Node.js Features

Mohamed Mukthar Ahmed

1

Node.js Features

Outline

- Node.js Killer Feature
- Blocking
- The Call Stack
- The Event Loop
- Callback Queue
- Promises!
  - Syntax, Promise Status
  - Invoking response, reject Callbacks
  - Methods
  - Chaining .then Methods
- Microtasks and Macrotasks
  - Priority to tasks
  - Microtasks Queue
- Async Function
  - Using async and await features

Mohamed Mukthar Ahmed

2

Node.js Killer Feature

- The **Event Loop** is one of the most **important** aspects to understand about **Node.js**.
- Why is this so important?
  - Because it **explains** how Node.js can be
  - **Asynchronous** and
  - Have **non-blocking I/O**.
- It explains basically the "**killer feature**" of Node.js, the thing that made it this **successful**.

3

Node.js Event Loop

- The Node.js JavaScript code runs on a **single thread**. There is just one thing happening at a time.
- This is a **limitation** that's actually very helpful, as it **simplifies a lot how you program** without worrying about concurrency issues.
- You just need to pay attention to how you write your code and avoid anything that could block the thread,
  - Like synchronous network calls or
  - Infinite loops

4

Node.js Event Loop

- In most browsers there is an **event loop** for every browser tab.
- Done to make every process isolated and avoid a web page with
  - Infinite loops or
  - Heavy processing to block your entire browser.
- The environment manages multiple concurrent event loops, to handle API calls.
- You mainly need to be concerned that your code will run on a single event loop and write code with this thing in mind to avoid blocking it.

5

Blocking the event loop

- Any JavaScript code that takes **too long** to return back control to the event loop will
  - Block the execution of any JavaScript code in the web-page,
  - Even block the UI thread, and the user cannot click around, scroll the page, and so on.
- Almost all the
  - I/O primitives in JavaScript are non-blocking.
  - Network requests, filesystem operations, and so on.
- Being blocking is the exception, and this is why JavaScript is
  - based so much on **callbacks**, and
  - more recently on **promises** and **async/await**.

6

The Call Stack

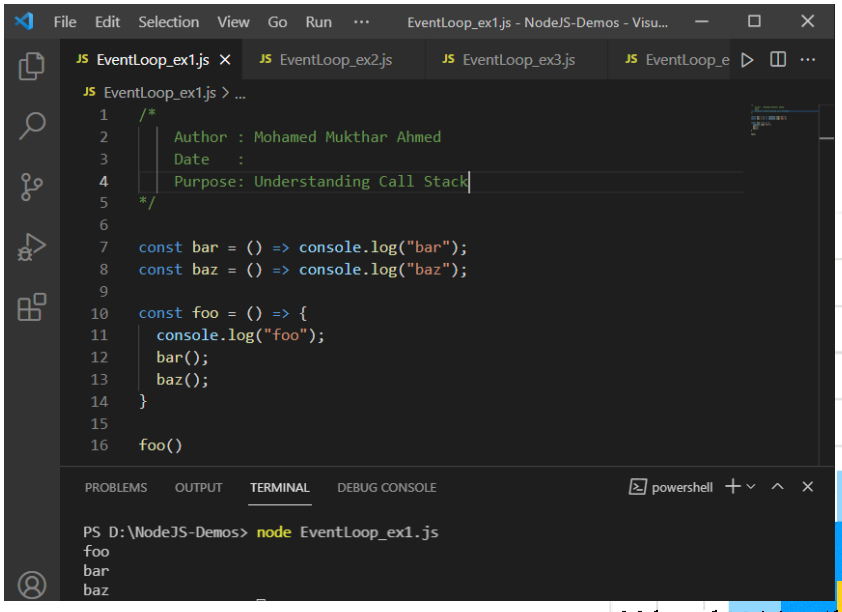
- The call stack is a **LIFO** (**Last In, First Out**) stack.
- The event loop continuously checks the call stack to see if there's any function that needs to run.
- While doing so, it adds any function call it finds in the call stack and executes each one in order.

```
const bar = () => console.log("bar");
const baz = () => console.log("baz");

const foo = () => {
  console.log("foo");
  bar();
  baz();
}

foo()
```

Mohamed Mukthar Ahmed

7

The Call Stack



Mohamed Mukthar Ahmed

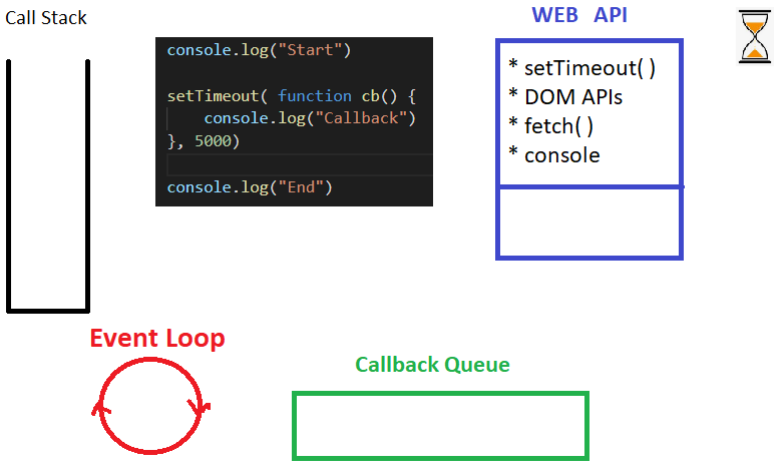
8

The Event Loop

- **Call stack** will execute any execution context which enters it.
- **Time**, **tide** and **JS** waits for **none**.
- Call stack has no timer.
- Browser has JS Engine which has Call Stack
- But browser has many other superpowers –
  - Timer, Location, Geolocation access
  - Local storage space, place to enter URL, etc
- JS needs some way to connect the callstack with all these superpowers.
  - This is done using **Web APIs**.

9

The Event Loop



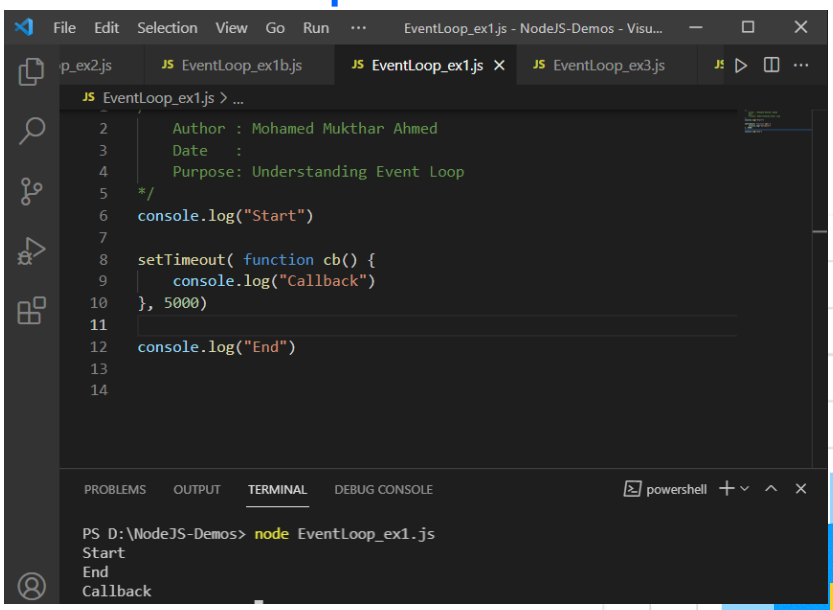
10

Code Explanation

- console.log("Start");
  - This calls console **WEB API** and logs in console window
- setTimeout(function cb() {
  - This calls the setTimeout **WEB API** which gives access to TIMER.
  - It stores the callback function cb() and starts timer
- console.log("End");
  - This calls console **WEB API** and logs in console window.
- While all this is happening, the **TIMER** is constantly ticking. After it becomes 0, the callback cb() has to run.
- Now we need this cb() to go into call stack. Only then will it be executed. For this we need **Event Loop** and **Callback Queue**.

11

The Event Loop

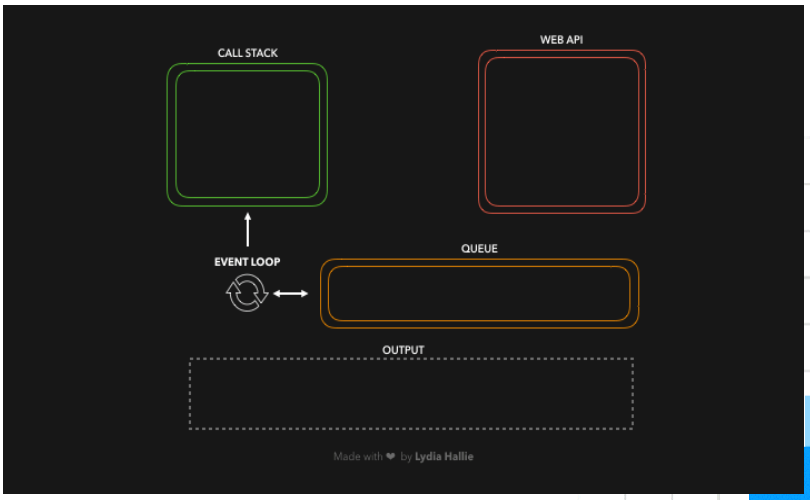


12

Event Loop & Callback Queue

- The callback **function cb()** cannot directly go to call stack for its execution.
- It goes through the **Callback Queue** when timer expires.
- **Event Loop** keep checking the **Callback Queue**, and see if it has any element to puts it into call stack.
  - **It is like a gate keeper.**
- Once **function cb()** is in Callback Queue, Event Loop pushes it to call stack to run.
- **Event Loop** has just one job to keep checking Callback Queue and if found something push it to call stack and delete from callback queue.

13



14

Promises!

- Time to talk about **Promises**!
- Why would you use them?
- How do they work "under the hood", and how can we write them in the most modern way?
- When writing JavaScript, we often have to deal with tasks that rely on other tasks!
  - Let's say that we want to get an image, compress it, apply a filter, and save it.
- How do you think the pseudocode will end up?

15

Promises!

- Time to talk about **Promises**!

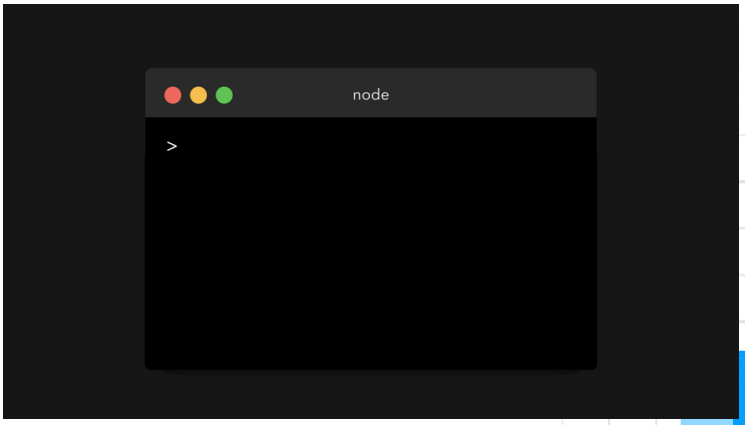
```
getImage('./image.png', (image, err) => {
  if (err) throw new Error(err)
  compressImage(image, (compressedImage, err) => {
    if (err) throw new Error(err)
    applyFilter(compressedImage, (filteredImage, err) => {
      if (err) throw new Error(err)
      saveImage(compressedImage, (res, err)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

- Too many nested callback functions that are dependent on the previous callback function.
- This is often referred to as a **callback hell**

16

Promise Syntax

- **ES6** introduced **Promises**.
- "*A promise is a placeholder for a value that can either **resolve** or **reject** at some time in the future*"



Mohamed Mukthar Ahmed

17

Promise Syntax

- **ES6** introduced **Promises**.
- "*A promise is a placeholder for a value that can either **resolve** or **reject** at some time in the future*“
- A Promise is an **object** that contains a status, ([[PromiseStatus]]) and a value ([[PromiseValue]]).
- Previous slide, you can see that the value of [[PromiseStatus]] is "**pending**", and the value of the promise is **undefined**.
- The values of these properties are important when working with promises.

Mohamed Mukthar Ahmed

18

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

PromiseStatus

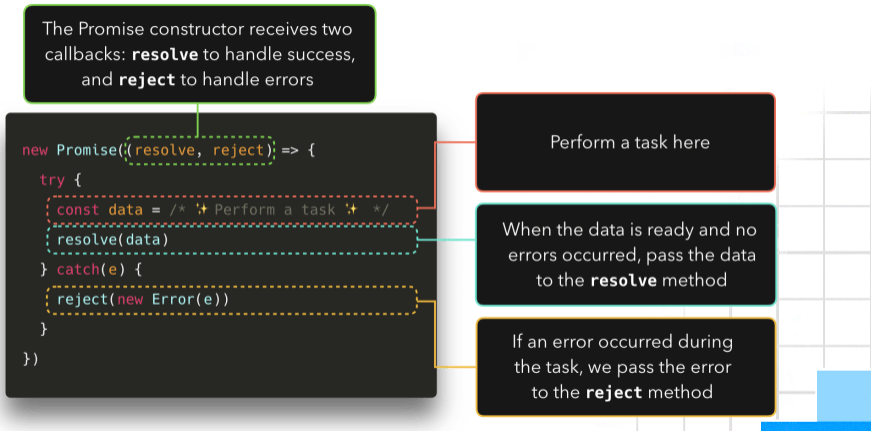
- The value of the **PromiseStatus**, the state, can be one of three values:
- ✅ **fulfilled**: The promise has been resolved. Everything went fine, no errors occurred within the promise 😸
- ✖ **rejected** : The promise has been rejected. Argh, something went wrong..
- ⏳ **pending**: The promise has neither resolved nor rejected (yet), the promise is still pending.

Mohamed Mukthar Ahmed

19

Promise Syntax

- **ES6** introduced **Promises**.

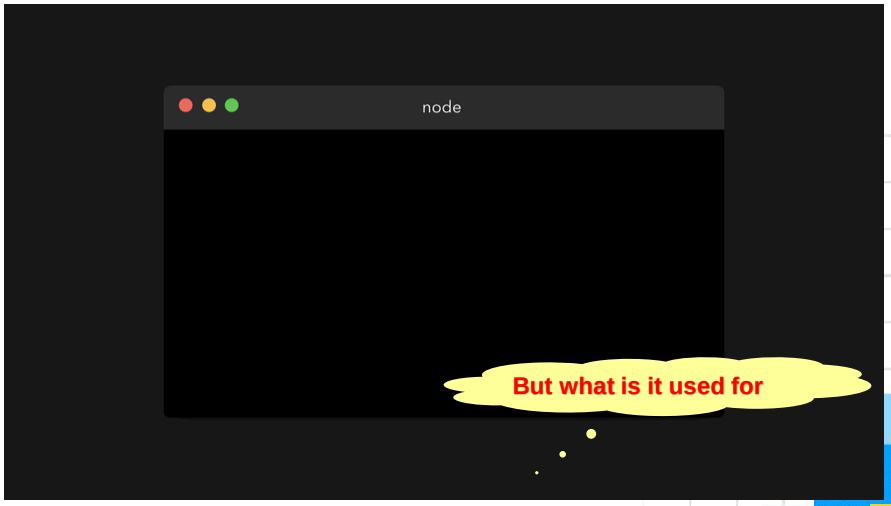


Mohamed Mukthar Ahmed

20

Promise Syntax

- Understanding Promise callback functions.



21

Promises

- Promises can help us **fix callback hell**!
- 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))
    }
  })
}
```


22

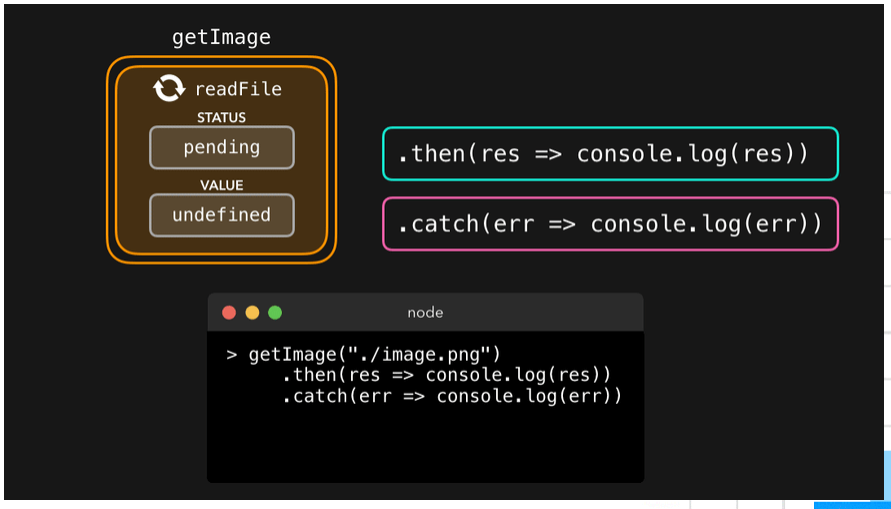
Promise - Methods

- 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.

```
getImage(file) {
    .then(image => console.log(image))
    .catch(error => console.log(error))
    .finally(() => console.log("All done!"))
```

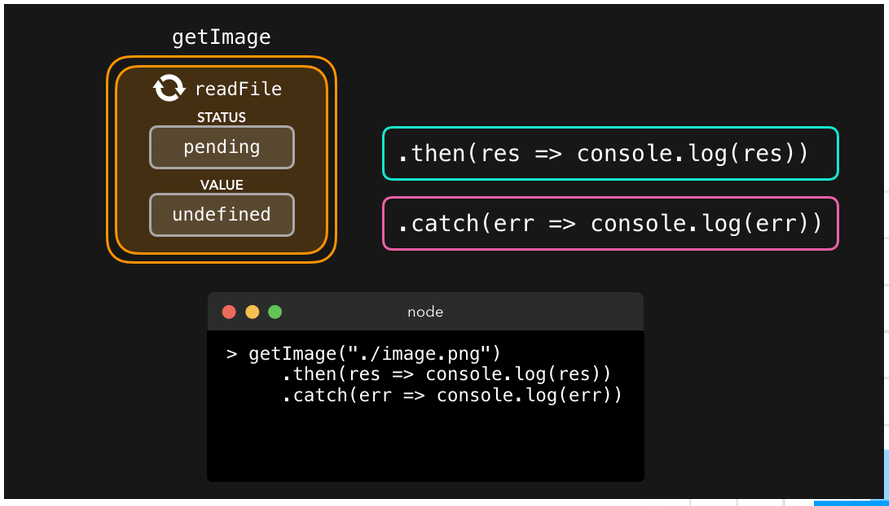

23

Promise - .then Method



24

Promise - .catch Method

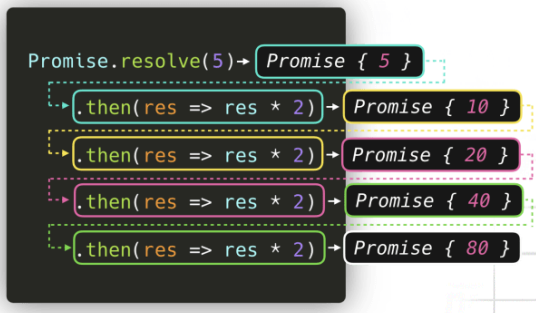


Mohamed Mukthar Ahmed

25

Chaining .then Methods

- 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!



Mohamed Mukthar Ahmed

26

Chaining .then Methods

- 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!

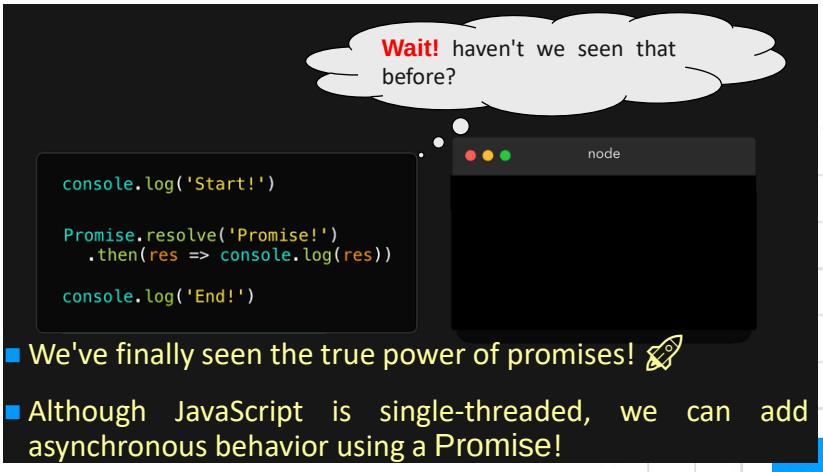
```
getImage('./image.png')
    .then(image => compressImage(image))
    .then(compressedImage => applyFilter(compressedImage))
    .then(filteredImage => saveImage(filteredImage))
    .then(res => console.log("Successfully saved image!"))
    .catch(err => throw new Error(err))
```

Mohamed Mukthar Ahmed

27

Microtasks and (Macro)tasks

- We have two type of tasks – microtasks and macrotaks



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

Mohamed Mukthar Ahmed

28

Microtasks and (Macro)tasks

- Within the **Event Loop**, there are actually two types of queues:
  - The (macro)task queue (or just called the task queue),
  - The microtask queue.
- The (macro)task queue is for (macro)tasks and the microtask queue is for microtasks.
- The most common are shown in the table below!

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

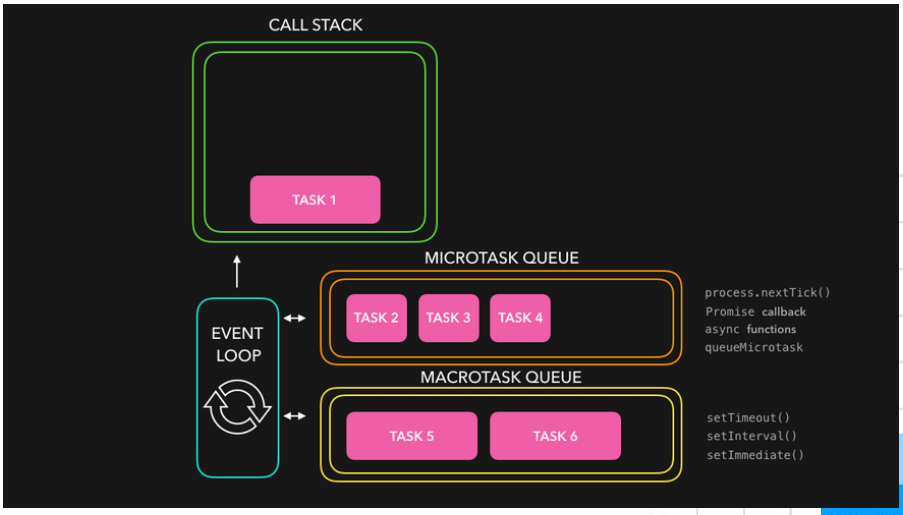
29

Priority to tasks

- The event loop gives a different **priority** to the tasks:
- All functions in that are currently in the **call stack** get executed. When they returned a value, they get popped off the stack.
- 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 😬)
- 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!

30

Microtask Queue



31

Microtask Queue

- All the callback functions that come through **promises** go in **Microtask Queue**.
- **Mutation Observer** : Keeps on checking whether there is mutation in DOM tree or not, and if there, then it executes some callback function.
- Callback functions that come through **promises** and **mutation observer** go inside **Microtask Queue**.
- All the rest goes inside **Callback Queue** aka. **Task Queue**.
- If the task in Microtask Queue keeps creating new tasks in the queue, element in Callback Queue never gets chance to run. This is called **starvation**

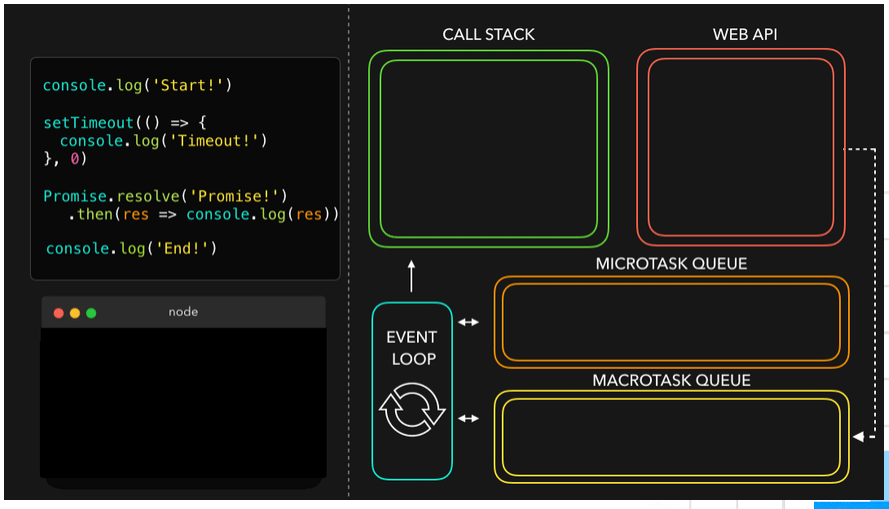
32

Code Example

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

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

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

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

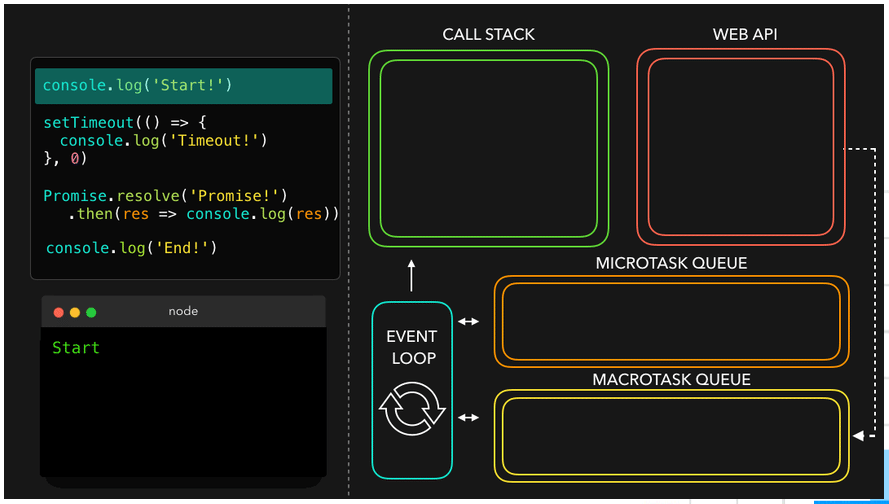

33

Microtask Queue



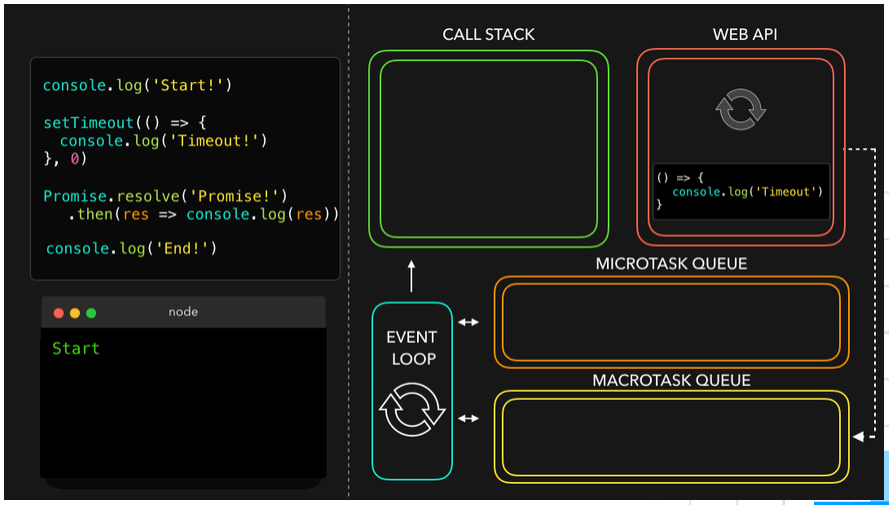
34

Microtask Queue



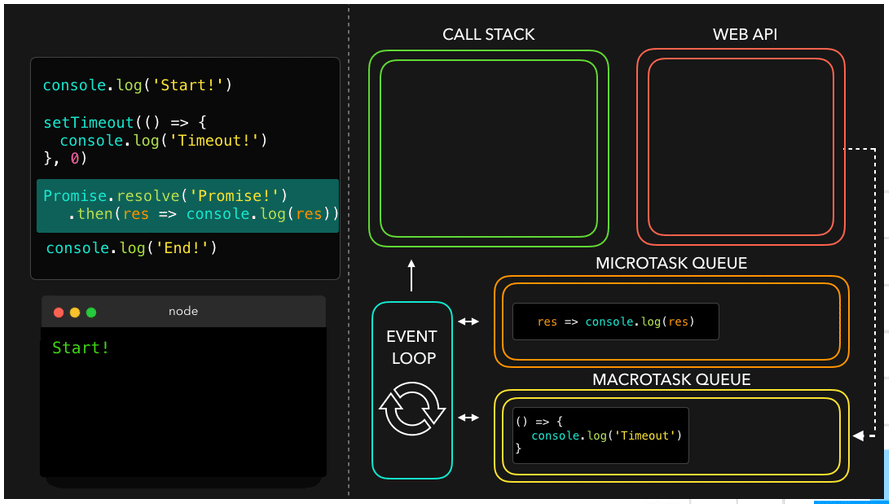
35

Microtask Queue



36

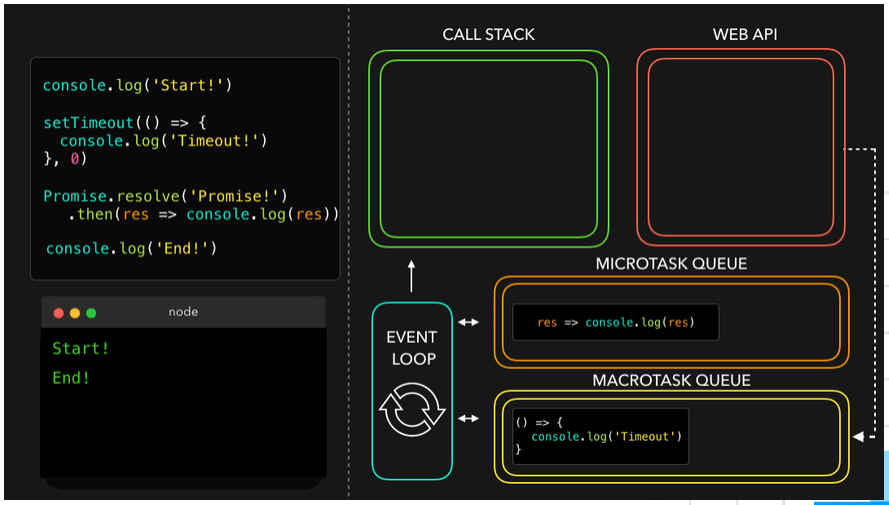
Microtask Queue



Mohamed Mukthar Ahmed

37

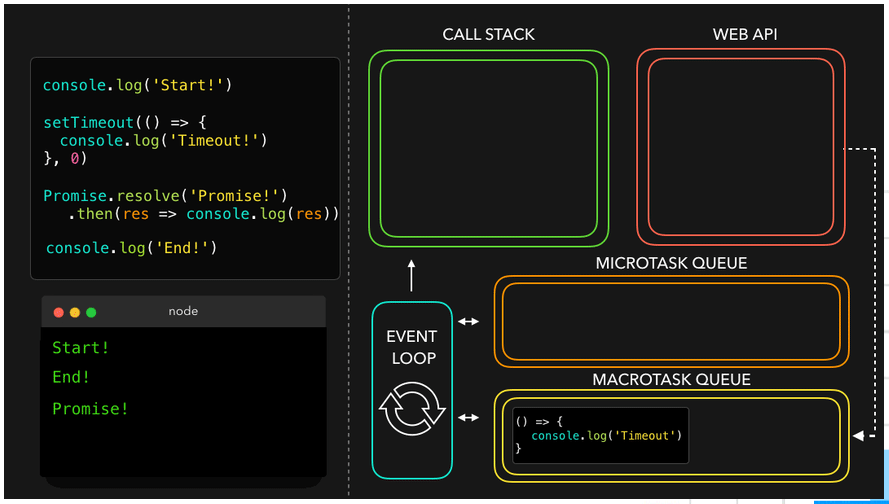
Microtask Queue



Mohamed Mukthar Ahmed

38

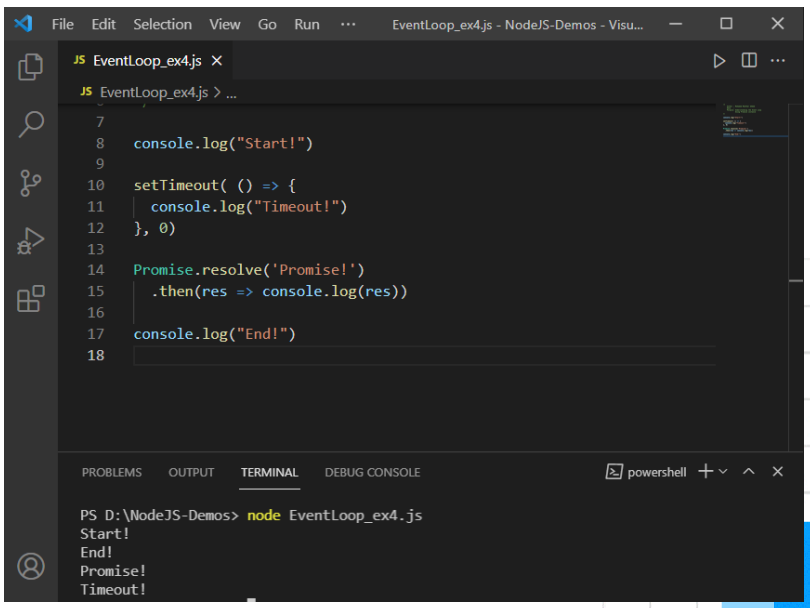
Microtask Queue



Mohamed Mukthar Ahmed

39

Microtask Queue

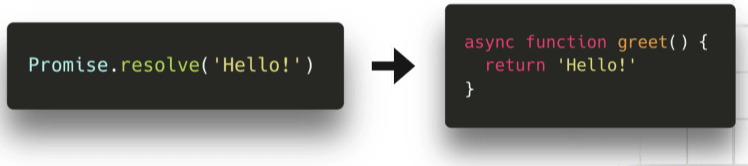


Mohamed Mukthar Ahmed

40

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 which implicitly return a promise.
- But.. how can we do that? 😮



41

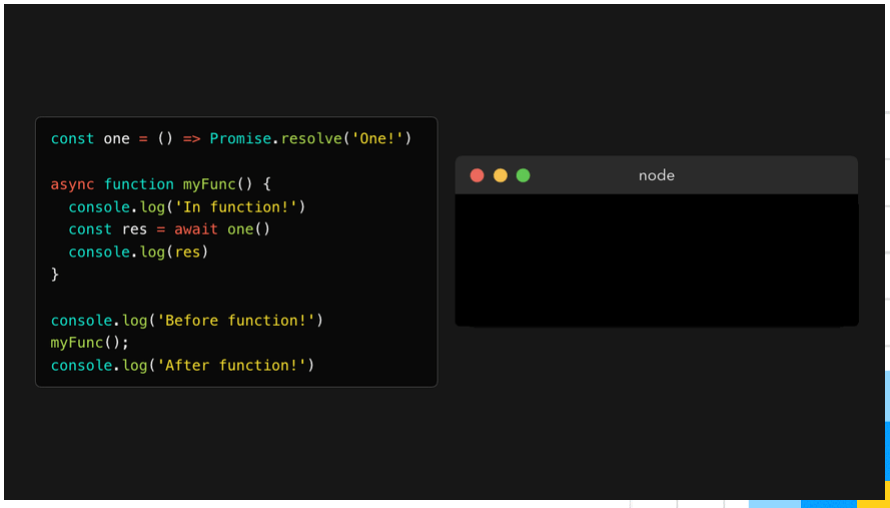
Async/Await

- 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 awaited value return a resolved promise.
- So, we can suspend an async function? Okay great but.. what does that even mean?

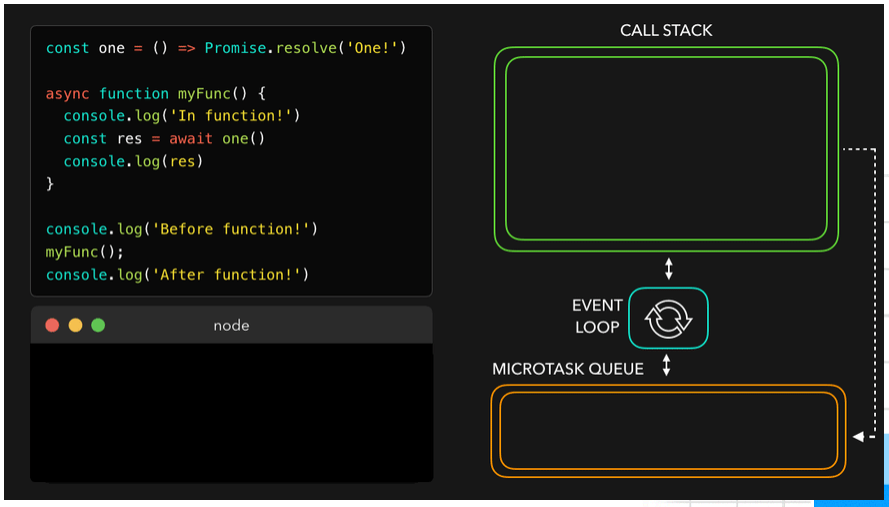
42

Async/Await

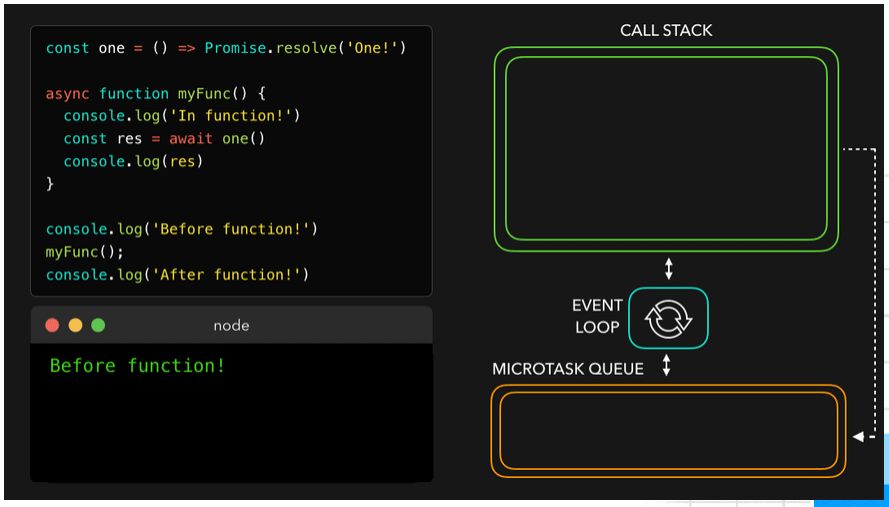
- Let's see what happens when we run the following block of code:



Async/Await



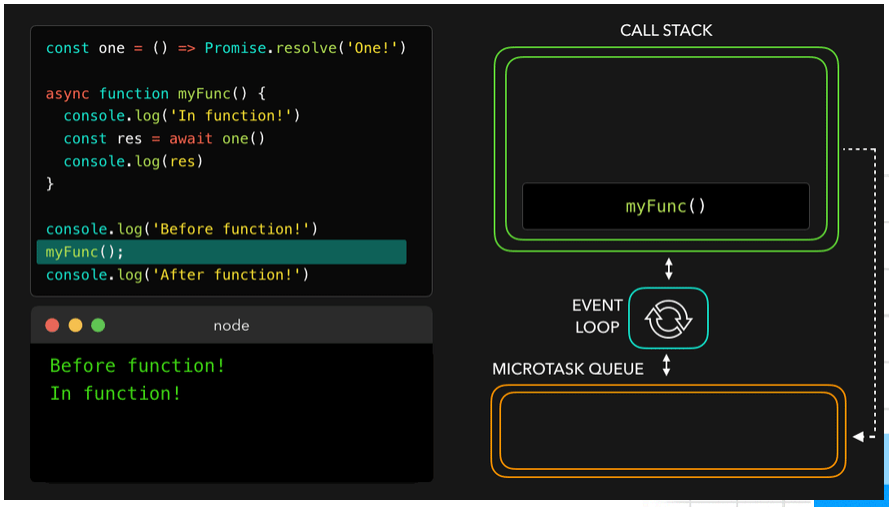
Async/Await



Mohamed Mukthar Ahmed

45

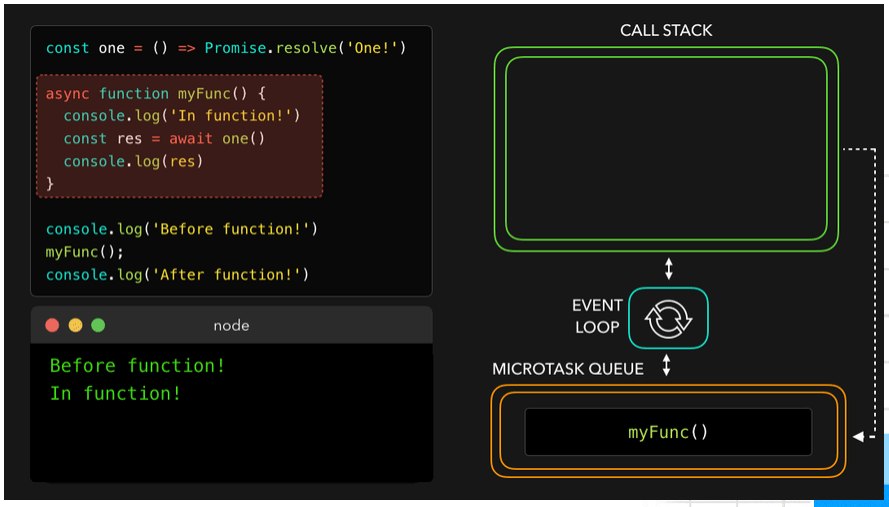
Async/Await



Mohamed Mukthar Ahmed

46

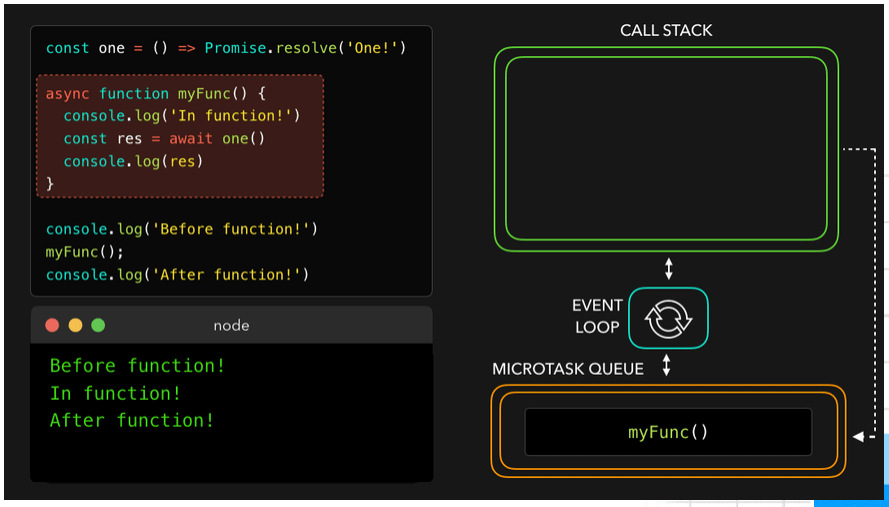
Async/Await



Mohamed Mukthar Ahmed

47

Async/Await



Mohamed Mukthar Ahmed

48

Async Function vs Promise

- 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**!

Quite a lot of information! 

I personally feel that it just takes experience to notice patterns and feel confident when working with asynchronous JavaScript.

Mohamed Mukthar Ahmed