Synchronous and Asynchronous in JavaScript

Synchronous JavaScript

As the name suggests synchronous means to be in a sequence, i.e. every statement of the code gets executed one by one. So, basically a statement has to wait for the earlier statement to get executed.

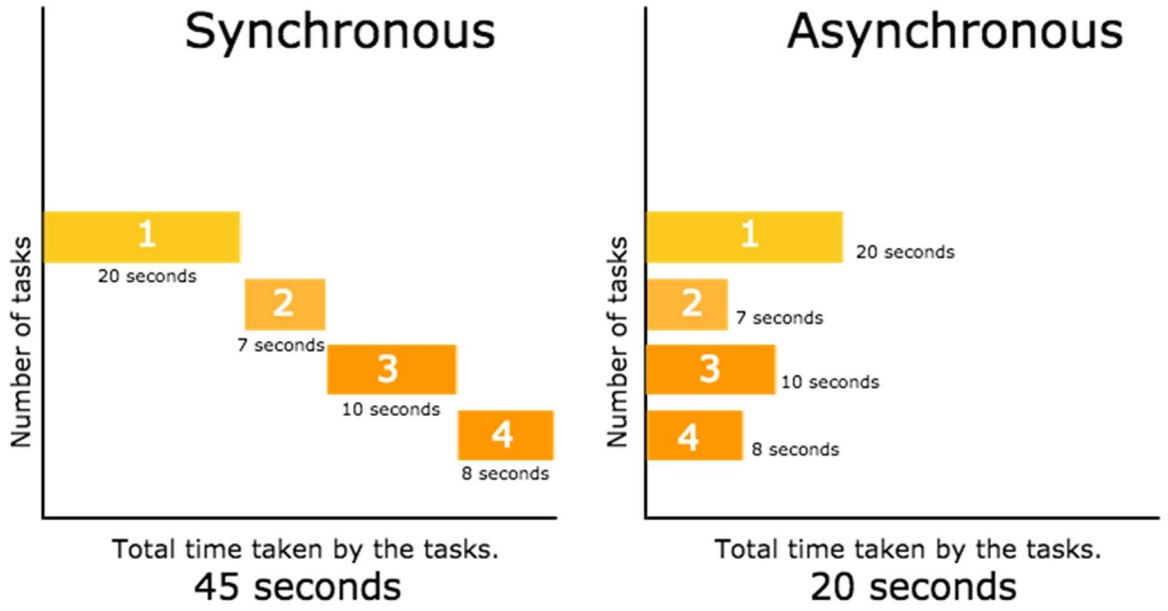
Example:

```
console.log('line 1');

console.log('line 2');

console.log('line 3');

console.log('line 4');
```

Asynchronous JavaScript

Asynchronous code allows the program to be executed immediately where the synchronous code will block further execution of the remaining code until it finishes the current one. Let us see the example how Asynchronous JavaScript runs.

```
setTimeout(() => {

    console.log("line 1");

console.log("line 2");

console.log("line 3");

console.log("line 4"); }, 2000);
```



Callback Functions

- A callback function is a function that is passed *as an argument* to another function, to be "called back" at a later time.
- A function that accepts other functions as arguments is called a **higher-order function**, which contains the logic for *when* the callback function gets executed.
- It's the combination of these two that allow us to extend our functionality.
- Eg: setTimeout(()=>)

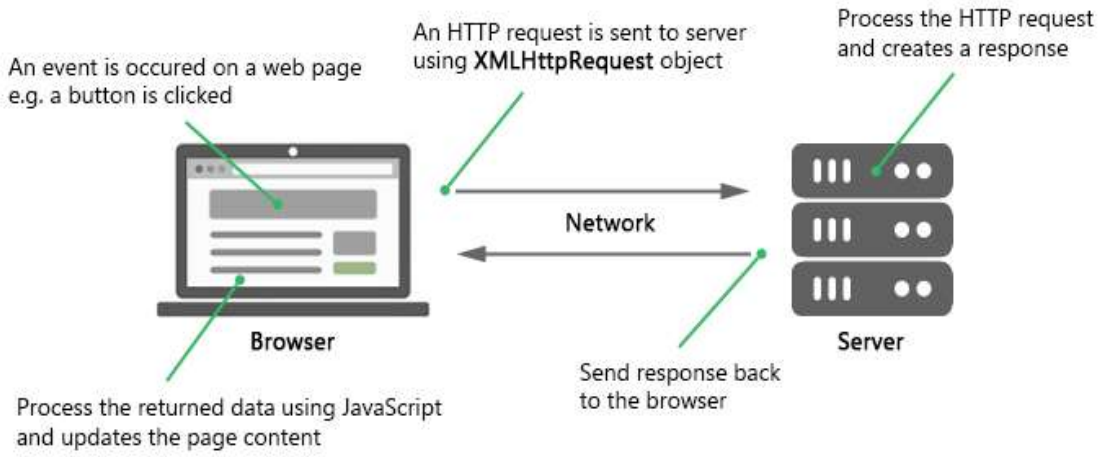
AJAX

- Ajax stands for **A**synchronous **J**avascript **A**nd **X**ml.
- AJAX is not a programming language.
- AJAX just uses a combination of: A browser built-in XMLHttpRequest object (to request data from a web server) JavaScript and HTML DOM (to display or use the data)
- Ajax is just a means of loading data from the server and selectively updating parts of a web page without reloading the whole page.
- Basically, what Ajax does is make use of the browser's built-in XMLHttpRequest (XHR) object to send and receive information to and from a web server asynchronously, in the background, without blocking the page or interfering with the user's experience.
- Ajax has become so popular that you hardly find an application that doesn't use Ajax to some extent.
- The example of some large-scale Ajax-driven online applications are: Gmail, Google Maps, Google Docs, YouTube, Facebook, Flickr, and so many other applications.

Understanding how Ajax works

- To perform Ajax communication JavaScript uses a special object built into the browser—an XMLHttpRequest (XHR) object—to make HTTP requests to the server and receive data in response.
- All modern browsers (Chrome, Firefox, IE7+, Safari, Opera) support the XMLHttpRequest object.

The following illustrations demonstrate how Ajax communication works:

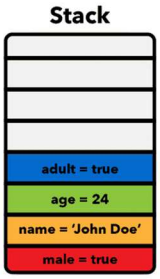


The memory heap and stack

- JavaScript engines have two places where they can store data: The memory heap and stack.
- Heaps and stacks are two data structures that the engine uses for different purposes.

Stack: Static memory allocation

- A stack is a data structure that JavaScript uses to store static data.
- Static data is data where the engine knows the size at compile time.
- In JavaScript, this includes primitive values (strings, numbers, booleans, undefined, and null) and references, which point to objects and functions.



- A stack is a data structure that JavaScript uses to store static data.
- Static data is data where the engine knows the size at compile time.
- It will allocate a fixed amount of memory for each value.
- The process of allocating memory right before execution is known as static memory allocation.
- Because the engine allocates a fixed amount of memory for these values, there is a limit to how large primitive values can be.
- The limits of these values and the entire stack vary depending on the browser.

Memory Heap:

- Memory heap is the place where the memory is allocated for the variables and functions etc.
- Dynamic memory allocation.
- The engine doesn't allocate a fixed amount of memory for these objects. Instead, more space will be allocated as needed.

  Eg:

```
const person = {
  name: 'John',
  age: 24,

    };
```

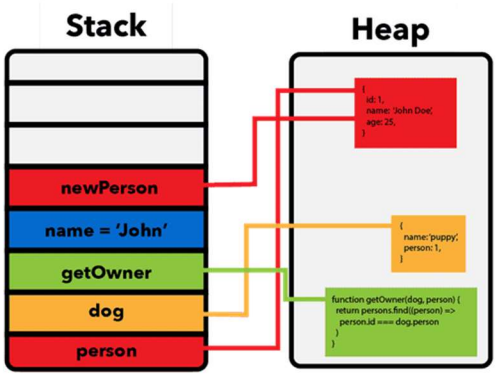
```
const hobbies = ['hiking', 'reading'];
```

Arrays are objects as well, which is why they are stored in the heap.

```
let name = 'John'; // allocates memory for a string

const age = 24; // allocates memory for a number


name = 'John Doe'; // allocates memory for a new string

const firstName = name.slice(0,4); // allocates memory for a new string
```

Primitive values are immutable, which means that instead of changing the original value, JavaScript creates a new one

```
const person = {
  id: 1,
  name: 'John',
  age: 25,
}

const dog = {
  name: 'puppy',
  personId: 1,
}

function getOwner(dog, persons) {
  return persons.find((person) =>
    person.id === dog.person
  )
}

const name = 'John';

const newPerson = person;
```



person and newPerson both point to the same object.

Person creates a new object in the heap and a reference to it in the stack.

Stack Overflow:

- When the function runs inside and inside, the call stack will be filled and overflows.
- When the stack overflows, Maximum call stack size exceeded error will be thrown.
- The below function runs inside and inside and the stack will be overflowed.

Ex:

```
function inception() {

   inception();

}

 inception();
```

Garbage Collection:

- JavaScript automatically cleans out the memory allocated to the data after its execution.
- For example when the memory allocated for the variable created inside the function, it will be cleared after the function execution automatically.
- It will manages the memory leaks.
- It will automatically controls the memory heap.
- It follows Mark and Sweep algorithm to handle Garbage collection.

Memory Leaks:

- Memory leaks occurred when the data is overloaded (Infinite data into an array), it breaks the browser.

Ex:

```
let array = [];

for(let i=1; i>0; i++) {

   array.push(i);

}
```

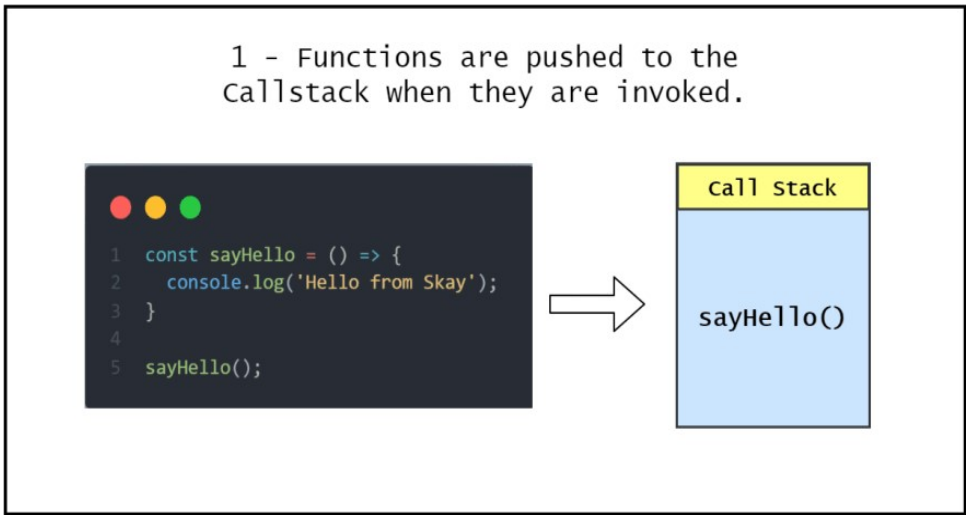
Call Stack

- The Call Stack is a part of the JavaScript Engine and it is simply a stack in which you can add an item and the item added first is processed last.
- In other words, it follows the FILO - 'First In Last Out' principle.
- A Call Stack acts as a placeholder or a holding area for all the JavaScript functions that have been fired for execution.

Let us look at the below code sample and understand how the function 'sayHello' gets added to the Call Stack.

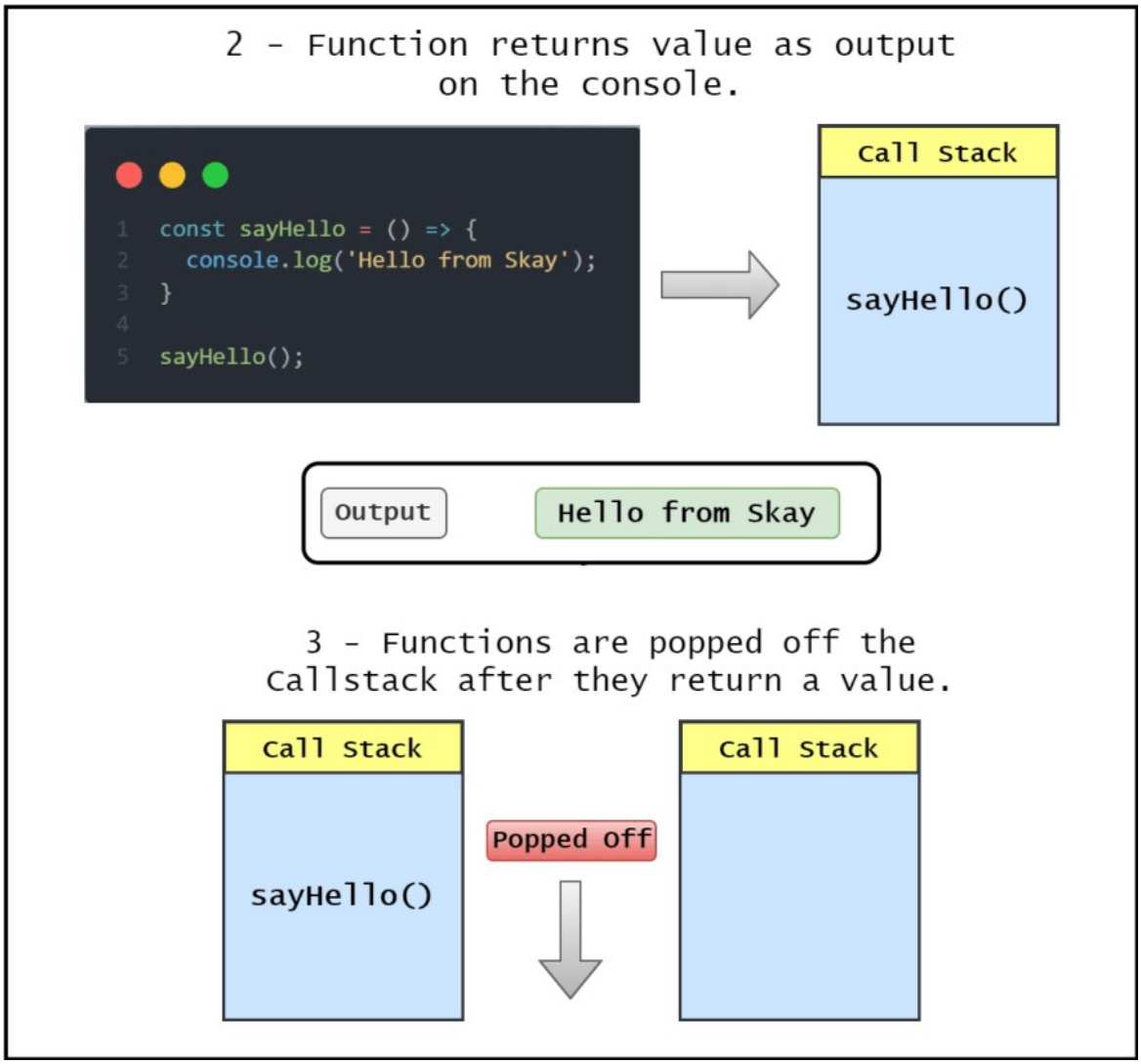
```
const sayHello = () => {
  console.log('Hello from Skay');
}

sayHello();
```

Step 1 - sayHello Function gets added to the Call Stack



Step 2 & 3 - Function returns a value & gets popped from the Call Stack

Once the function returns a value, the output 'Hello from Skay' gets displayed on the console. As soon as the function returns, almost immediately, the function is also popped off from the call stack.



Web API

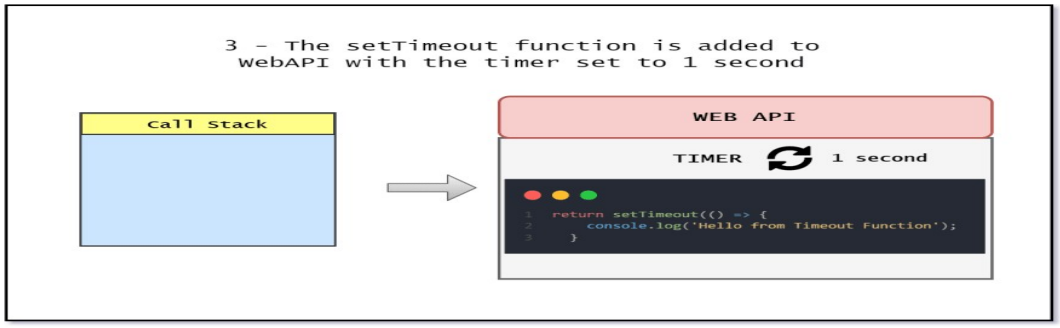
- Web APIs are built into your web browser and they are able to process data from the browser to do complex useful things.
- The main thing to understand is that, they are not a part of the JavaScript language, but, they are built on top of the core JS language and provide additional capabilities such as Geolocation, LocalStorage, etc.

```
const sayHello = () => {
  console.log('Hello from Skay');
}

const timeoutFunction = () => {
  return setTimeout(() => {
    console.log('Hello from Timeout Function');
  }, 1000)
}

sayHello();
timeoutFunction();
```

When the function 'timeoutFunction' is executed, it returns the 'setTimeout' function and is popped off the stack.

Since the 'setTimeout' function is part of the Web API, it gets moved to the Web API

Along with that, the timer function in Web API is set to 1 second (1000 ms) based on the argument we had passed to the 'setTimeout' function.

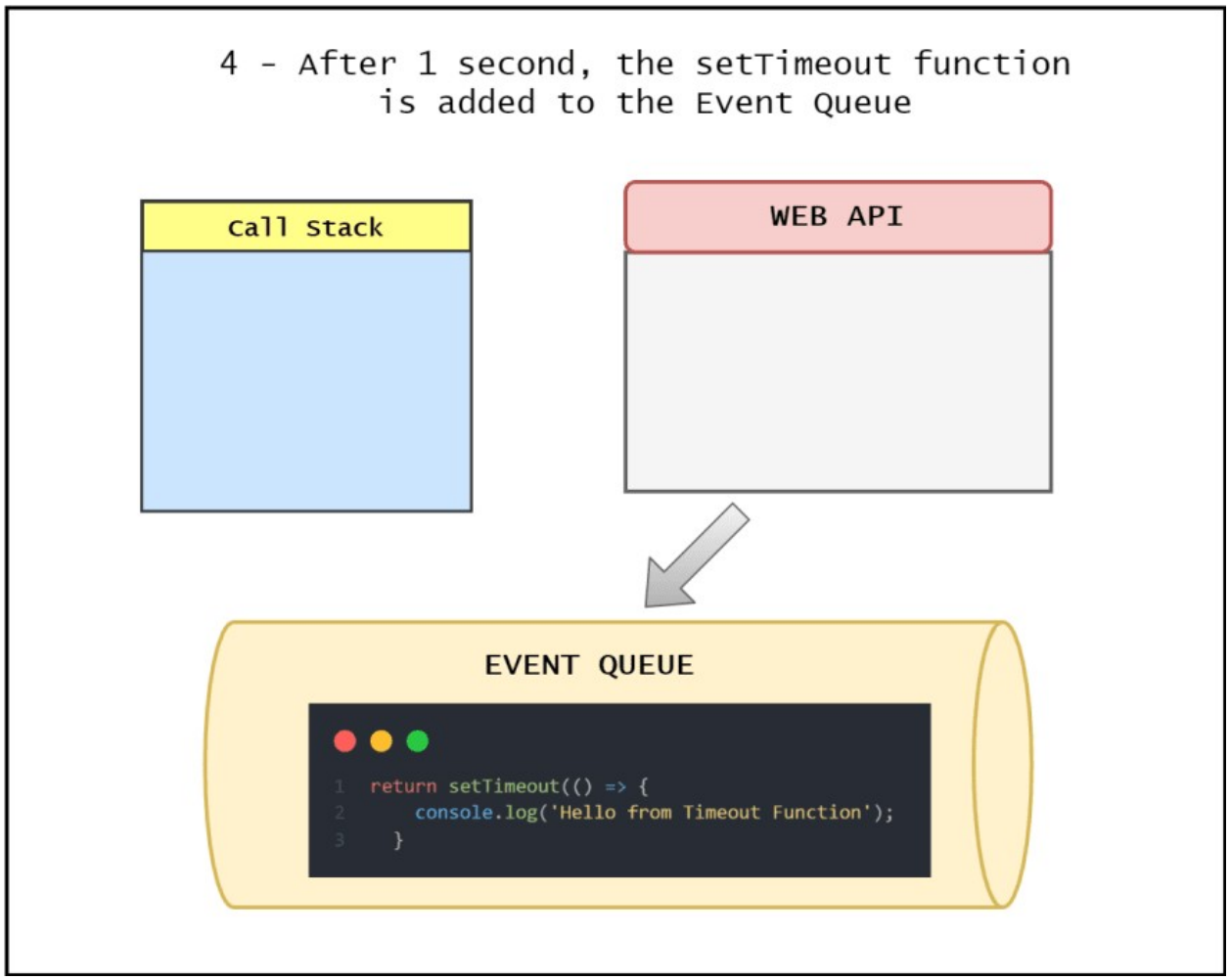


It adds it to the Event Queue.

Event Queue

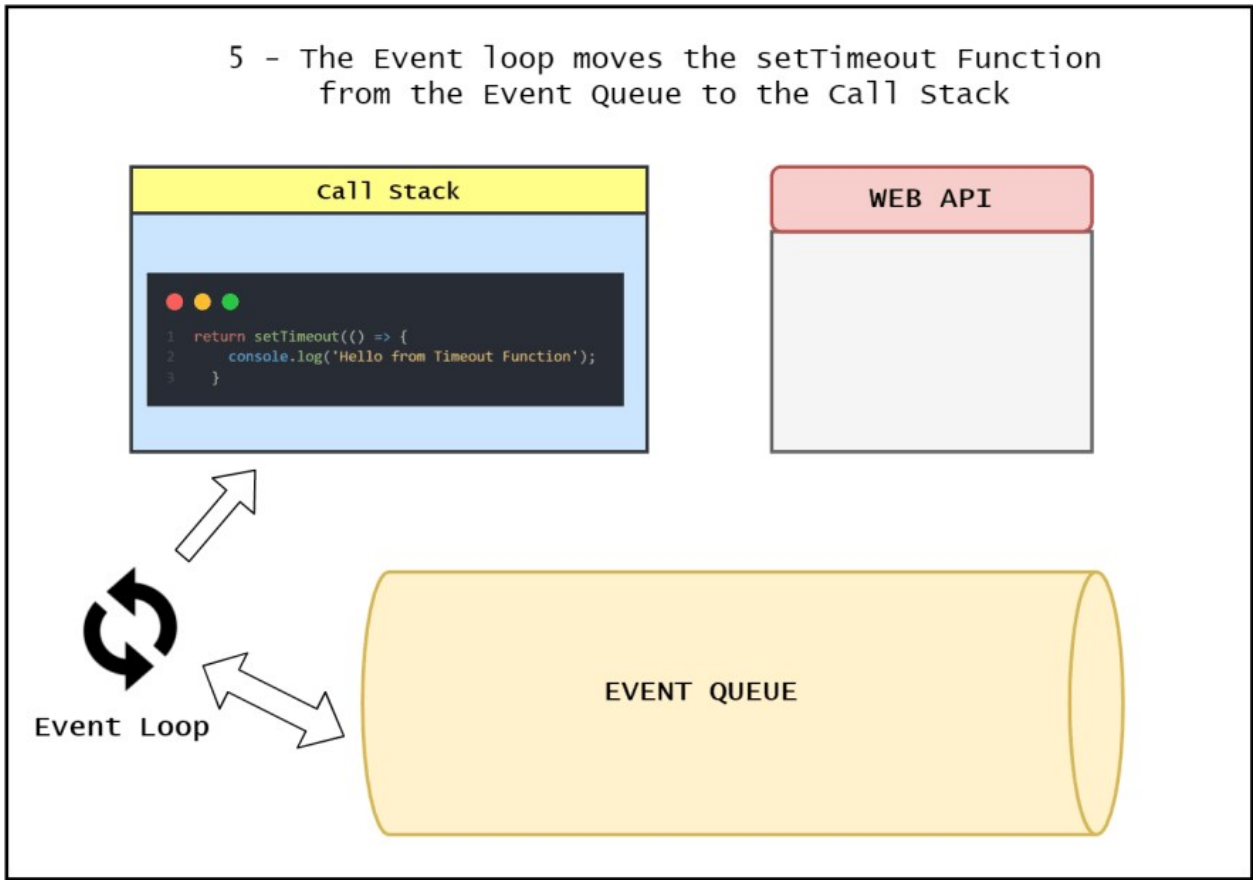
- Event Queue is a data structure similar to Stack, which holds the data temporarily and the important thing to note is that the data added first is processed first.
- In other words, it follows the FIFO -> 'First in First out' principle.

So after one second, the 'setTimeout' function from Web API is added to the Queue



Event Loop

- The function of Event Loop can simply be explained as connecting the Event Queue to the Call Stack.



Event Loop does the following:

- Checks if the Call Stack is empty, i.e., if all the functions have completed their execution and they have been popped off the Call Stack.
- Once the Call Stack is empty, it moves the first item from the Event Queue to the Call Stack.

When the setTimeout function gets added to the Call Stack, the function simply returns the output 'Hello from Timeout Function' on the console and is popped out of the Stack.