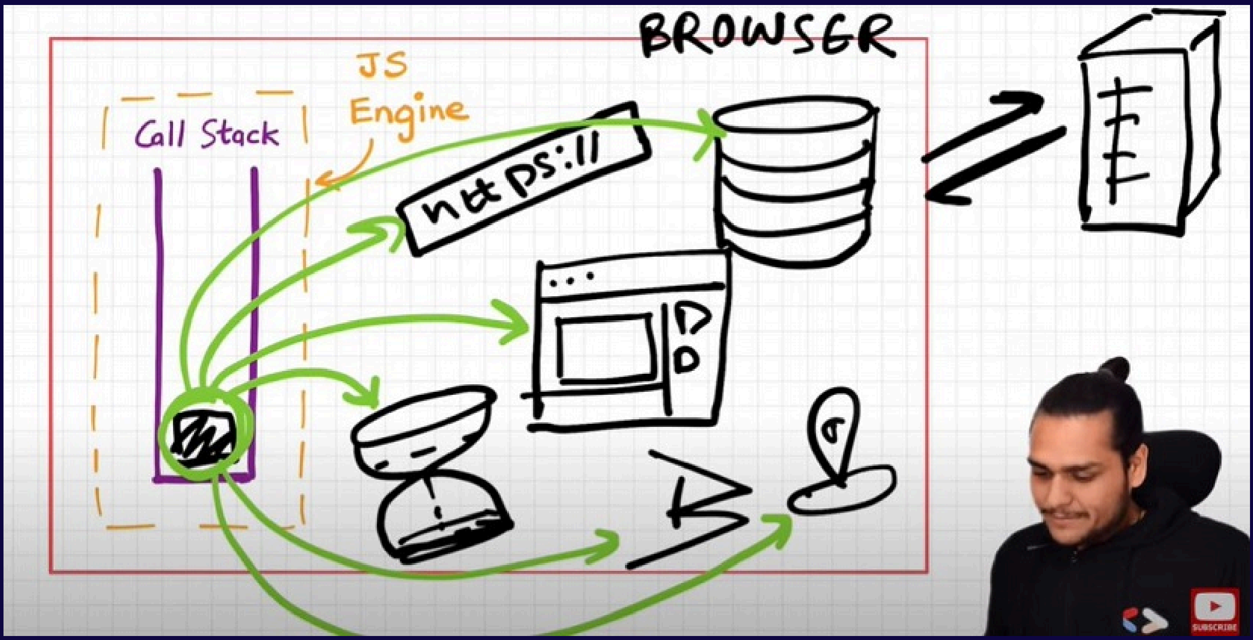
Episode 15 : Asynchronous JavaScript & EVENT LOOP from scratch

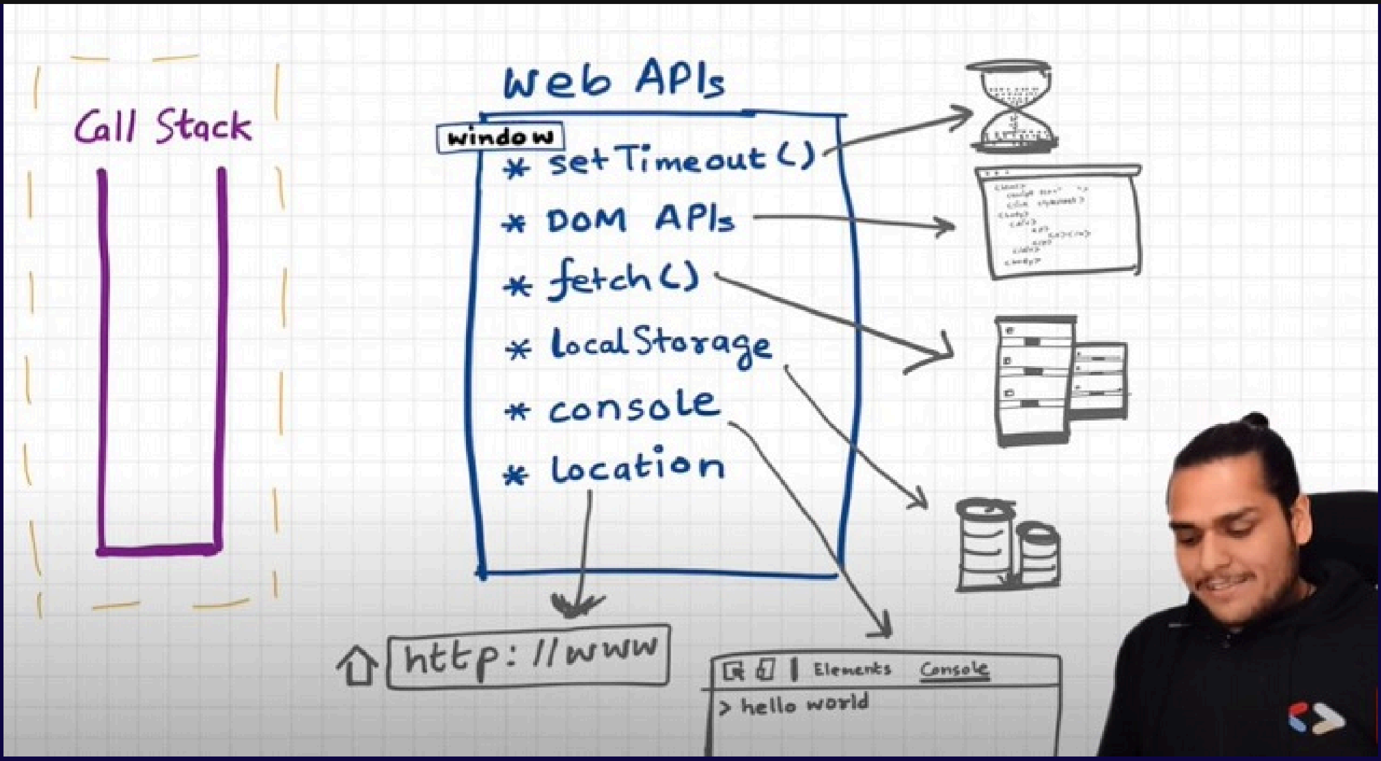
> Note: Call stack will exeecute any execeution context which enters it. Time, tide and JS waits for none. TLDR; Call stack has no timer.

- Browser has JS Engine which has Call Stack which has Global execution context, local execution context etc.
  - But browser has many other superpowers - Local storage space, Timer, place to enter URL, Bluetooth access, Geolocation access and so on.
  - Now JS needs some way to connect the callstack with all these superpowers. This is done using Web APIs.



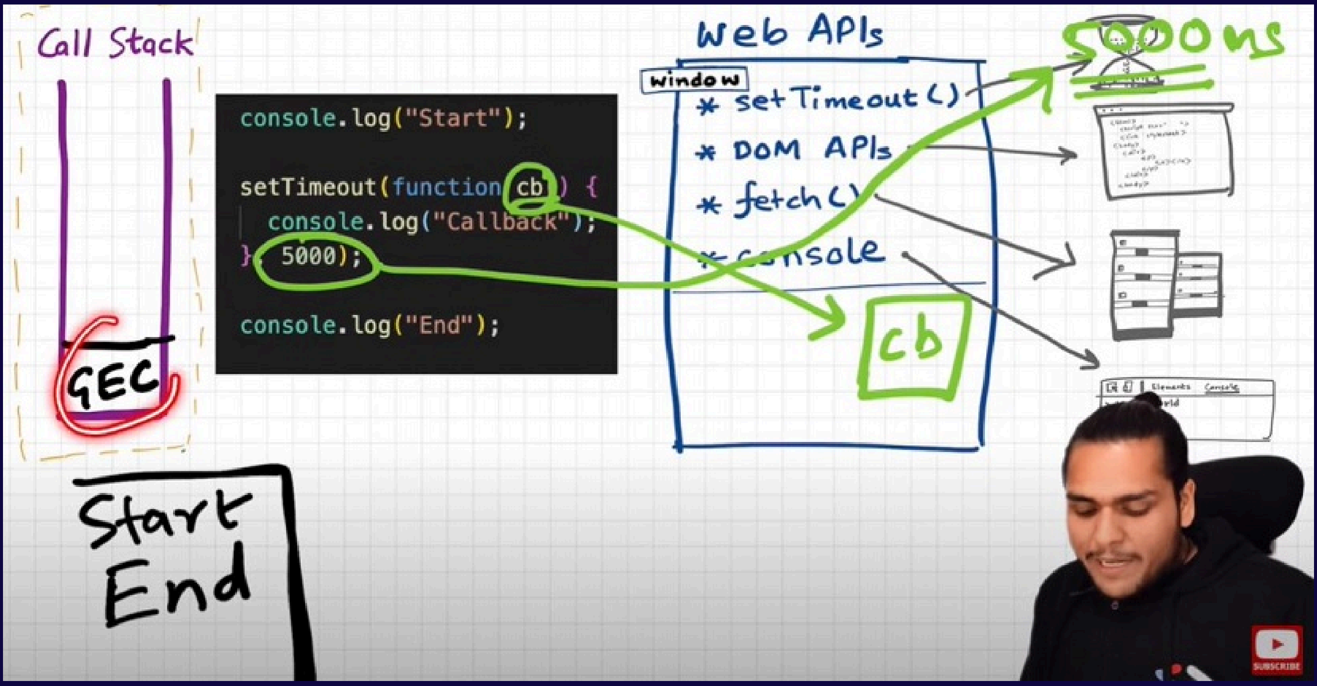
WebAPIs

None of the below are part of Javascript! These are extra superpowers that browser has. Browser gives access to JS callstack to use these powers.



- setTimeout(), DOM APIs, fetch(), localstorage, console (yes, even console.log is not JS!!), location and so many more.
    - setTimeout() : Timer function
    - DOM APIs : eg.Document.xxxx ; Used to access HTML DOM tree. (Document Object Manipulation)
    - fetch() : Used to make connection with external servers eg. Netflix servers etc.
- We get all these inside call stack through global object ie. window
    - Use window keyword like : window.setTimeout(), window.localstorage, window.console.log() to log something inside console.
    - As window is global obj, and all the above functions are present in global object, we don't explicity write window but it is implied.

- Let's undertand the below code image and its explaination:

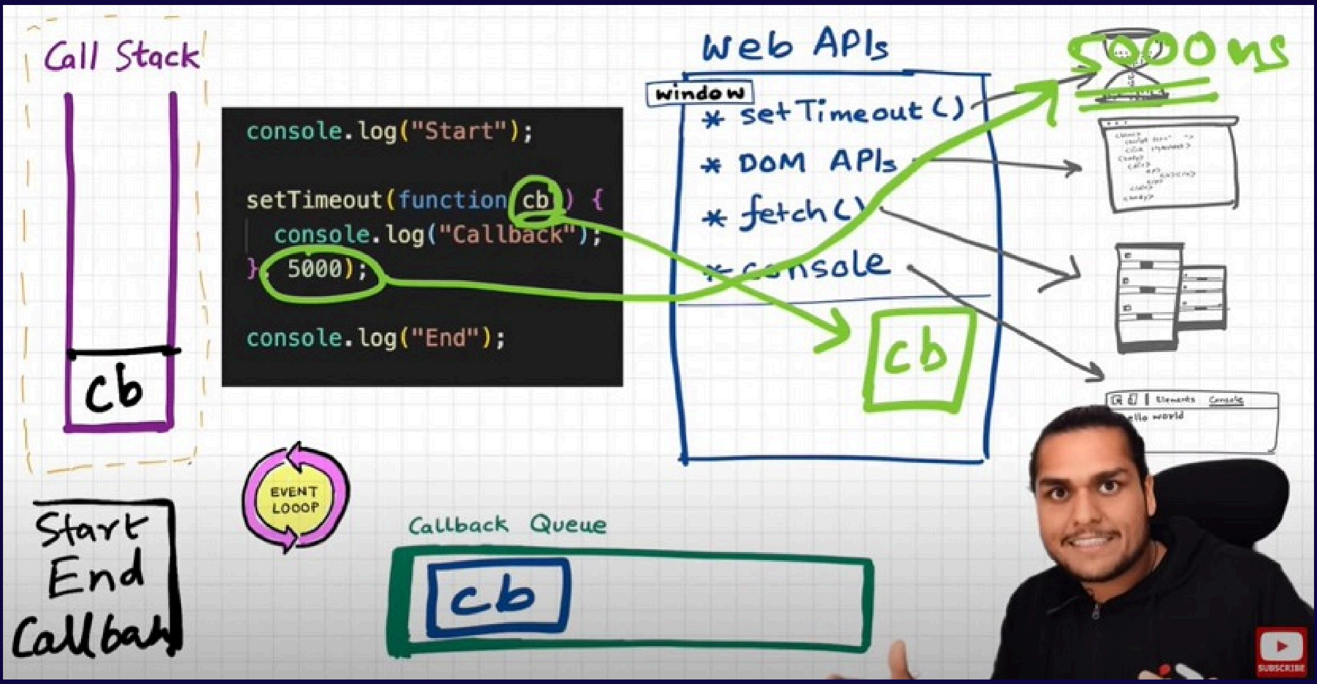


```
console.log("Start");

setTimeout(function cb() {
  console.log("Callback");
}, 5000);

console.log("End");
```

  -
    ```
    console.log('start');
    setTimeout(function cb() {
      console.log('timer');
    }, 5000);
    console.log('end');
    // start end timer
    ```
  - First a GEC is created and put inside call stack.
  - console.log("Start"); // this calls the console web api (through window) which in turn actually modifies values in console.
  - setTimeout(function cb() { //this calls the setTimeout web api which gives access to timer feature. It stores the callback cb() and starts timer. console.log("Callback");}, 5000);
  - console.log("End"); // calls console api and logs in console window. After this GEC pops from call stack.
  - 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**

Event Loops and Callback Queue

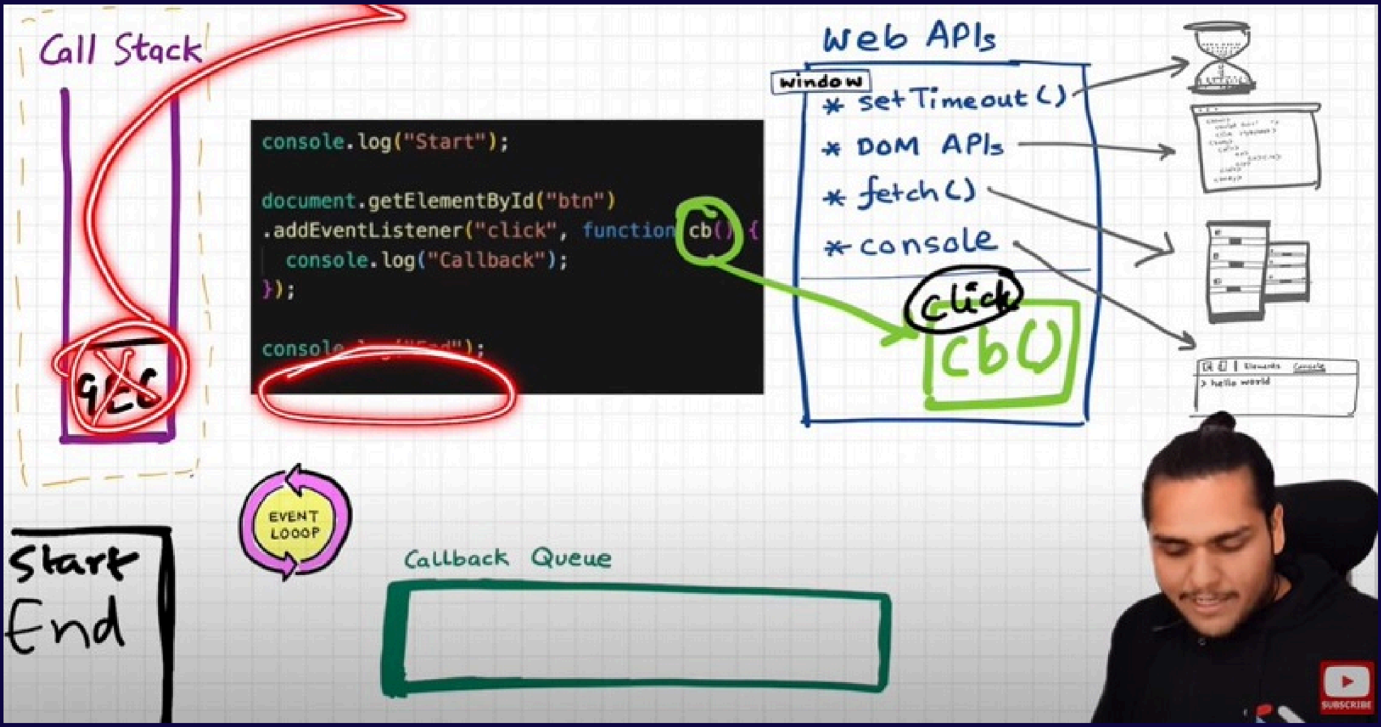
Q: How after 5 secs timer is console?

- cb() cannot simply directly go to callstack to be exeecuted. 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 cb() is in callback queue, eventloop pushes it to callstack to run. Console API is used and log printed
- 

```
console.log("Start");

setTimeout(function cb() {
  console.log("Callback");
}, 5000);

console.log("End");
```

Q: Another example to understand Eventloop & Callback Queue.

See the below Image and code and try to understand the reason:



```
console.log("Start");

document.getElementById("btn")
.addEventListener("click", function cb() {
  console.log("Callback");
});

console.log("End");
```

Explaination?

-
  ```
  console.log('Start');
  document.getElementById('btn').addEventListener('click', function cb() {
    // cb() registered inside webapi environment and event(click) attached to it. i.e. REGISTERING CALLBACK AND ATTACHING EVENT TO IT.
    console.log('Callback');
  });
  console.log('End'); // calls console api and logs in console window. After this GEC get removed from call stack.
  // In above code, even after console prints "Start" and "End" and pops GEC out, the eventListener stays in webapi env(with hope that user may click it some day) until explicitly removed, or the browser is closed.
  ```
- Eventloop has just one job to keep checking callback queue and if found something push it to call stack and delete from callback queue.

Q: Need of callback queue?

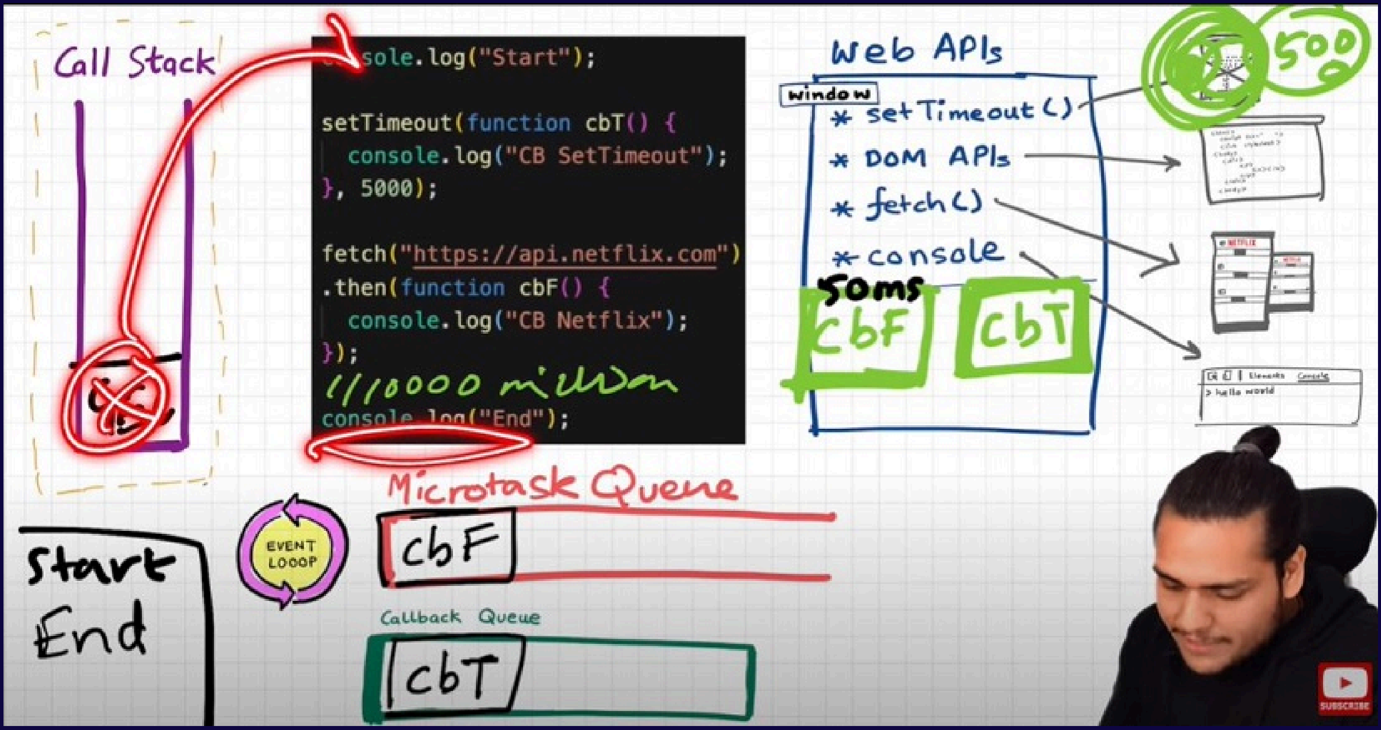
Ans: Suppose user clciks button x6 times. So 6 cb() are put inside callback queue. Event loop sees if call stack is empty/has space and whether callback queue is not empty(6 elements here). Elements of callback queue popped off, put in callstack, executed and then popped off from call stack.

Behaviour of fetch (**Microtask Queue?**)

Let's observe the code below and try to understand

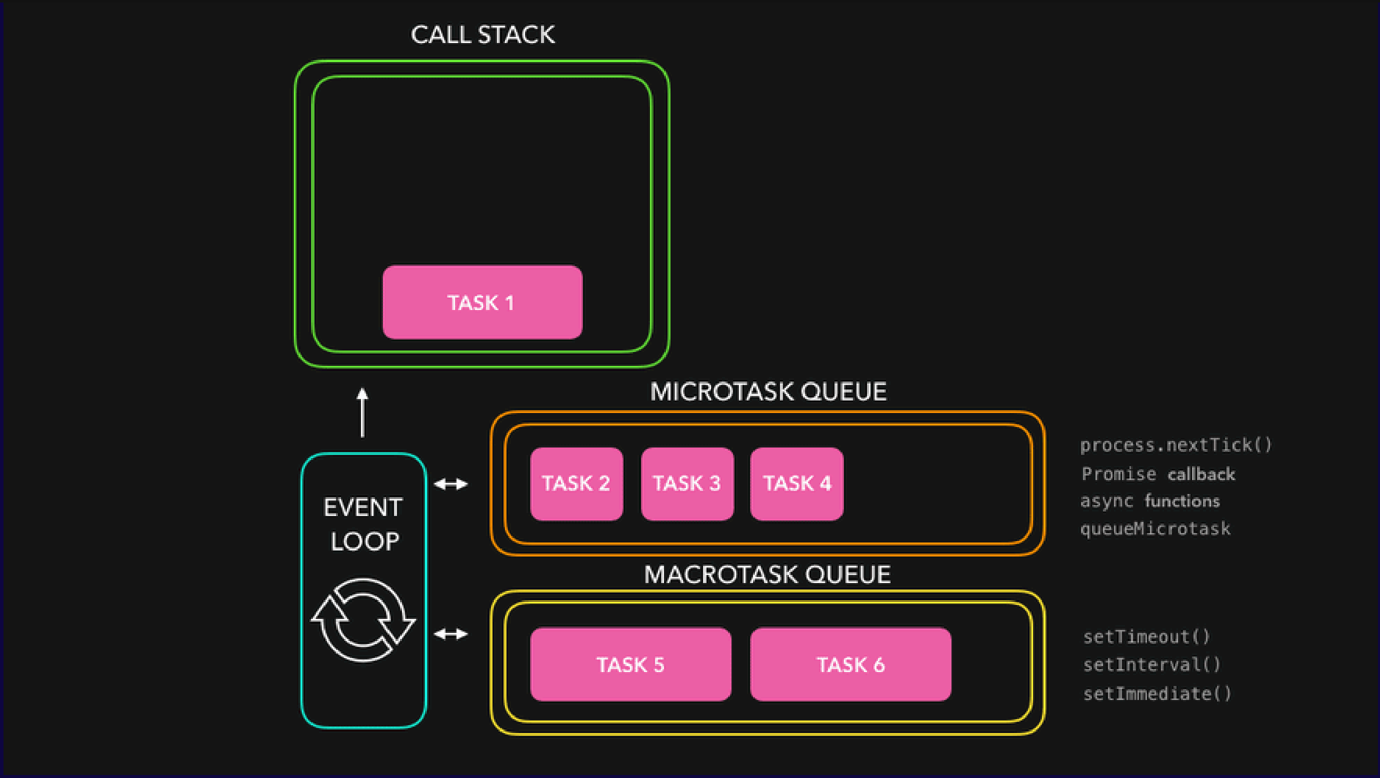
```
console.log("Start"); // this calls the console web api (through window) which in turn actually modifies values in console.
setTimeout(function cbT() {
  console.log("CB Timeout");
}, 5000);
fetch("https://api.netflix.com").then(function cbF() {
    console.log("CB Netflix");
}); // take 2 seconds to bring response
// millions lines of code
console.log("End");

Code Explaination:
* Same steps for everything before fetch() in above code.
* fetch registers cbF into webapi environment along with existing cbT.
* cbT is waiting for 5000ms to end so that it can be put inside callback queue. cbF is waiting for data to be returned from Netflix servers gonna take 2 seconds.
* After this millions of lines of code is running, by the time millions line of
```

```
code will execute, 5 seconds has finished and now the timer has expired and
response from Netflix server is ready.
* Data back from cbF ready to be executed gets stored into something called a
Microtask Queue.
* Also after expiration of timer, cbT is ready to execute in Callback Queue.
* Microtask Queue is exactly same as Callback Queue, but it has higher priority.
Functions in Microtask Queue are executed earlier than Callback Queue.
* In console, first Start and End are printed in console. First cbF goes in
callstack and "CB Netflix" is printed. cbF popped from callstack. Next cbT is
removed from callback Queue, put in Call Stack, "CB Timeout" is printed, and cbT
removed from callstack.
* See below Image for more understanding
```



```
console.log("Start");

setTimeout(function cbT() {
  console.log("CB SetTimeout");
}, 5000);

fetch("https://api.netflix.com")
.then(function cbF() {
  console.log("CB Netflix");
});
//10000 million
console.log("End");
```

Microtask Priority Visualization



What enters the 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 execeutes 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 be run. This is called **starvation**

Some Important Questions

1. **When does the event loop actually start ? -** Event loop, as the name suggests, is a single-thread, loop that is *almost infinite*. It's always running and doing its job.

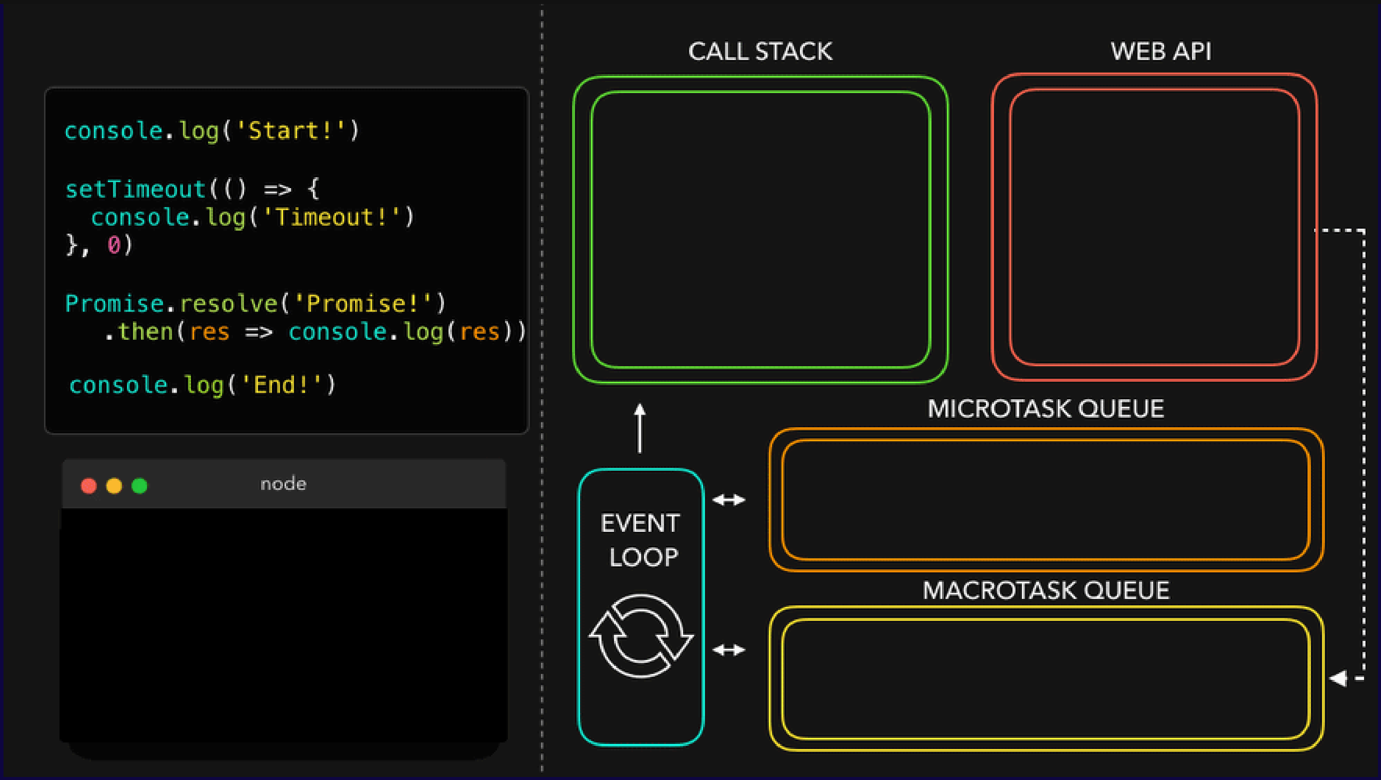
2. **Are only asynchronous web api callbacks are registered in web api environment? -** YES, the synchronous callback functions like what we pass inside map, filter and reduce aren't registered in the Web API environment. It's just those async callback functions which go through all this.

3. **Does the web API environment stores only the callback function and pushes the same callback to queue/microtask queue? -** Yes, the callback functions are stored, and a reference is scheduled in the queues. Moreover, in the case of event listeners(for example

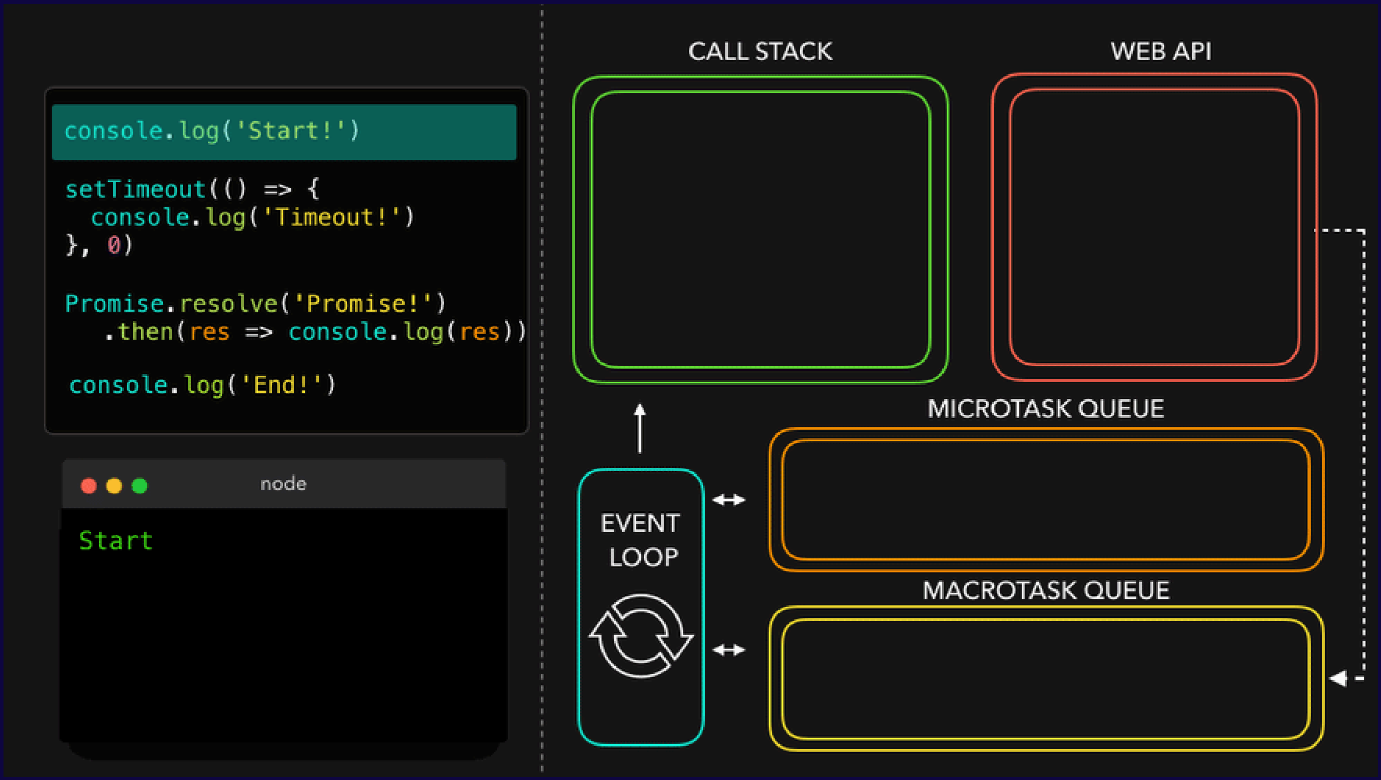
click handlers), the original callbacks stay in the web API environment forever, that's why it's adviced to explicitly remove the listeners when not in use so that the garbage collector does its job.

4. **How does it matter if we delay for setTimeout would be 0ms. Then callback will move to queue without any wait ? -** No, there are trust issues with setTimeout() 😅. The callback function needs to wait until the Call Stack is empty. So the 0 ms callback might have to wait for 100ms also if the stack is busy.

Observation of Eventloop, Callback Queue & Microtask Queue [**GiF**]

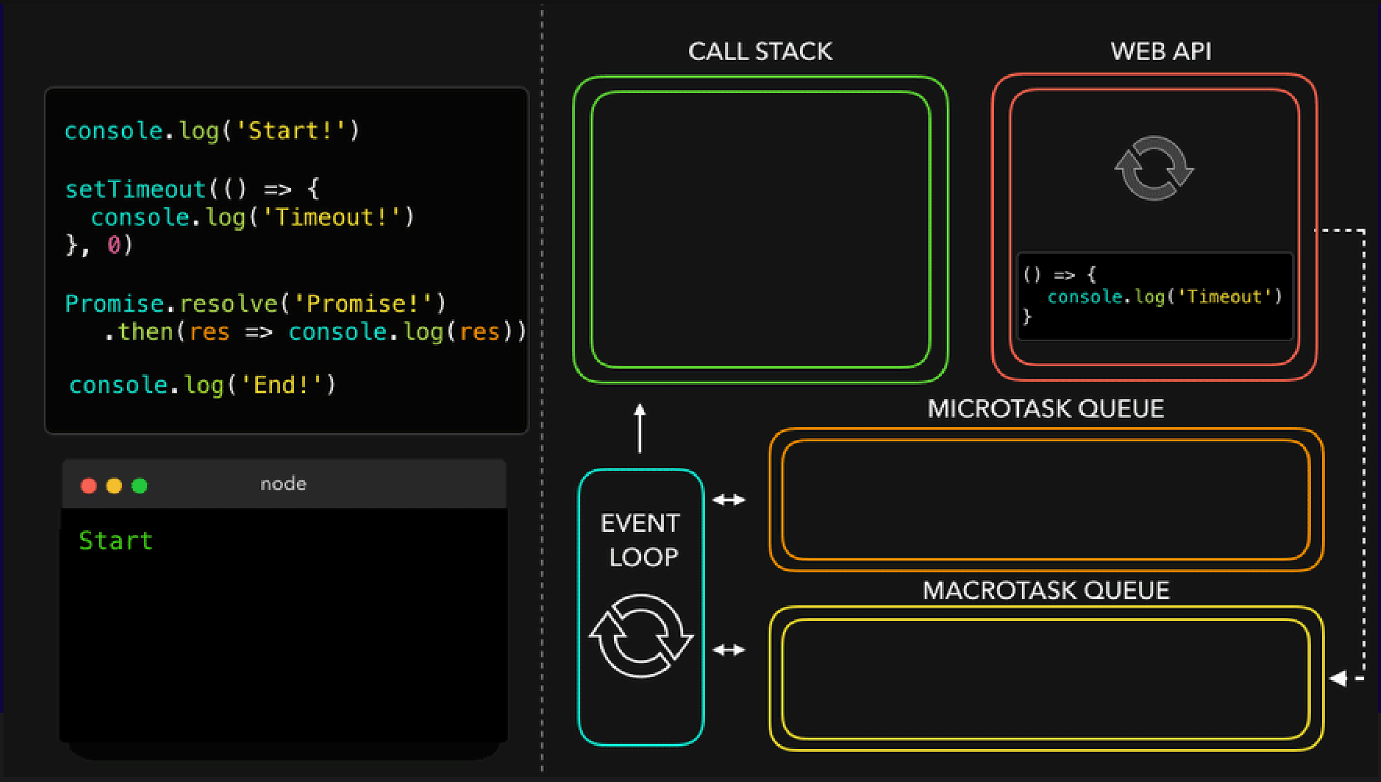


```
console.log('Start!')

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

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

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



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

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

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

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

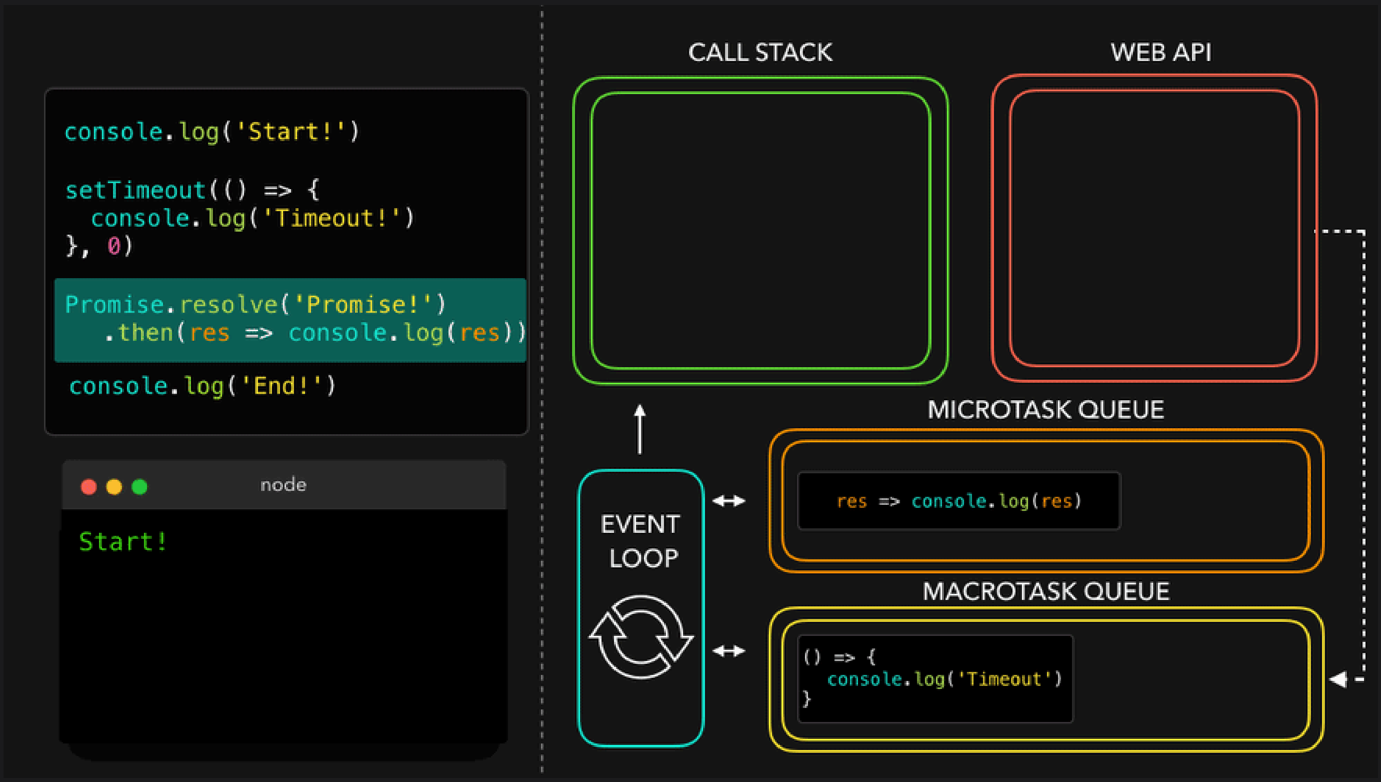
```
Start
```



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

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

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

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

```
Start
```

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

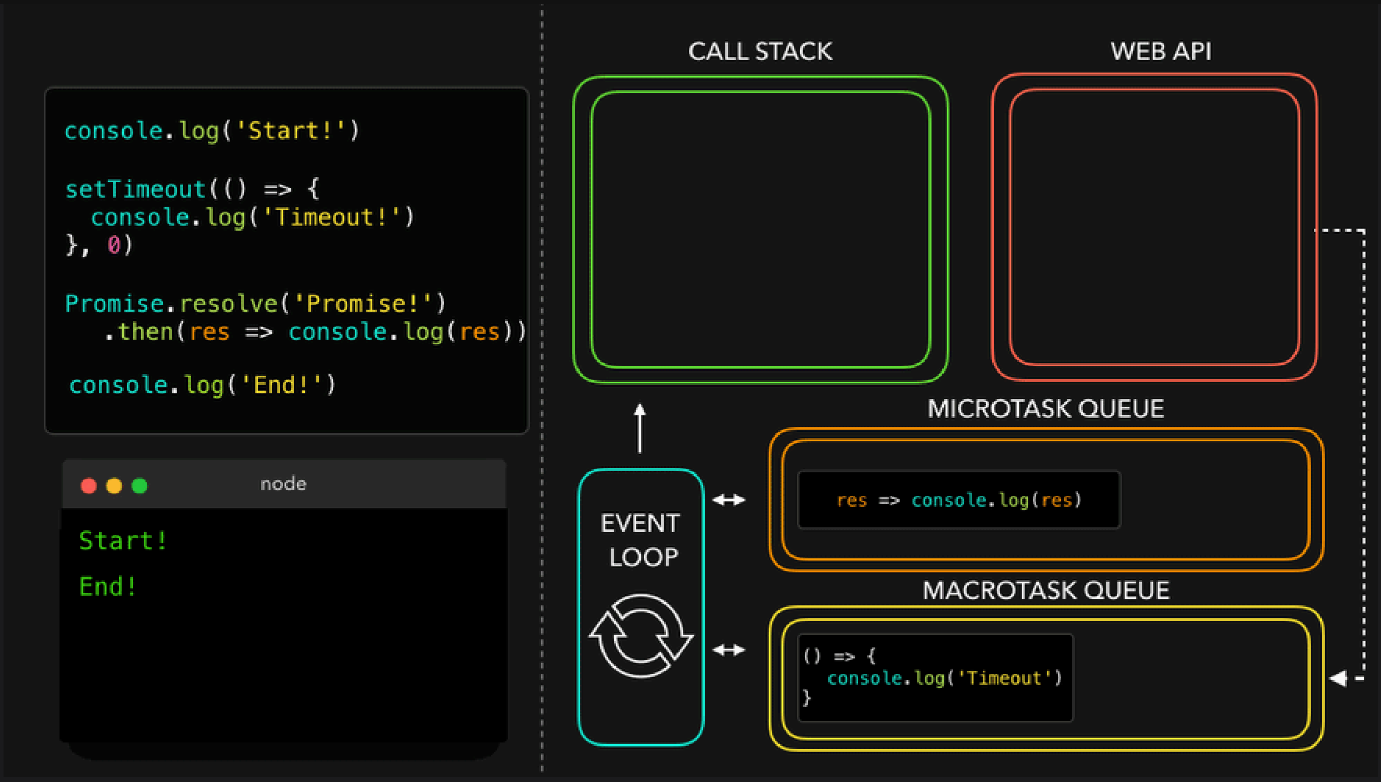


```
console.log('Start!')

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

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

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

```
Start!
```

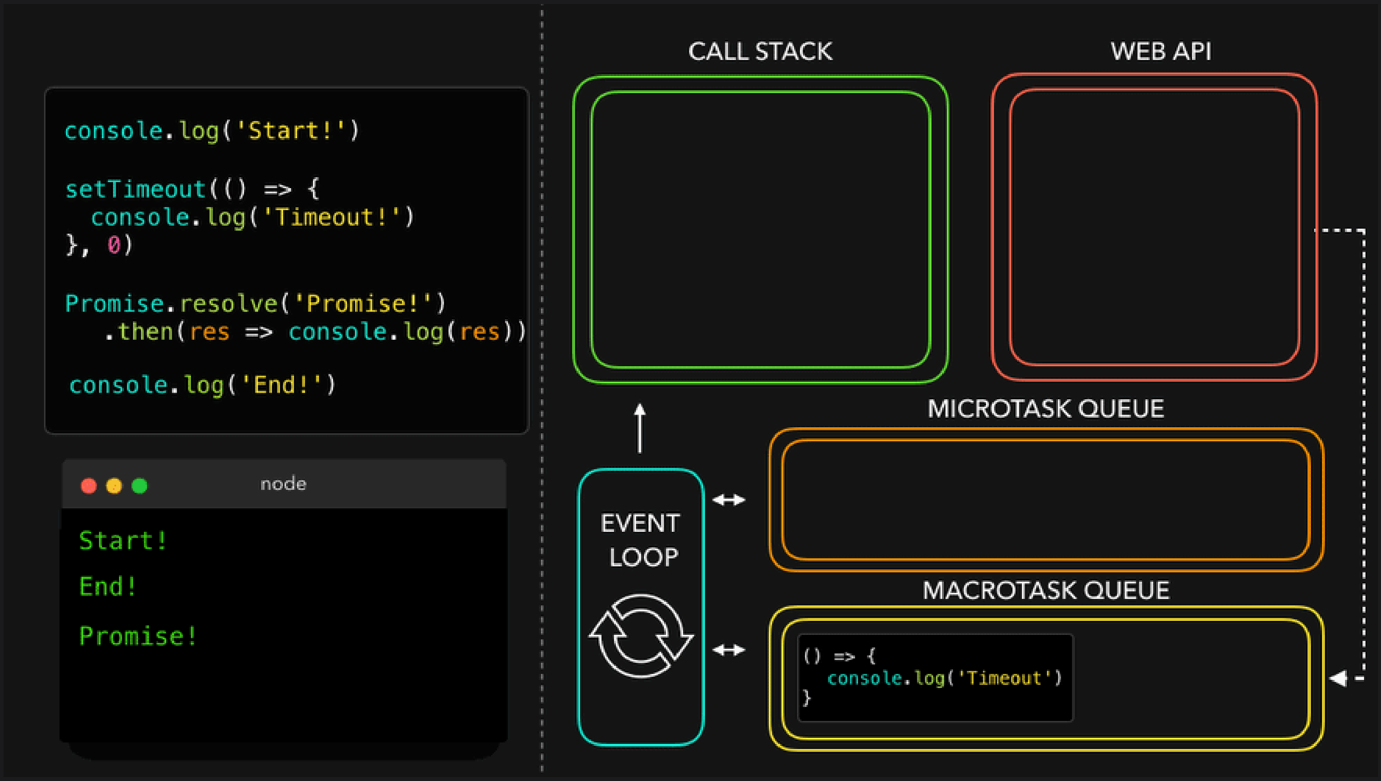
```
res => console.log(res)
```

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



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

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

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

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

```
Start!
End!
```



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

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

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

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

```
Start!
End!
Promise!
```

Watch Live On Youtube below:

Edit this page