Async JS

Event loops, message queues,, callbacks and promises :)

Who am I?

- Still Dannie
- Still working at Klarna
- Still 27 years old
- Here to teach you… again

😜

The agenda of today

- Sync vs async JS
- What makes async js possible?
- From callbacks to promises and async/await
- Some examples in the mix 😬
- Key takeaways

Before we start

This is usually one of the hardest topics for JS developers to understand. Having said that, I would recommend two things:

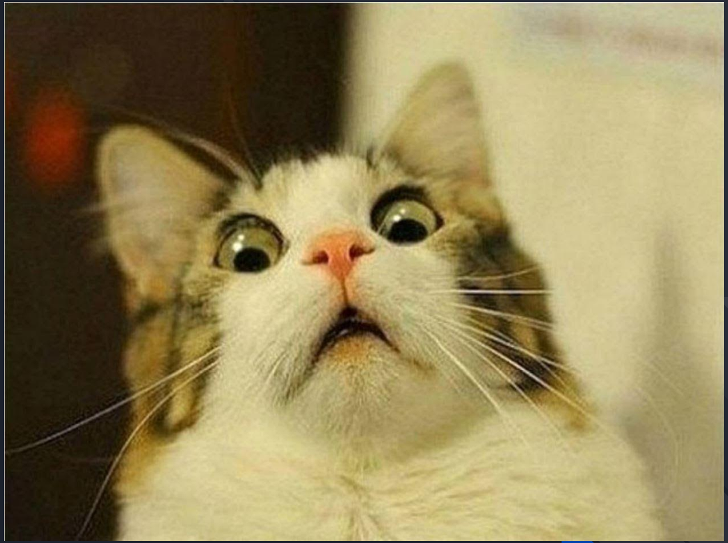
1. Be open minded about the fact that it will take time for this to really sink in.
2. Ask as many questions as you can 👀

Truth is, there are many people in the professional field of frontend that don't really grasp how **Async JS** works.

That said, I hope I can provide you with the foundations to become familiar enough with it.

Because it is **everywhere** 😬

LET’S BEGIN!



Synchronous JS is like...

You are in a queue to get a movie ticket. You cannot get one until everybody in front of you gets one, and the same applies to the people queued behind you.

Synchronous JS is...

A program that runs top to bottom. An instruction or task is run from start to finish before starting a new one.

“The way the program is written is also how it runs”

When **A** runs, it has to finish before **B** can start, and **B** has to finish before **C** can start.

Very deterministic and understandable

```
|--------A--------|
                  |--------B--------|
                                    |--------C--------|
```

/1-sync-example

Asynchronous JS is like...

You wake up and go to the kitchen. You start making coffee. Before the coffee is finished you start making oatmeal. While the oatmeal is cooking, the coffee finishes, so you pour it into your cup. When the oatmeal is finished you start eating.

Asynchronous JS is...

A program that does NOT run top to bottom. An instruction or task being run from start to finish is not dependent on the previous one being finished(we can start with the oatmeal before the coffee has finished brewing).

When **A** runs, it does NOT have to finish before **B** can start, and **B** does NOT have to finish before **C** can start.

Not as deterministic and understandable

```
|--------A--------|
      |--------B--------|
            |--------C--------|
```

How does **sync** and **async** *really* relate to **JS** though?

Enter the world of...

Call Stack, WebAPIs, The Event Loop and Message Queue

The Call Stack

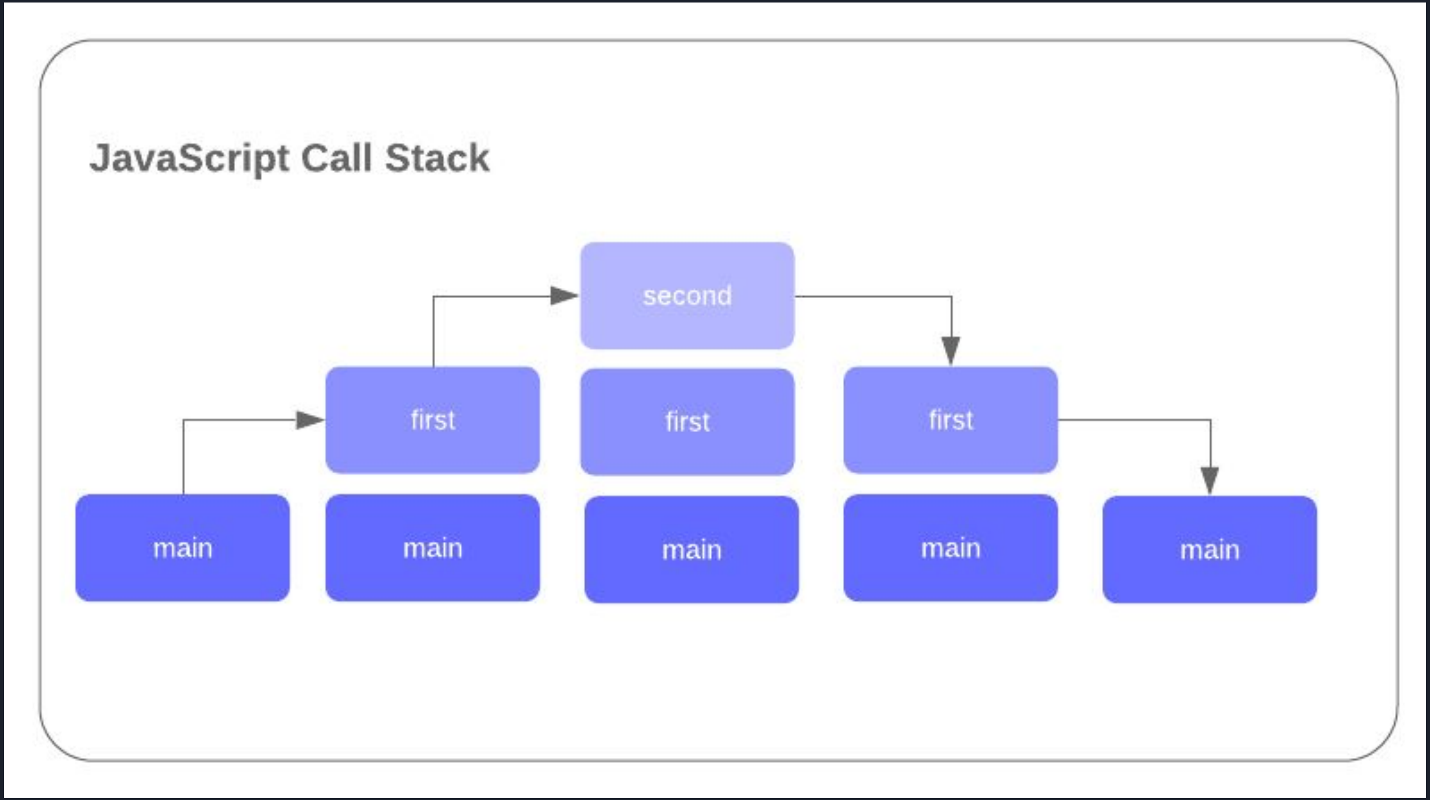
The JavaScript engine in the browser implements what’s called a **call stack**.

When you write a line of code, it will run on this call stack.

Runs on a “single thread”, which means only one thing can happen at once.

The code that you write gets put on the call stack, and when it finishes it is removed from it.

The Call Stack



The Call Stack

WebAPIs

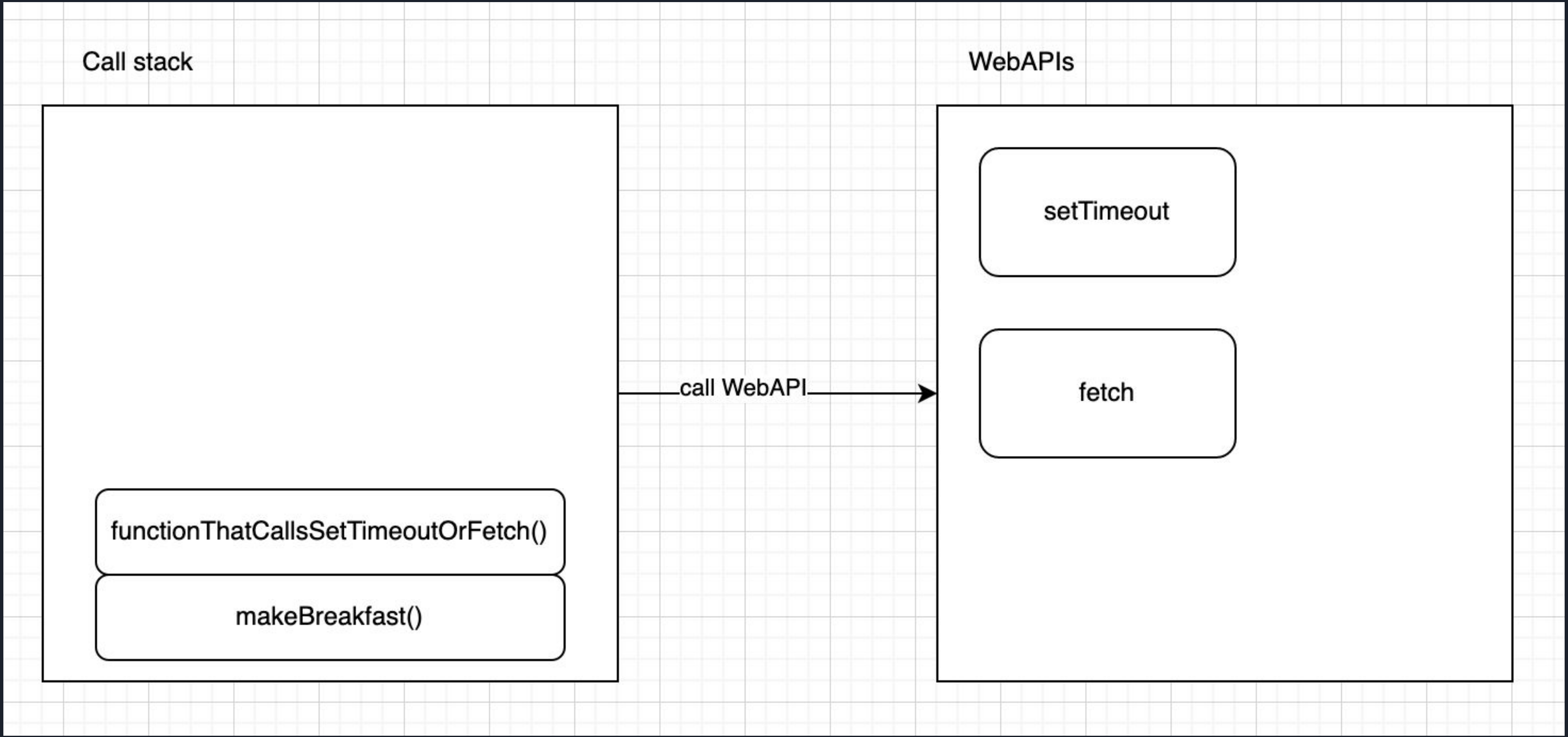
The browser provides what’s called **WebAPIs**.

setTimeout and **fetch** are two very common examples, that some of you might be aware of already.

Something running on the **call stack** is responsible for using the **WebAPIs**. The way to use them is to provide a **callback function**. Remember that **callbacks** is an important concept we will cover.

The mentioned **WebAPIs** run asynchronously, **outside of the JS runtime environment**. Yes, even outside of the **call stack**.

We will get to how this is made possible.


Call stack
WebAPIs
setTimeout
fetch
call WebAPI
functionThatCallsSetTimeoutOrFetch()
makeBreakfast()

The **WebAPIs** accept a function to be run when applicable...

For example:

setTimeout: When this amount of time has passed, I want to run ***this function***

fetch: When this data is returned, I want to run ***this function***

What is ***this function***?

Callbacks, the earliest way to express async behavior

Callbacks, the earliest way to express async behavior

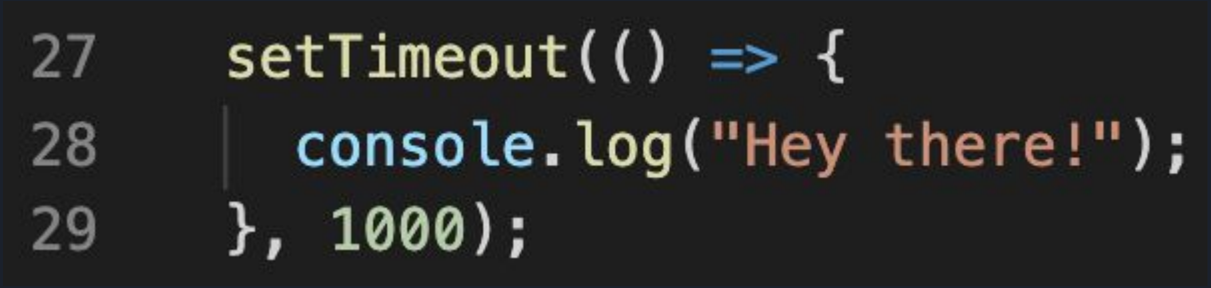
Also called **asynchronous callbacks**

Is a function that is passed to an async function(like **setTimeout**) to be called “later”.

The **WebAPI** pushes the **callback** onto a what’s called the “**Message Queue”** whenever it is time for it to be executed.

We will soon look at how the **WebAPIs** relate to the **Message Queue :)**

setTimeout ⏰



```javascript
setTimeout(() => {
  console.log("Hey there!");
}, 1000);
```

/2-async-example-set-timeout

Like mentioned, the JS runtime is single threaded, meaning only one thing(instruction/task) can happen at once. It might sound like this makes everything synchronous, but as you saw in the previous example, this is not the case.

Sorry, what!?

The Event Loop and the **Message Queue**

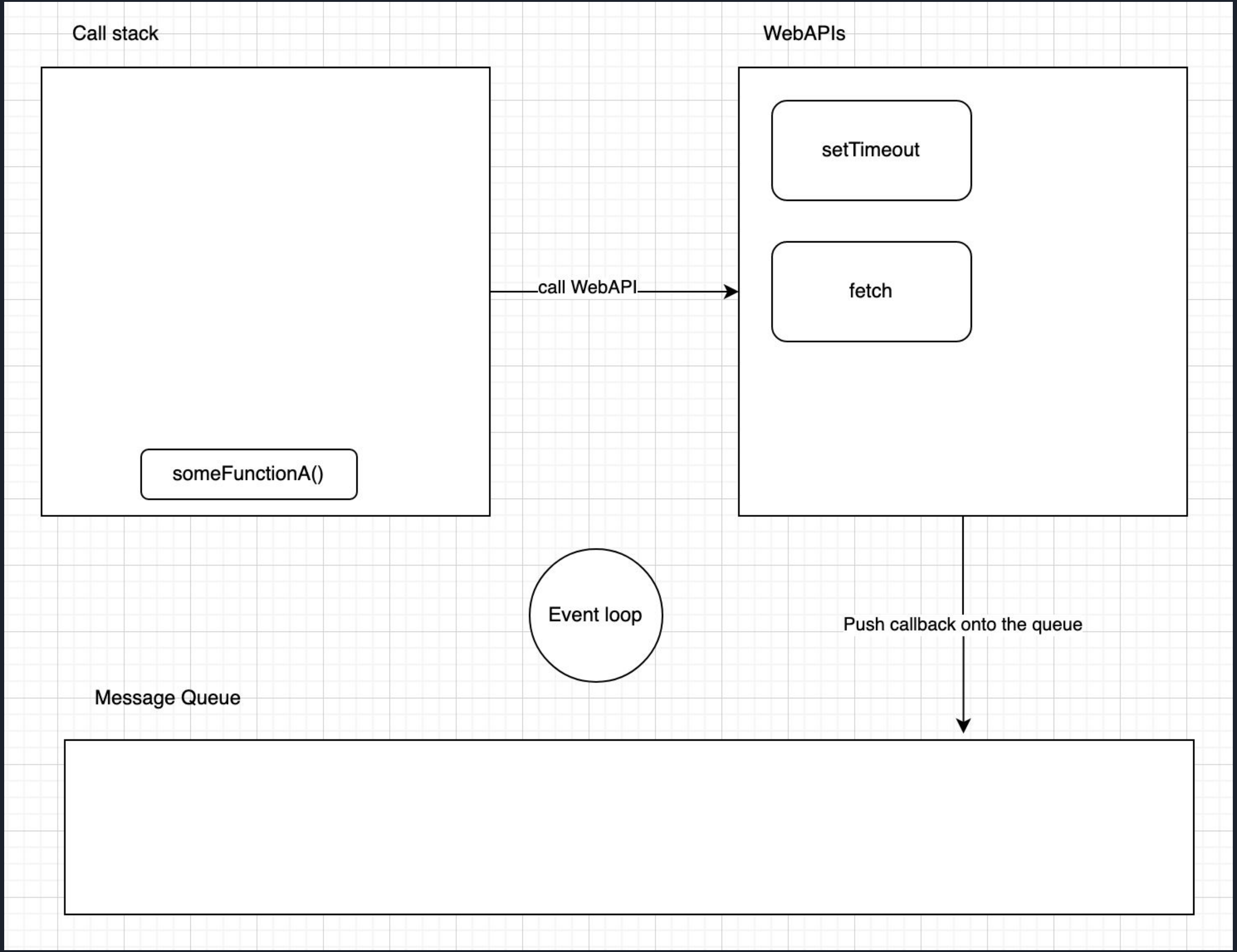
The Event Loop is implemented in the browser as a mechanism to handle the execution of synchronous as well as asynchronous code.

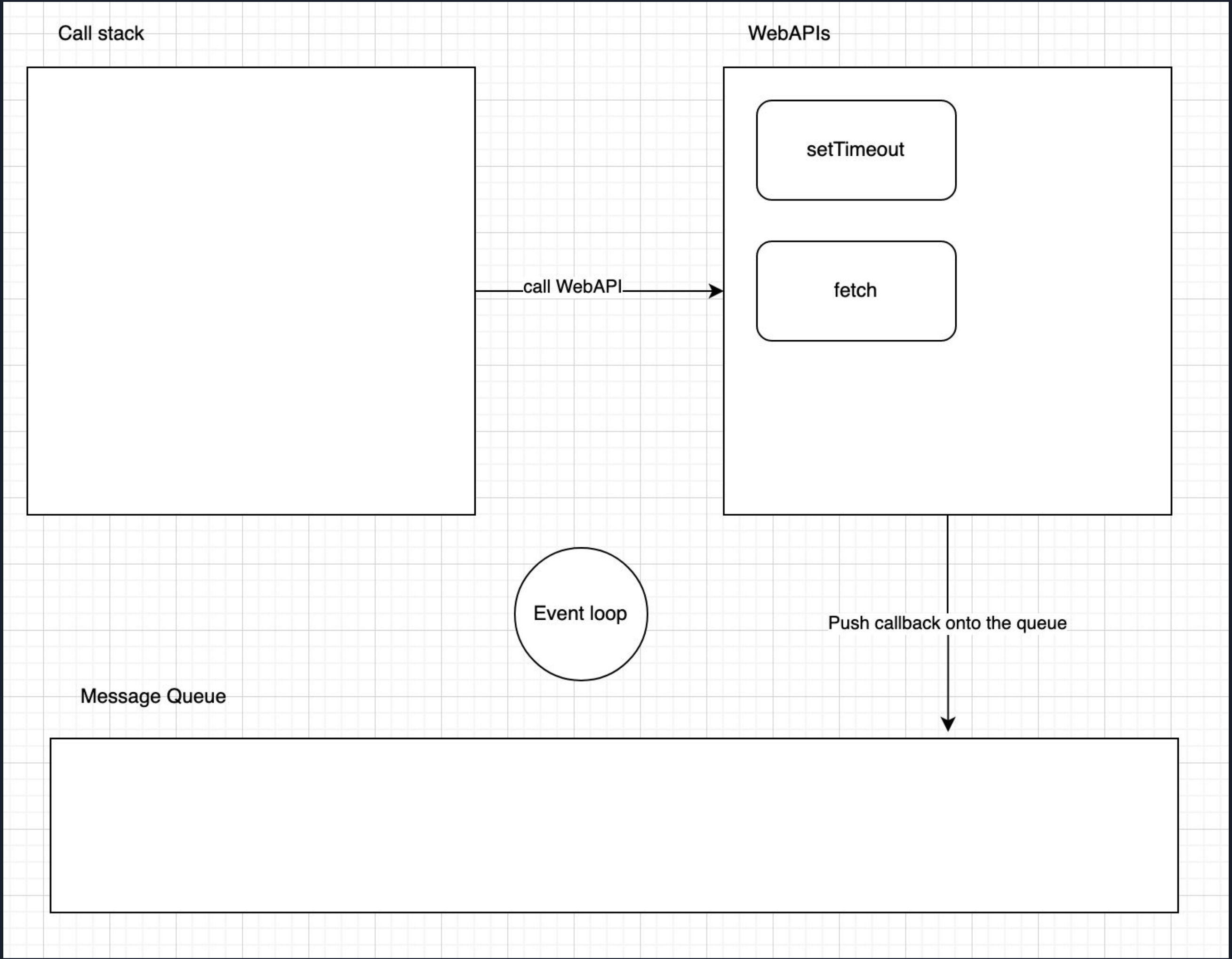
It is responsible to make sure that every function call is run in the **call stack**. Remember, everything in the **JS runtime** runs in the **call stack**.

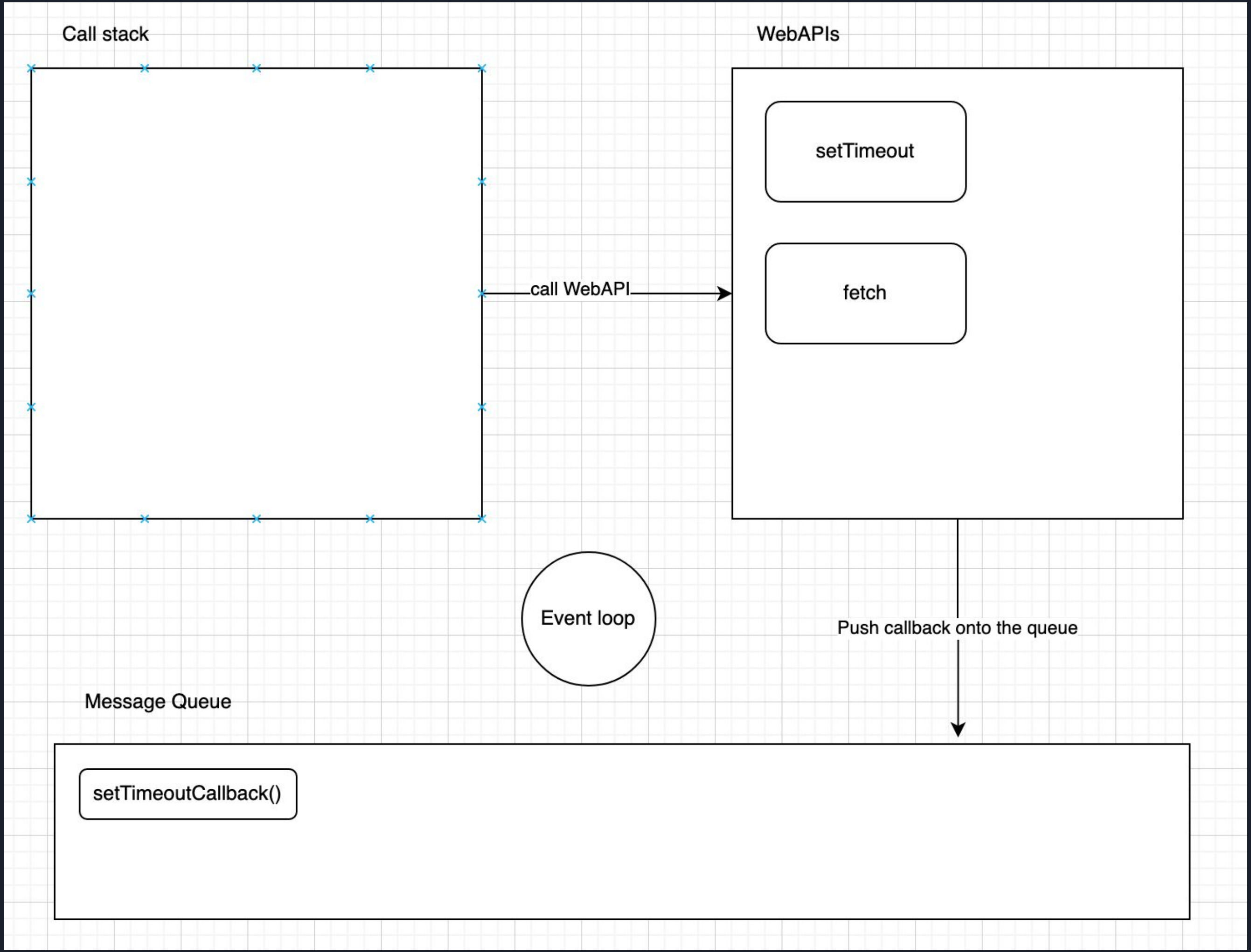
When the call stack is empty, *it moves messages from the* ***Message Queue*** *and onto the call stack to be executed.*

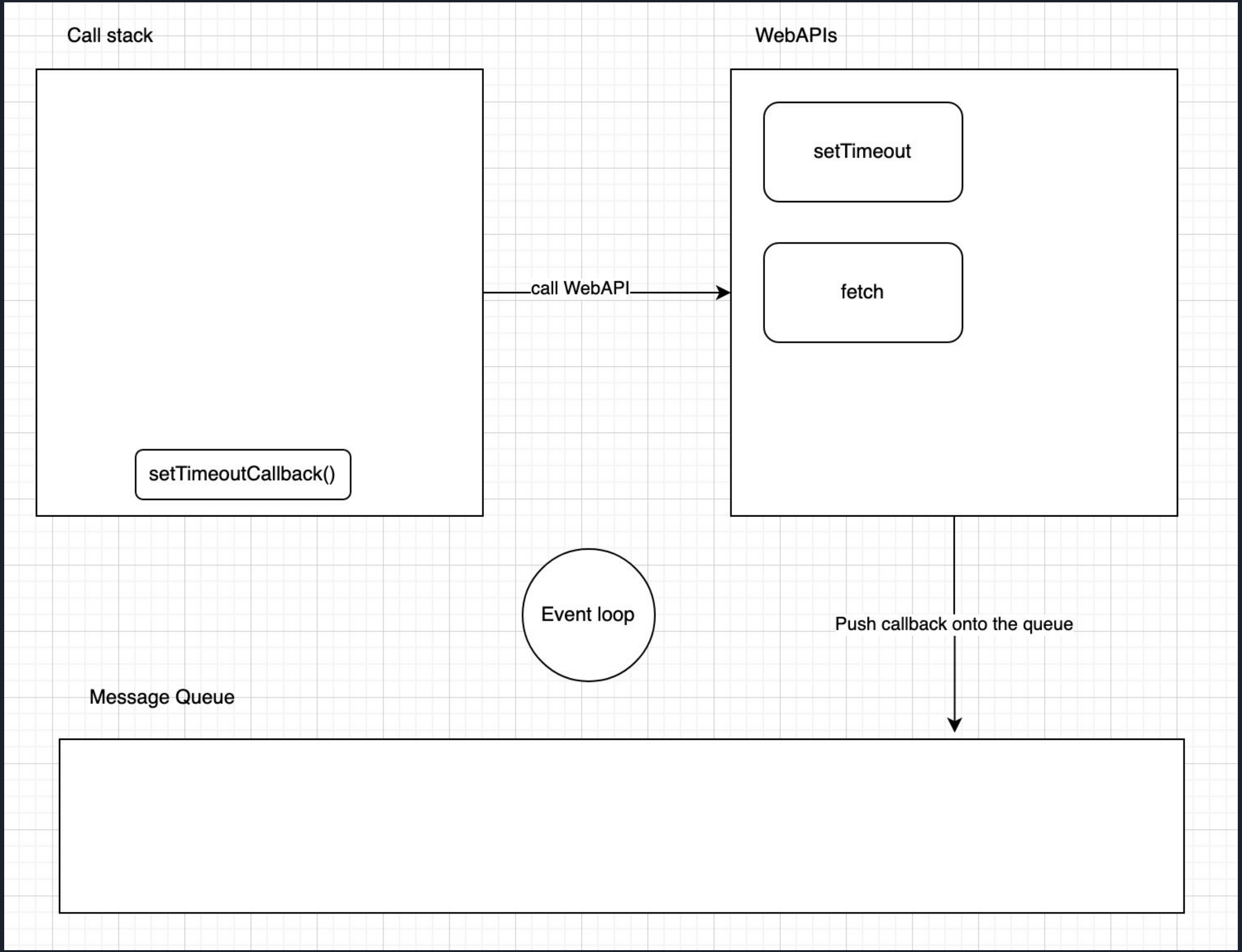
A message could be a **callback** that was passed to **setTimeout** earlier.

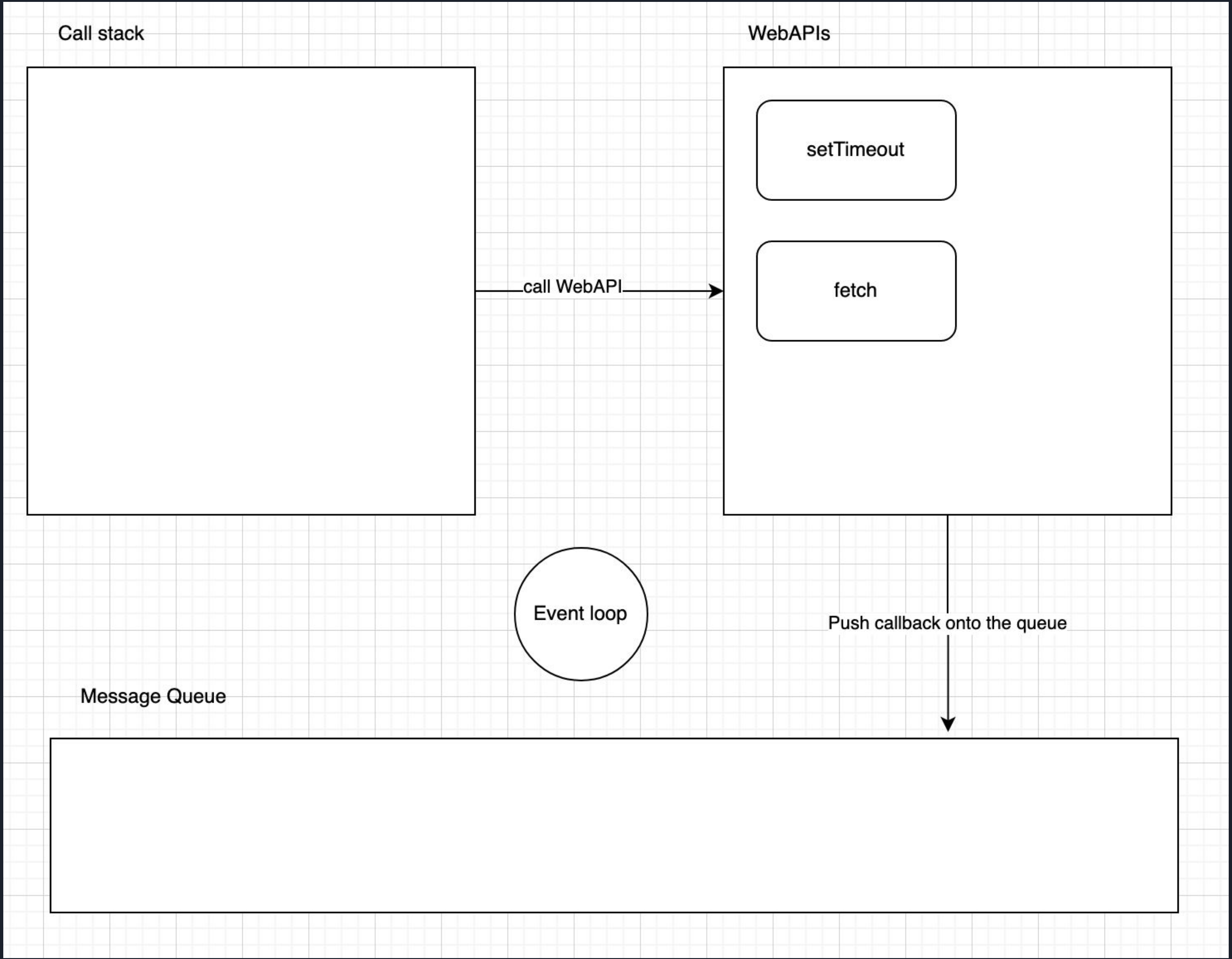
This is what makes “everything happens at once” possible.


Call stack
WebAPIs
setTimeout
fetch
call WebAPI
someFunctionA()
Event loop
Push callback onto the queue
Message Queue


Call stack
WebAPIs
setTimeout
fetch
call WebAPI
Event loop
Push callback onto the queue
Message Queue


Call stack
WebAPIs
setTimeout
fetch
call WebAPI
Event loop
Push callback onto the queue
Message Queue
setTimeoutCallback()

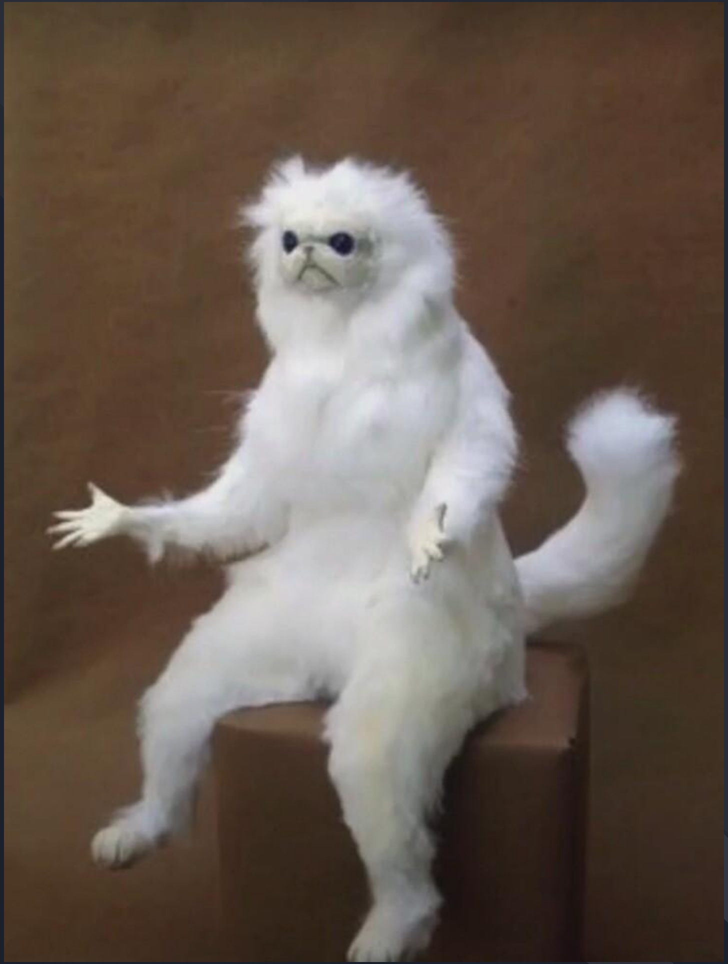

Call stack
WebAPIs
setTimeout
fetch
call WebAPI
setTimeoutCallback()
Event loop
Push callback onto the queue
Message Queue


Call stack
WebAPIs
setTimeout
fetch
call WebAPI
Event loop
Push callback onto the queue
Message Queue

The fact that your browser can listen for click events, while rendering newly applied CSS styles, while downloading an audio file, while animating a cat that is spinning 360 degrees while moving from left to right, is all made possible by

The Call Stack, Event Loop and Message Queue.

Async JS 🤘



Callback hell and why it sucks

/3-callback-hell

Promises, the better way to express async behavior

What are **Promises**?

Is a construct, like **callbacks**, meant to handle asynchronous code, albeit in a cleaner way(you will see why soon).

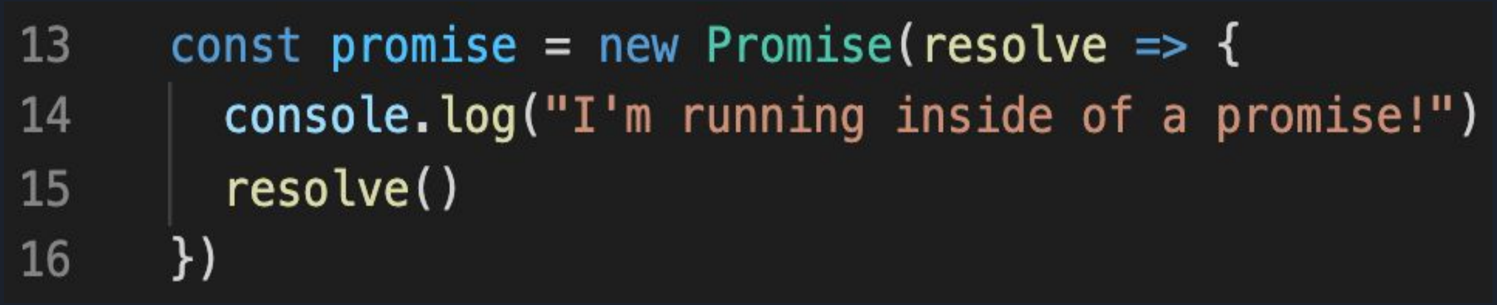
It’s an object that you create by calling its constructor, **Promise**.

“The **Promise** object represents the eventual completion of an asynchronous operation”

The **Promise** constructor accepts one parameter:

- The **executor** function that will run the promise.

What are **Promises**?



```
const promise = new Promise(resolve => {
  console.log("I'm running inside of a promise!")
  resolve()
})
```

What are **Promises**?

A promise can be in one of three possible states:

1. *pending*: initial state, neither fulfilled nor rejected.
2. *fulfilled*: meaning that the operation was completed successfully.
3. *rejected*: meaning that the operation failed.

When a **promise** is created, it is in the pending state. When it “settles”, it transitions into either fulfilled or rejected.

What are **Promises**?

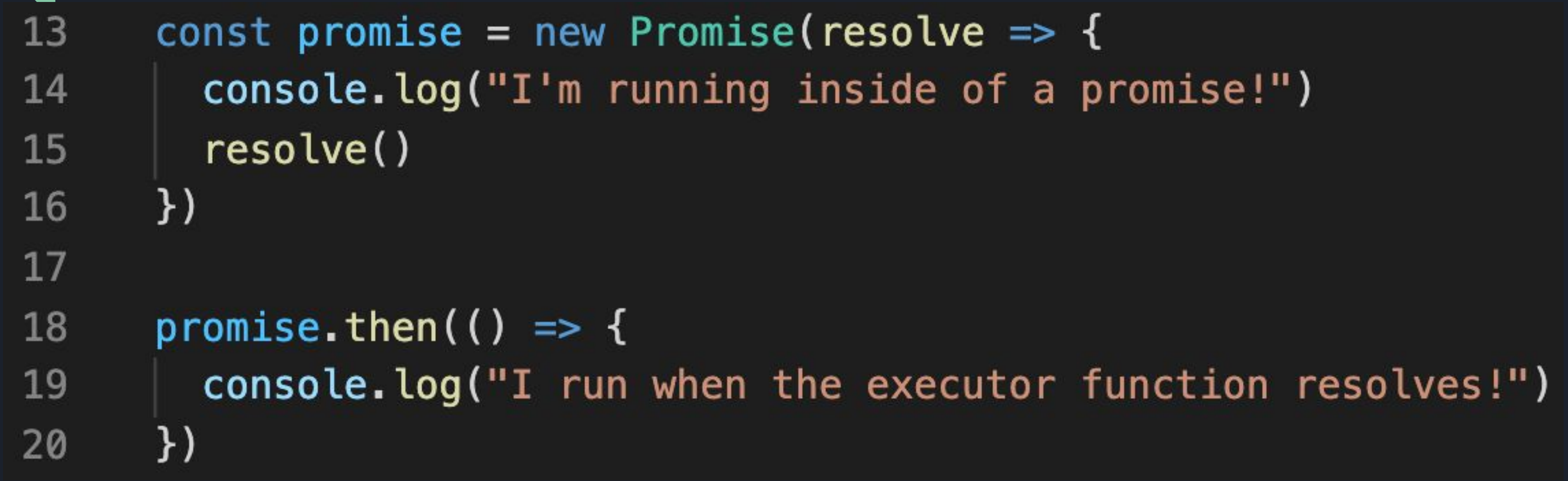
When you have created your **promise**, you have access to a function called **then**, which you can call on the **promise** object.

When you call **then,** you have to pass one argument:

- A function that will run when the **promise** “resolves” or “completes”

When the **promise** resolves, the function passed to **then** will be put on the **Message Queue,** much like the **callbacks** we looked at earlier.

What are **Promises**?



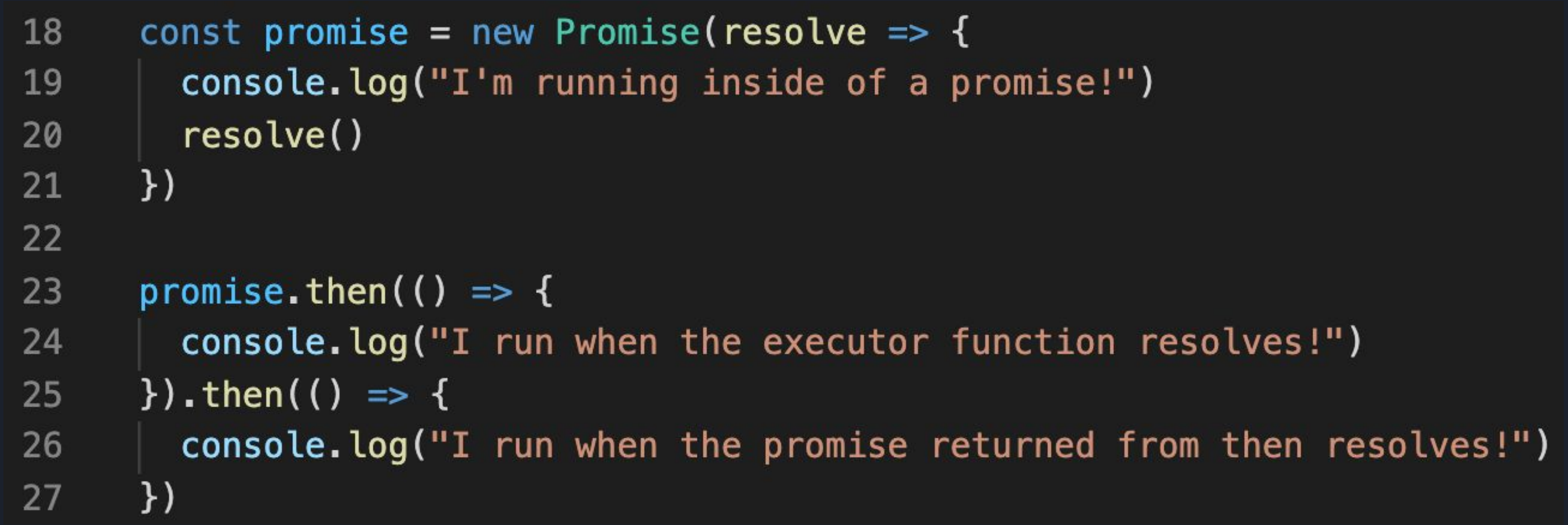
```
const promise = new Promise(resolve => {
  console.log("I'm running inside of a promise!")
  resolve()
})

promise.then(() => {
  console.log("I run when the executor function resolves!")
})
```

What are **Promises**?

In the same way you can call **.then** on a **promise**, you can also call **.then** on **.then**. How is this possible?

It’s made possible by the fact that **then** returns a **promise**.

What are **Promises**?



```javascript
const promise = new Promise(resolve => {
  console.log("I'm running inside of a promise!")
  resolve()
})

promise.then(() => {
  console.log("I run when the executor function resolves!")
}).then(() => {
  console.log("I run when the promise returned from then resolves!")
})
```

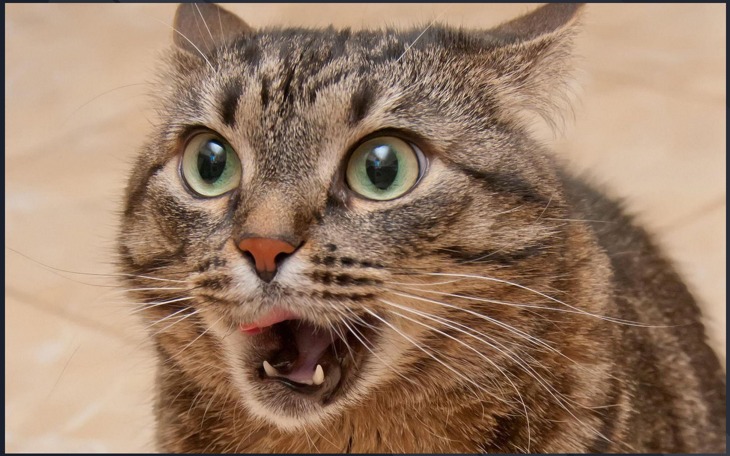
But why do we want to use **Promises**?

Why use **Promises**?

They are more readable than **callbacks.** They are more readable because they can be *chained***:**

"When this is resolved, I want this to run, and when that second thing resolves, I'm done" is much easier to express with **promises** than with **callbacks**. You will see why soon.

Let’s look at how we can improve our callback hell scenario with promises



/4-callback-hell-solved

A lot of APIs that you will be using will be **Promise** based

One example is **fetch**, which you have all probably used by now.

/5-fetch

Async/await, the even better way to express async behavior

What is **async/await**?

The reality is, **callbacks** can get messy, but so can **promises**.

As a result, a newer mechanism built on top of promises, which *hides the underlying details* of ***promise*** *execution*, was implemented.

It’s basically “a better way to write and use **promises**”.

What is **async/await**?

The concept is that you declare an **async** function, which will return a **promise** implicitly.

This **async** function will then have access to the keyword **await**.

await can be placed in front of promises “in order to await their resolve”.

From the eyes of **JS** and the browser, **async/await** looks exactly like regular **promises**.

It’s built for you as the developer, for readability.

/6-callback-hell-better-solved

Main takeaways

1. You might be very confused right now. That is OK :)
2. You will encounter **callbacks, promises** and **async/await** in your forthcoming careers A LOT. Try to become comfortable with them.
3. **Promises** are the new standard, and old API's written in the "callback style" can, as you saw an example of, be rewritten to use **promises** instead.
4. You will be able to work with all of this without understanding everything of what you've just been taught about what is happening behind the scenes.
5. Revisit these slides when you feel like you need to really get a better grasp of the peculiar asynchronous nature of JS.

If you want to dive even deeper

Error handling is a concept very related to **async js**. Feel free to google this concept on how it has evolved in tandem with **callbacks, promises** and **async/await**.

More to read/watch

The JS event loop => https://flaviocopes.com/javascript-event-loop/

What the heck is the event loop anyway? => https://www.youtube.com/watch?v=8aGhZQkoFbQ

Using async/await => https://javascript.info/async-await

If you haven’t already connected with me

danniehakansson.com

linkedin.com/in/danniehakansson/

danniehakan@gmail.com

Thank you! 🙌