Callbacks vs. Promises vs. Async Await: A Step by Step Guide

And a bit under the hood.

Anton Franzen Feb 17 · 8 min read ★

I found this confusing and thought you might, too, like callbacks vs promises vs async, what to use? what is it even? how does each one work? how does it work under the hood? that's what I want to explain in this post once for all.

into another function that accepts another function as an argument. And that function we pass in can be invoked at any time in the future by the function we pass it into. It is then called a `higher order function` , which accepts a function as an argument.

This is a callback:

```
function someFunctionAcceptingCallback(number, callback){
return callback(number + 10)
}
function divide(answer) {
return answer / 2

}

someFunctionAcceptingCallback(5, divide) // 7.5 if we console.log it
```

Another good example is a `addEventListener` in javascript.

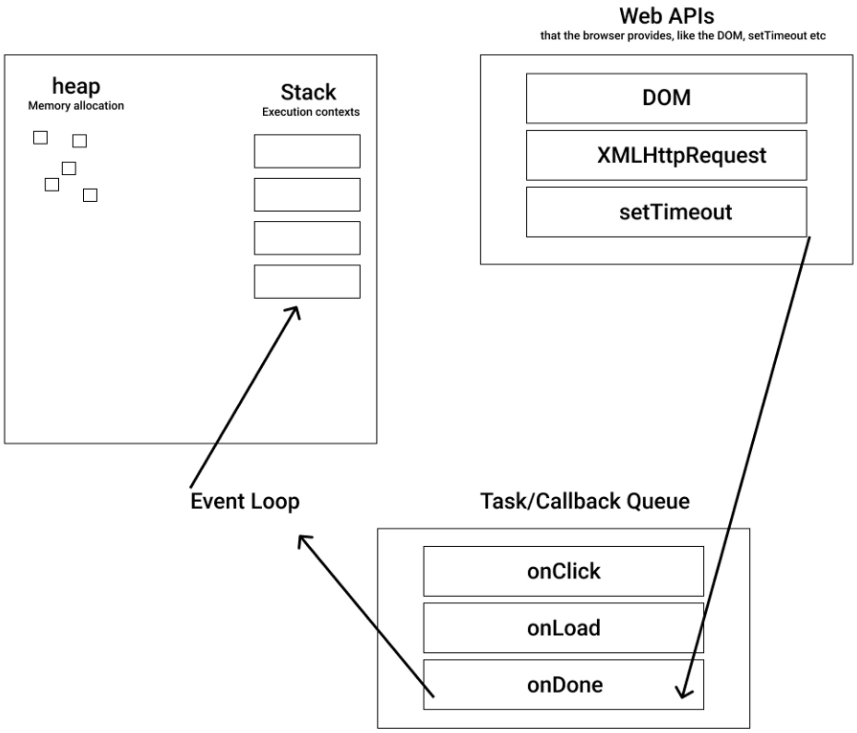
```
document.getElementById('addUser').addEventListener('click',
function() {// Do something})
```

So what happening here is that the `addEventListener` allows us to wait with the execution until a particular moment later on, in this case when the `addUser` button gets

But how does it even work?.. Javascript is single-threaded meaning it can't execute multiple codes at the same time, javascript has a `call stack` which runs 1 task at a time from top to bottom.

So when we add our `addEventListener` , we don't really call a javascript natively method, we call a method in the WEB API. And the WEB API we can visualise as another thread basically.

So the WEB API `addEventListener` is a DOM event, it waits for our button to be clicked, then when it gets clicked, it takes our `callback` function we passed as an argument, and push it to something called the `task queue/callback queue` , and then something called the `event loop` picks the callback function up and push it to the `call stack` and it gets executed, but only when the callback function is the first function in the `task queue/callback queue` . I made a visual example of how it works:



What is a perfect example of good use of a callback function?.. You guessed it right, network requests.

But as javascript is single-threaded, there was no way to do this natively in javascript before promises was introduced in ec6.

What would have been happening if we would call the network request without using any WEB API, is that our code would get blocked, the `call stack` wouldn't execute any code before our network requests were done. This is called code blocking, it happens if our code is synchronous and takes a long time to execute, then our web page would have been frozen, no javascript code would have been executed as we call the network request directly on the call stack and it blocks the rest of the javascript code to execute.

So the way we have solved this issue was to make use of another WEB API called `XMLHttpRequest .`

The `XMLHttpRequest` is a WEB API, we pass in our callback function, once the `XMLHttpRequest` is done, it pushes it to the `task queue/callback queue` and the `event loop` will pick it up, and push it to the `call stack` for it to execute our callback function.

Here is an example:

All good right? We can make network requests asynchronous by using the WEB API and pass callback functions it should execute once we got a response from the network request.

The catch

Callback hell

If we would like to make network requests in order, so the first one, then the second one, and so on, it will get real messy, let me show you:

It is hard to read as brains is used to think in a sequential order, it is error prone, and can get way worse, we need to check for an error in each request also, so the final code will end up much longer and less readable and therefore more error prone and harder to maintain.

How was this solved? There is some neat ways to make it better, and also there was plenty of libraries accessible to use for escaping callback hell, and make the code just better.

Now let's talk about Promises, what they can do and how they work.

Promises

What is a promise then?

Before promises were introduced into native javascript we needed to use other methods and libraries for dealing with async programming, there was simply no way to use plain native javascript for that.

When promises were introduced it was a huge thing for the javascript developers, now you could write async network requests with ease and escape callback hell.

Before we talked about how `XMLHttpRequest` was a WEB API, because javascript is single-threaded and we must pass our async functions to something else than the `call stack` for avoiding code blocking.

Micro queue

This was introduced with ec6, basically, it is like the `micro queue` but for Promises and it differs a little bit. The `micro queue` have priority over the `callback queue` , so for example if we would run this code:

Then, even tho our `setTimeout` will run the callback directly after 0 seconds and the Promise is set out to do so too, which is immediate, the promise will run its callback first because it using the `micro queue` and that has priority over the `callback queue` which the `setTimout` WEB API method is using.

Let's dig into the Promise, promises has 3 states, `pending fulfilled rejected`.

`pending` means the promise has not been resolved or rejected, in context to an API request, when we make a request and the server have not sent back any response, that's when the promise would be in pending state.

`fulfilled` means the promise has resolved, in context to an API request, when we get back a successful response, that's when the promise would resolve.

`rejected` means our promise has been rejected, in context to an API request, when we get back a 404 response, that's when the promise would reject, and we could `catch` the error.

We can for example create a new promise with a timeout of 3 seconds, and after 3 seconds have passed, we can resolve the promise. What that means is that we can do stuff after something has been completed in pure native JavaScript without using a library for asynchronous executions. And this applies to any execution that, which we don't know when we will get back a response.

After the promise have been resolved we can do `.then()` and whatever we have inside `.then()` will only be executed when the promise resolves. To demonstrate this let's make a promise that is resolved after 3 seconds that pass a string to the resolve:

We can wrap a `fetch` in a promise too:

`Fetch` is promise-based also, so when we call `.then()` on the `fetch`, what we really are doing is waiting until the server resolves the promise/return the response.

Chaining

Instead of callback hell, we can chain promises in different ways, example, if we would like to make another request once our first request, has resolved we can do this:

call a function `doSomething()` with our JSON data and then we log that we are done.

Promise.all

Promises have something called `Promise.all` which allows us to wait for any number of promises to resolve, and then execute a code block. This can be very very useful, for example, if we would have an array of requests we need to call and wait for all requests to finish, then we can do this in a few lines of code like this:

Async awaits

is not really sequential as we would like it.

That's what async-await is all about.

Async await is a new way to write asynchronous code and was basically created for simplifying how we can write chained promises.

Async await is nonblocking like we would expect it to be as it is asynchronous, and each async-await is returning a promise with its resolved state.

So with Promise chaining, this is what we do:

This is the thought behind async-await.

So the accurate way to write async-await is this:

Instead of we doing `fetchData().then(resolvedData)...` we just need to do `const data = await fetchData()` and the `data` will hold the resolved data, and the next line won't execute until `fetchData()` has resolved it's promise.

One thing to remember is that when we have an async function, that is basically as having a promise, in fact, the whole function is a promise. And we do the same as when a promise resolves for the async function, So what we could do is `asyncFunction.then()` to execute code after the function has resolved its promise, which happens after all of our `awaits` has resolved and we return from the function.

So what will happen in our `asyncFunc()` is that first it will make the `fetchData()` request and wait until the promise resolves from the server. Then when it resolves, it will call the `fetchUserData()` request and wait until the promise resolves from the server. Then, only then it will update the `userState` .

And if we would like to call this function and wait until it has done all of those steps, we just do `asyncFunc().then()` , the same we did for our promise, because async is a promise, and when we return something from the function it is basically the same as we resolve a promise.

up, such as how to catch errors, optimization techniques, and so on. But I think this will give you a basic grasp of what the difference between the 3 is, how they work, how you can use them.

You'll probably also like:

EC6 Magic You Wish You Knew
A few neat ec6 tricks for faster coding
medium.com

Boost the SEO of Your Websites With Next.js
How to add keywords, meta tags, and more
betterprogramming.pub

Machine Learning Engineering Is HARD
People make this look easy but it's brutally hard...
medium.com

Hey my name is Anton 👋, I'm a freelance front-end developer from Sweden.

Feel free to contact me for any front-end work or a remote position.

Thanks to Anupam Chugh.