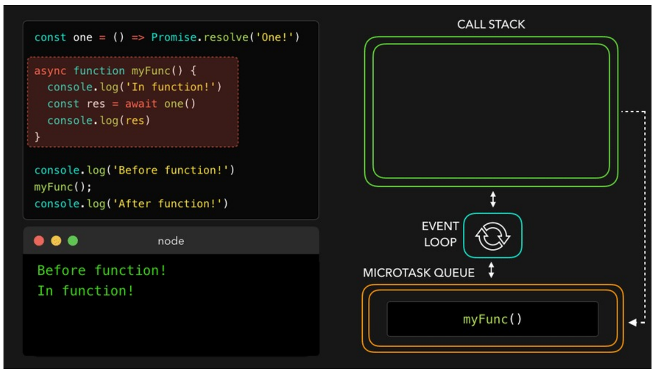
. . .

The function body keeps on being executed, which gets us to the second line. Finally, we see an `await` keyword! 🎉

The first thing that happens is that the value that gets awaited gets executed:

the function `one` in this case. It gets popped onto the call stack, and eventually returns a resolved promise. Once the promise has resolved and `one` returned a value, the engine encounters the `await` keyword.

When encountering an `await` keyword, the `async` function gets *suspended*. ✋ The execution of the function body **gets paused**, and the rest of the async function gets run in a *microtask* instead of a regular task!

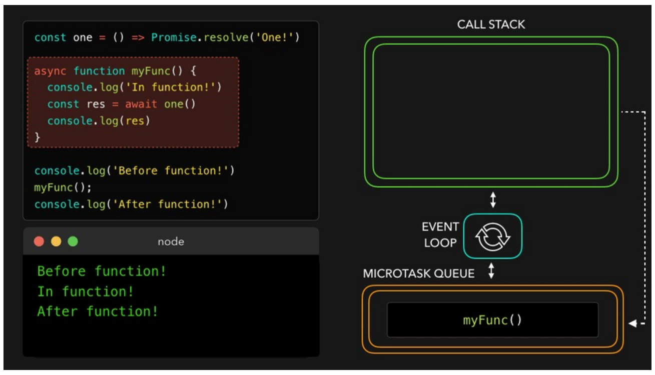


```
const one = () => Promise.resolve('One!')

async function myFunc() {
  console.log('In function!')
  const res = await one()
  console.log(res)
}

console.log('Before function!')
myFunc();
console.log('After function!')
```

```
Before function!
In function!
```

. . .

Now that the async function `myFunc` is suspended as it encountered the `await` keyword, the engine jumps out of the async function and continues executing the code in the execution context in which the async function got called: the **global execution context** in this case! 🏃‍♀️



```
const one = () => Promise.resolve('One!')

async function myFunc() {
  console.log('In function!')
  const res = await one()
  console.log(res)
}

console.log('Before function!')
myFunc();
console.log('After function!')
```

```
Before function!
In function!
After function!
```

. . .

Finally, there are no more tasks to run in the global execution context! The event loop checks to see if there are any microtasks queued up: and there are! The async `myFunc` function is queued up after resolving the valued of `one`. `myFunc` gets popped back onto the call stack, and continues running where it previously left off.

The variable `res` finally gets its value, namely the value of the resolved promise that `one` returned! We invoke `console.log` with the value of `res`: the string `One!` in this case. `One!` gets logged to the console and gets popped off the call stack! 😊

Finally, all done! Did you notice how `async` functions are different compared to a promise `then`? The `await` keyword *suspends* the `async` function, whereas the Promise body would've kept on being executed if we would've used `then`!

. . .

Hm that was quite a lot of information! 😱 No worries at all if you still feel a bit overwhelmed when working with Promises, I personally feel that it just takes experience to notice patterns and feel confident when working with asynchronous JavaScript.

However, I hope that the “unexpected” or “unpredictable” behavior that you might encounter when working with async JavaScript makes a bit more sense now!

And as always, feel free to reach out to me! 😊

✨ Twitter 👩🏽‍💻 Instagram 💻 GitHub 💡 LinkedIn 📷 YouTube 💌 Email

If you want to know more about promises **states** (and **fates**!), this Github repo does an excellent job explaining the differences.

Recommended from Medium

Intspirit Ltd

Understanding the browser's Event Loop for building high-...

A deep dive into the Event Loop: task priorities, dealing with long tasks, using the...

8 min read · Sep 12, 2023

381

Rabail Zaheer

Understanding JavaScript Closures

JavaScript Closures are a fundamental and powerful concept that play a pivotal role in...

10 min read · Sep 28, 2023

36 2

Lists

Stories to Help You Grow as a Software Developer
19 stories · 889 saves

General Coding Knowledge
20 stories · 999 saves

Generative AI Recommended Reading
52 stories · 803 saves

Visual Storytellers Playlist
58 stories · 240 saves

Navneet Singh

setTimeout vs setImmediate in JavaScript

JavaScript provides two functions, setTimeout and setImmediate, for schedulin...

· 3 min read · Oct 2, 2023

12 2

devtalib

Lexical environment in JavaScript

A lexical environment in JavaScript is a data structure that stores the variables and...

2 min read · Oct 16, 2023

58

Satria Suria

Event propagation in JavaScript

What is event propagation?

3 min read · Oct 9, 2023

2

October

Synchronous and Asynchronous Programming in JavaScript :...

JavaScript is synchronous, blocking and single-threaded. With asynchronous...

3 min read · Feb 26, 2024

50

See more recommendations