Asynchronous JavaScript: The Event Loop

Open intro.html for the code | Images are taken from: JS Course by Jonas Schmedtmann

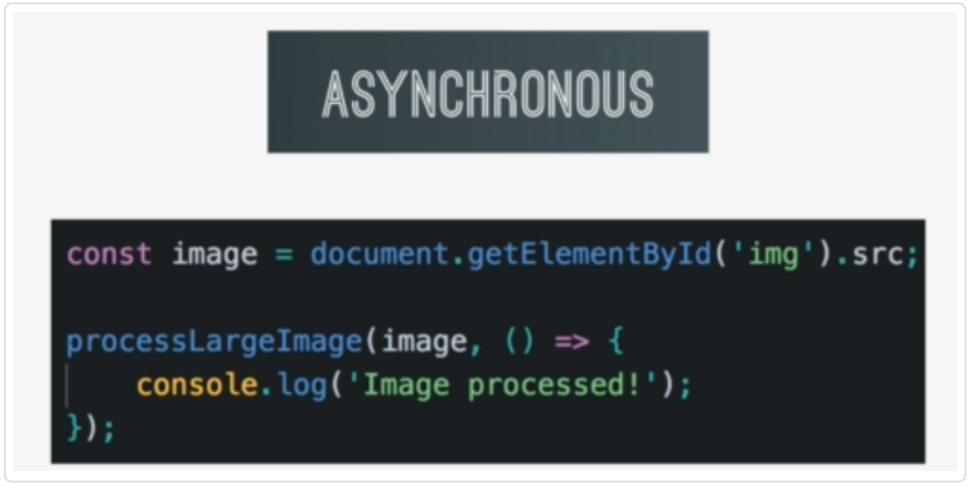
All the code we have been writing up until this point is Synchronous code, which simply means that one statement is processed after the other, line by line, in a single `thread`, in the JS Engine. The following is an example of Synchronous JavaScript code:

`first()` is called and then "Hey There!" is logged on to the console. After that, `second()` and it logs "How are you doing?" onto the console. `second()` finishes its execution and the control returns to `first()'s` Execution Context, where finally, "The end" is logged onto the console. Therefore here, the execution is done one instruction after the other in a synchronous way.

Now, an example of Asynchronous JavaScript code is given as follows:

Here, again the `first()` gets called and "Hey There!" gets logged onto the console and after that the `second()` function gets called. In `second()`'s execution context, `setTimeout()` is called with a callback function and 2000ms timeout. The `second()`'s execution context pops out of the execution stack and the control now returns to the `first()`'s execution context where "The end" is logged onto the console. After 2000ms have elasped, the callback inside the `setTimeout()` function is called, and that callback function logs "Async Hey There!" onto the console.

Another example of Asynchronous JavaScript code is given as follows:



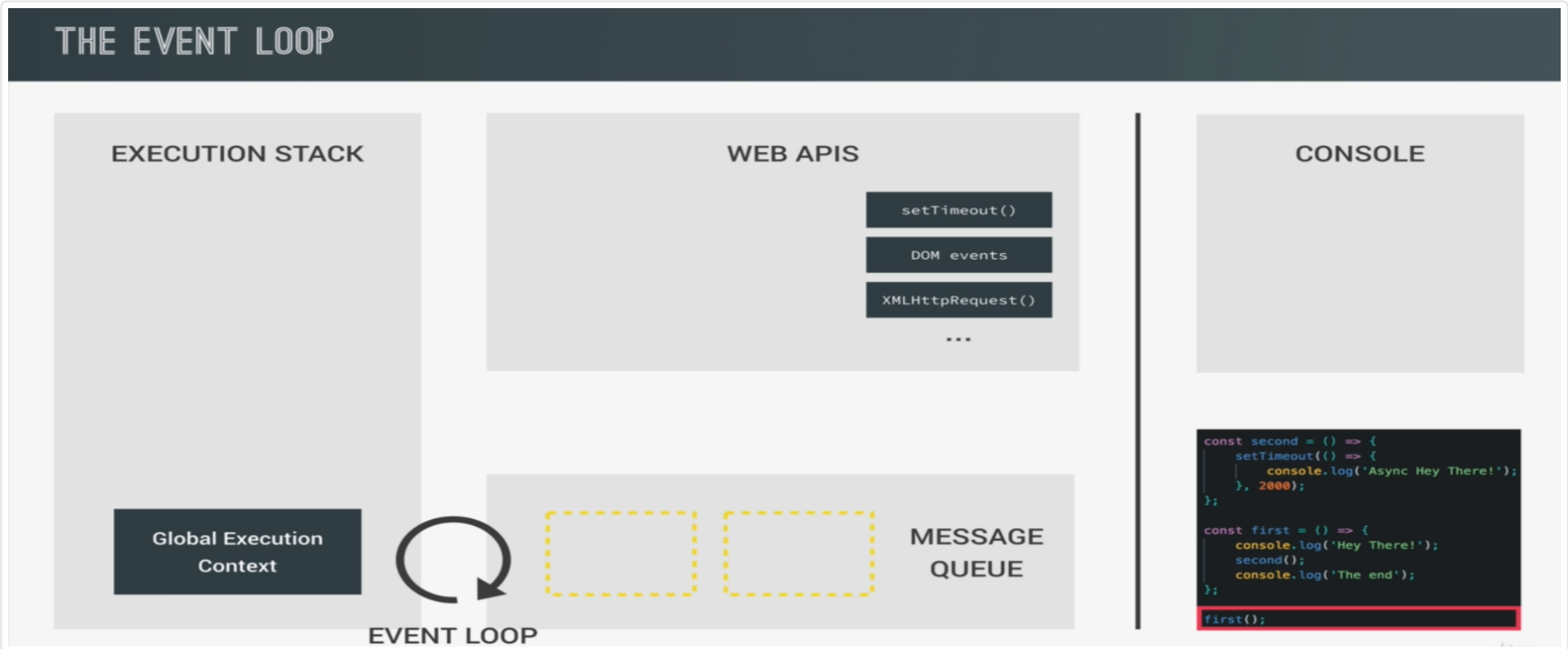
Here, we selected an image from our DOM and passed it onto the `processLargeImage()` function that we created. Now we know that this function will take some time to process the image. So just like before, we don't want the JS Engine to wait for the complete execution of `processLargeImage()` function. We can simply use Asynchronous JavaScript here. Now, we don't want the `processLargeImage()` function to stop executing while the image is being processed, because that's not what we want. Therefore, what we do is, we pass a callback function to the `processLargeImage()` function which we want to be called as soon as the `processLargeImage()` function is done processing. And that's the way we write Asynchronous JavaScript Code.

Therefore, there are 3 tenets of Asynchronous Coding:

1. Allow asynchronous functions to be run in the `background`.
2. We pass in callbacks that run once the main function has finished its work.
3. The JS Engine moves on immediately, so that the execution of the code is never blocked, which is also known as `Non-Blocking` code.

In summary, we can use callback functions to defer/postpone actions into the future in order to make our code non-blocking. But, how does that actually work behind the scenes of JavaScript? That's where the `Event Loop` comes in.

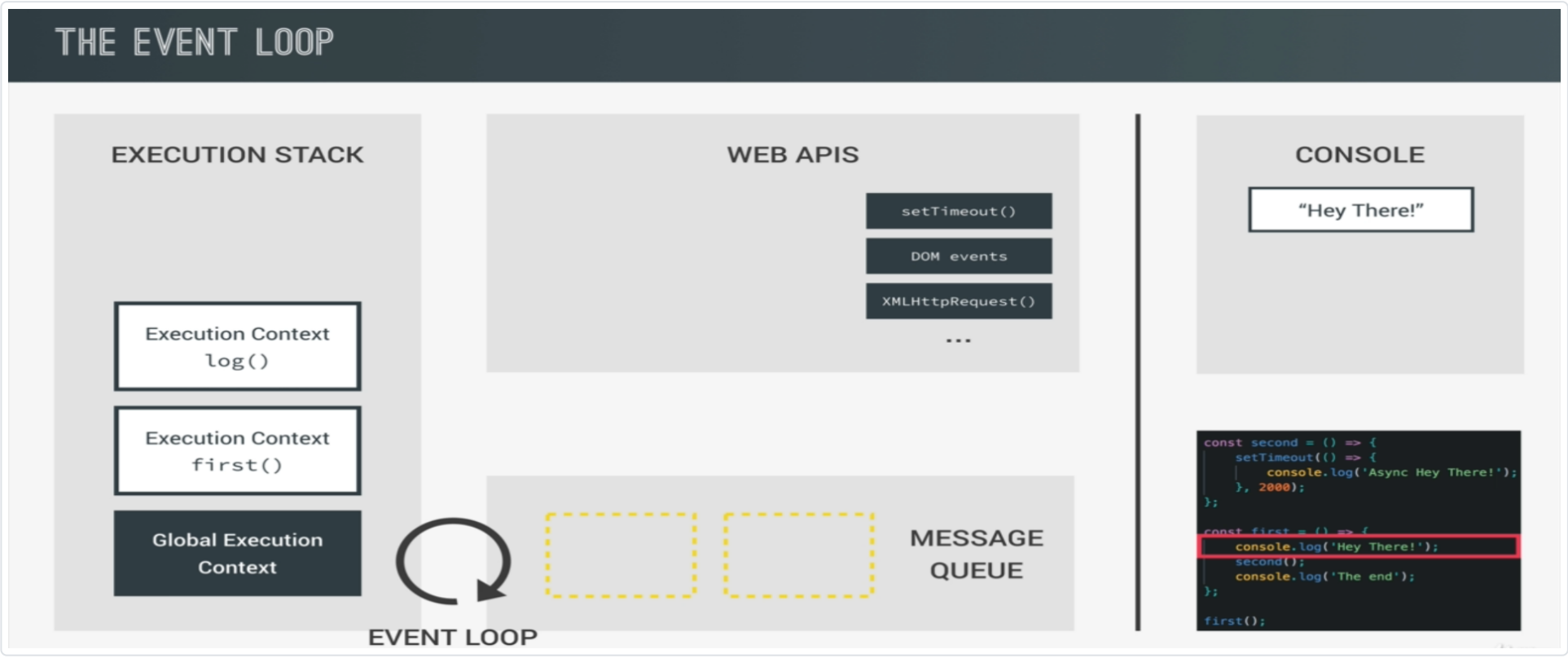
The `Event Loop` is part of the bigger picture of what happens behind the scenes of JavaScript when we call functions and handle events like DOM events. In the image below, we already know what the `Execution Stack` is and we also know what a `Message Queue` is, when we learned about `Events & Event Handling`. What's new is, the `Web API` and the `Event Loop`.



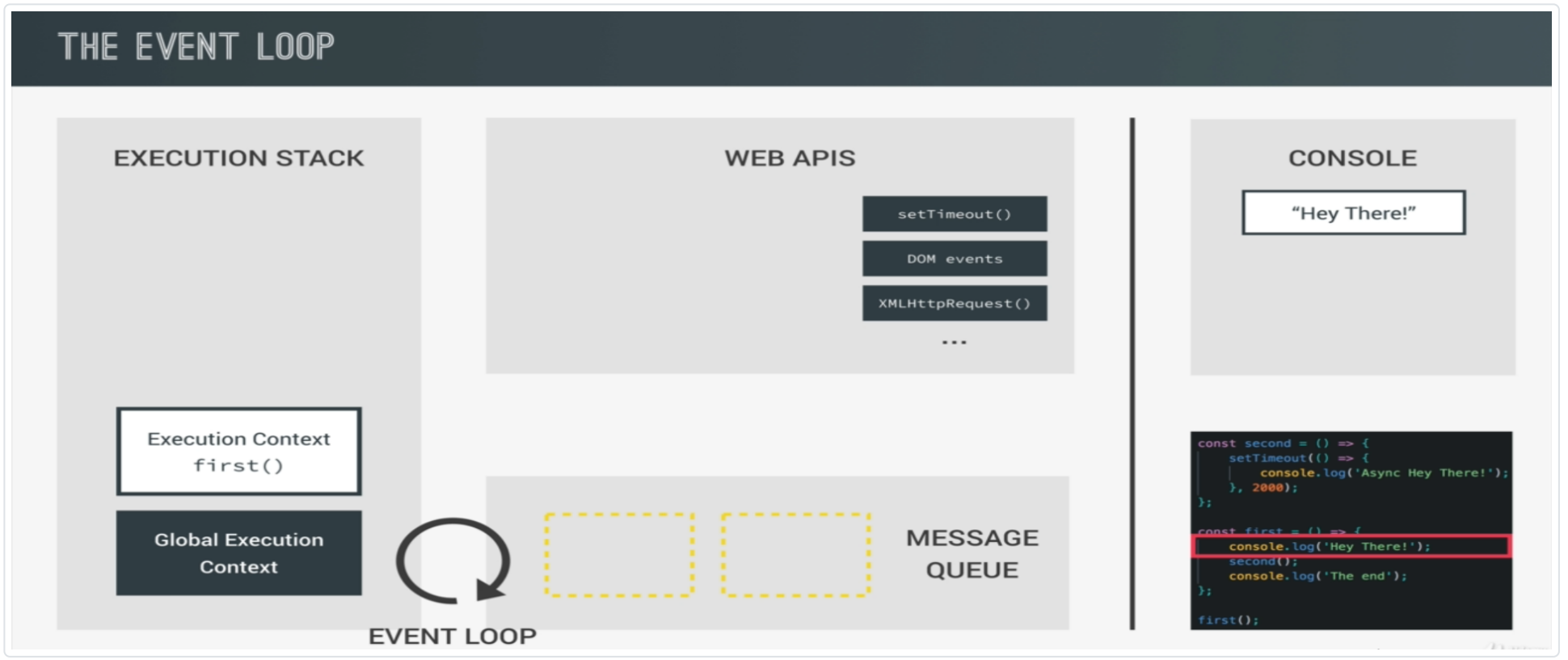
All of these together (i.e., Execution Stack, Message Queue, Web API, Console and the Event Loop) make up the JavaScript Runtime. This Runtime is responsible for how JS works behind the scenes as it executes our code, and it's extremely important to understand how all these pieces fit together in order to understand how an Asynchronous JavaScript code works. Let us understand how the Event Loop works using the code in the image above (bottom right corner).

It starts by calling the `first()` function as marked (with red border) in the code in the image above. Because of this, `first()` function's Execution Context is pushed on to the top of Execution Stack as shown below.

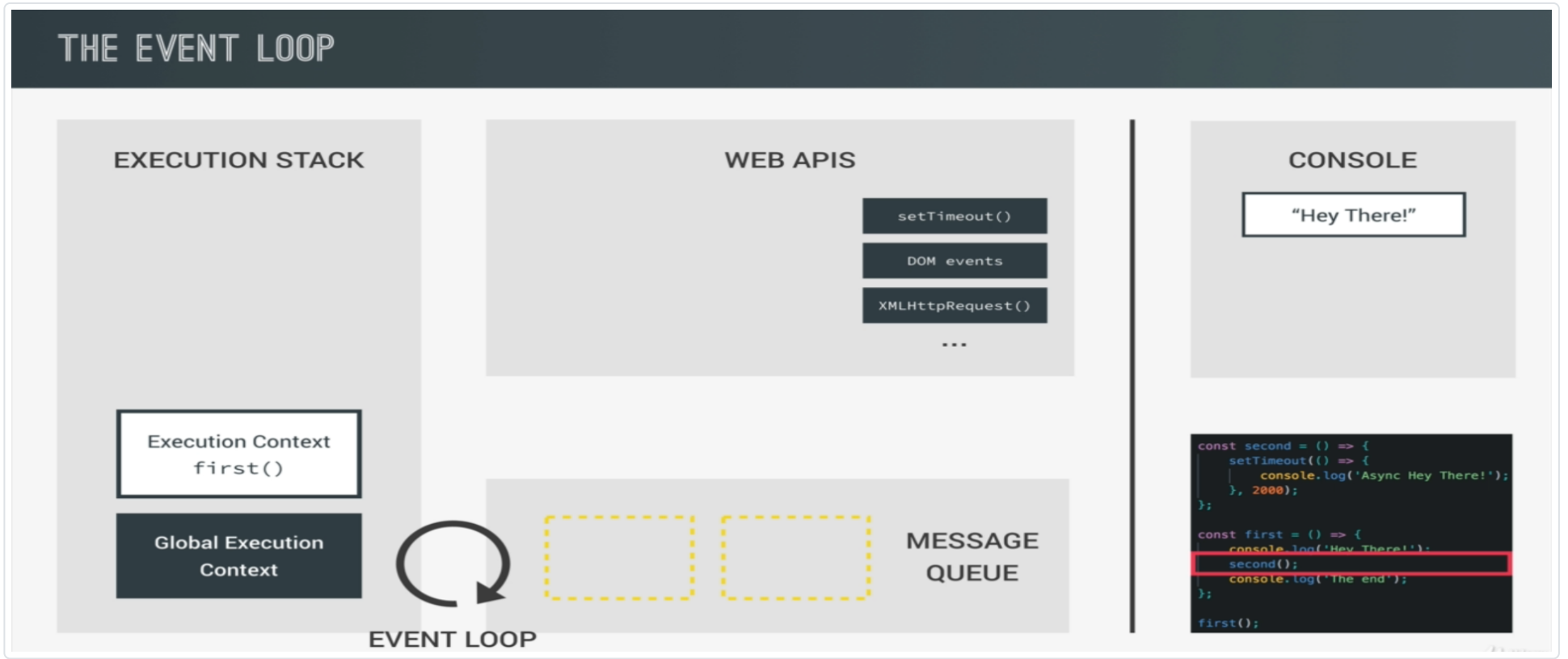
In the next line of code, `console.log("Hey There!");` is called, and the Execution Context of `log()` is pushed on top of the Execution Stack. Finally, "Hey There!" is logged onto the Console as shown below.



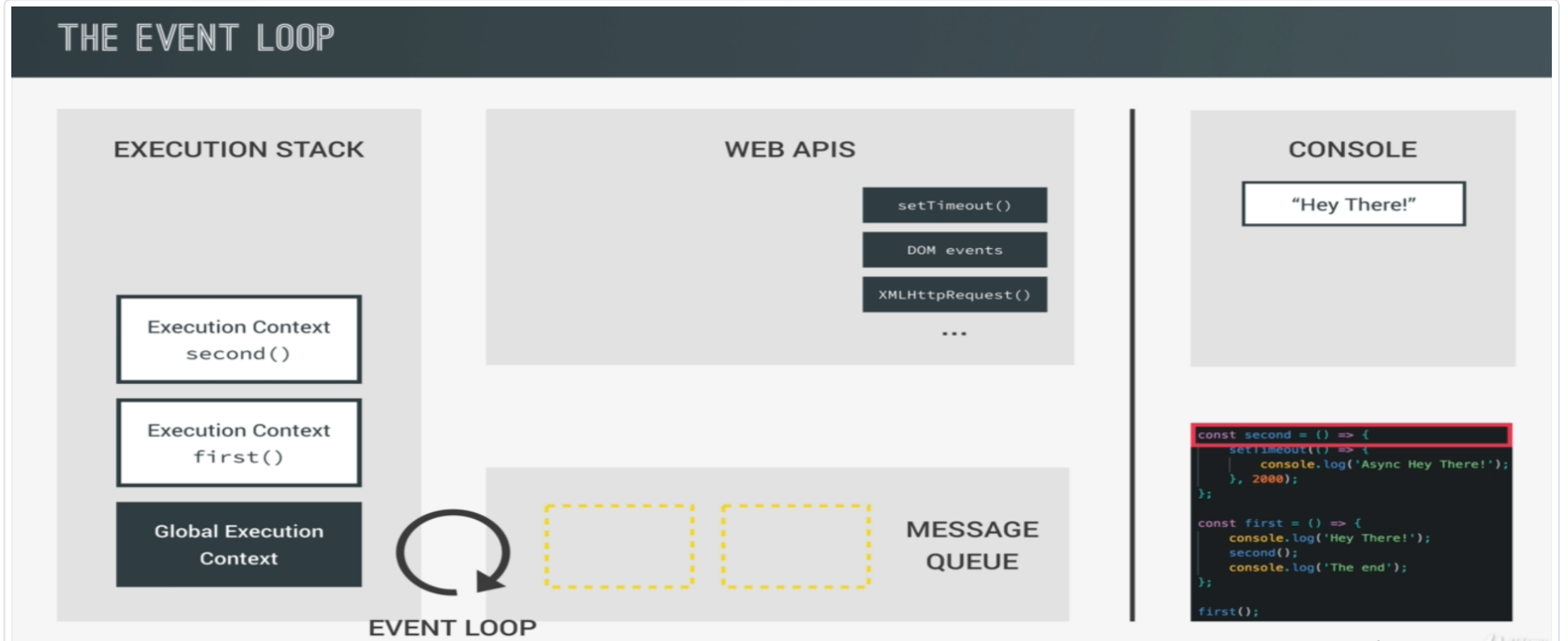
After that, the `log()` function returns and the Execution Context of `log()` function, pops of the Execution Stack as shown below.



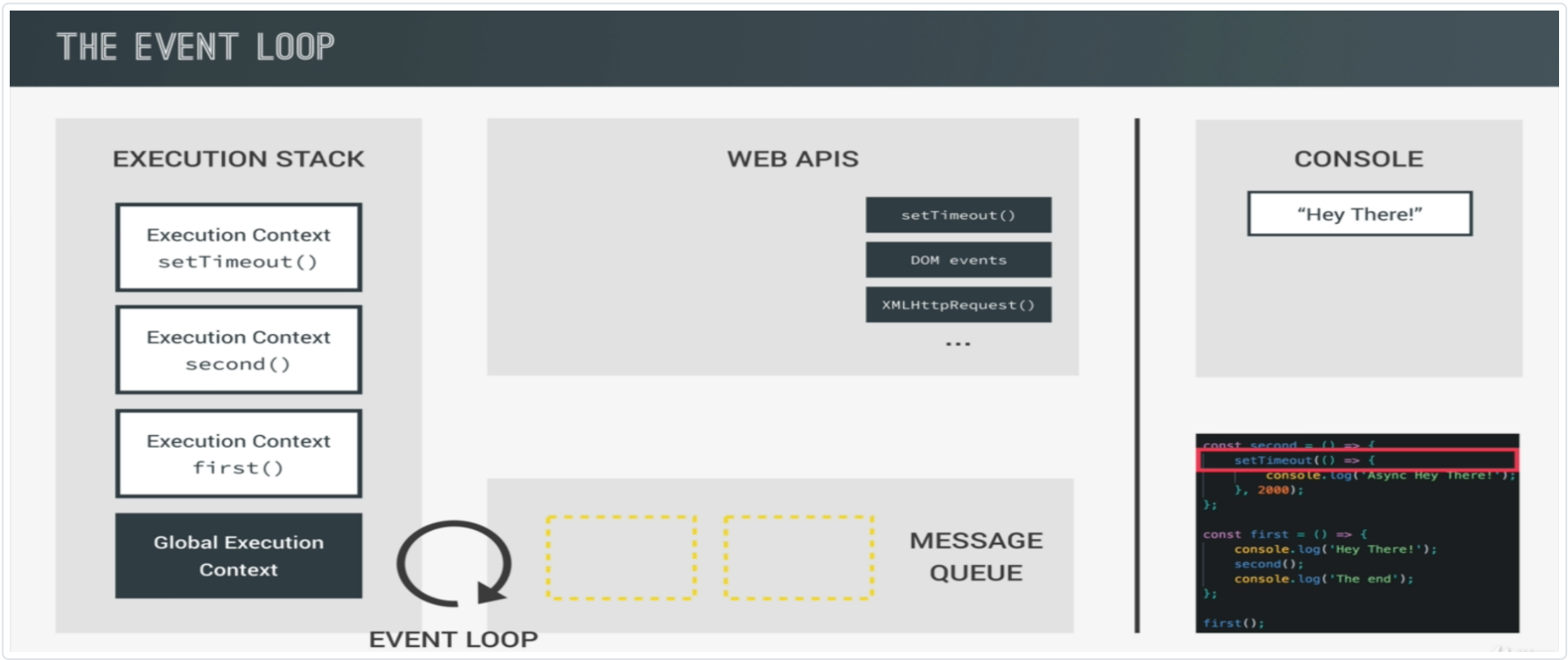
And now, in the `first()` function's Execution Context, `second()` is called as shown below.



After `second()` is called, its Execution Context is pushed on top of the Execution Stack as shown below.

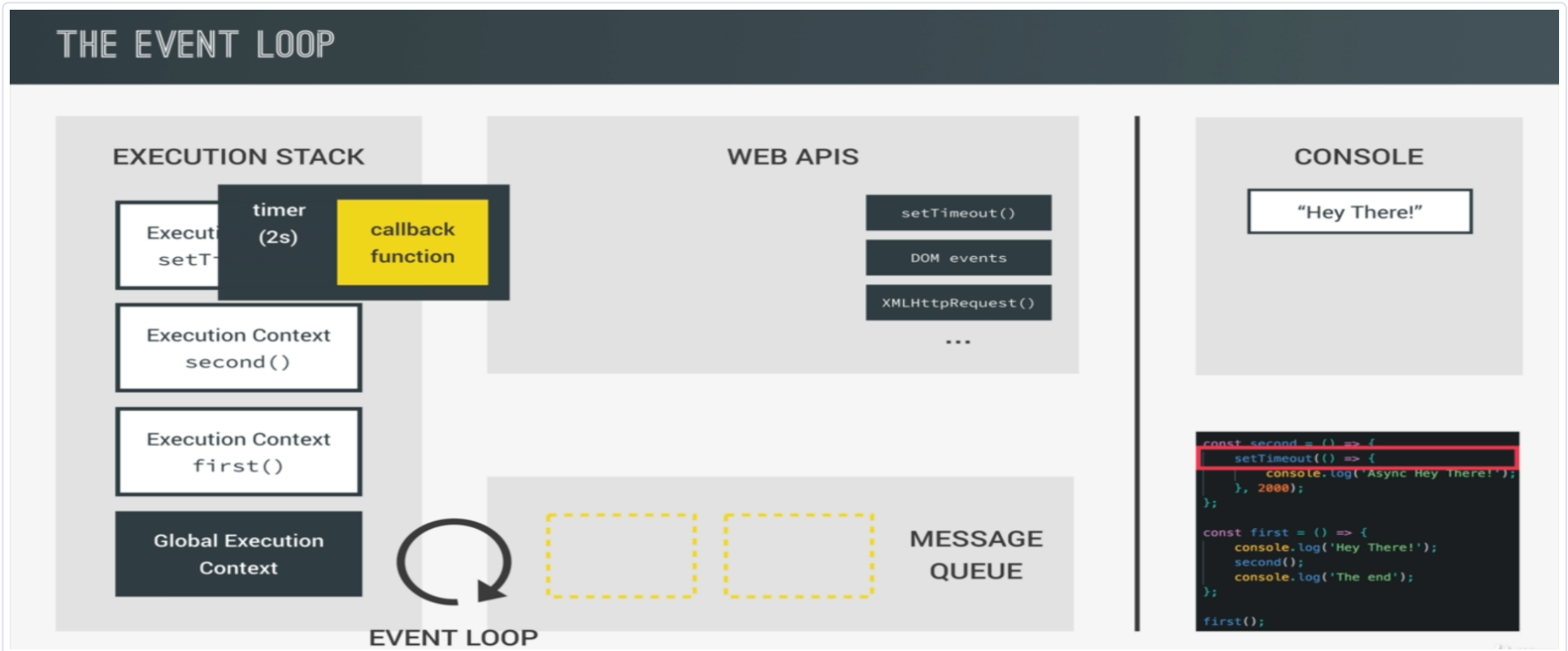


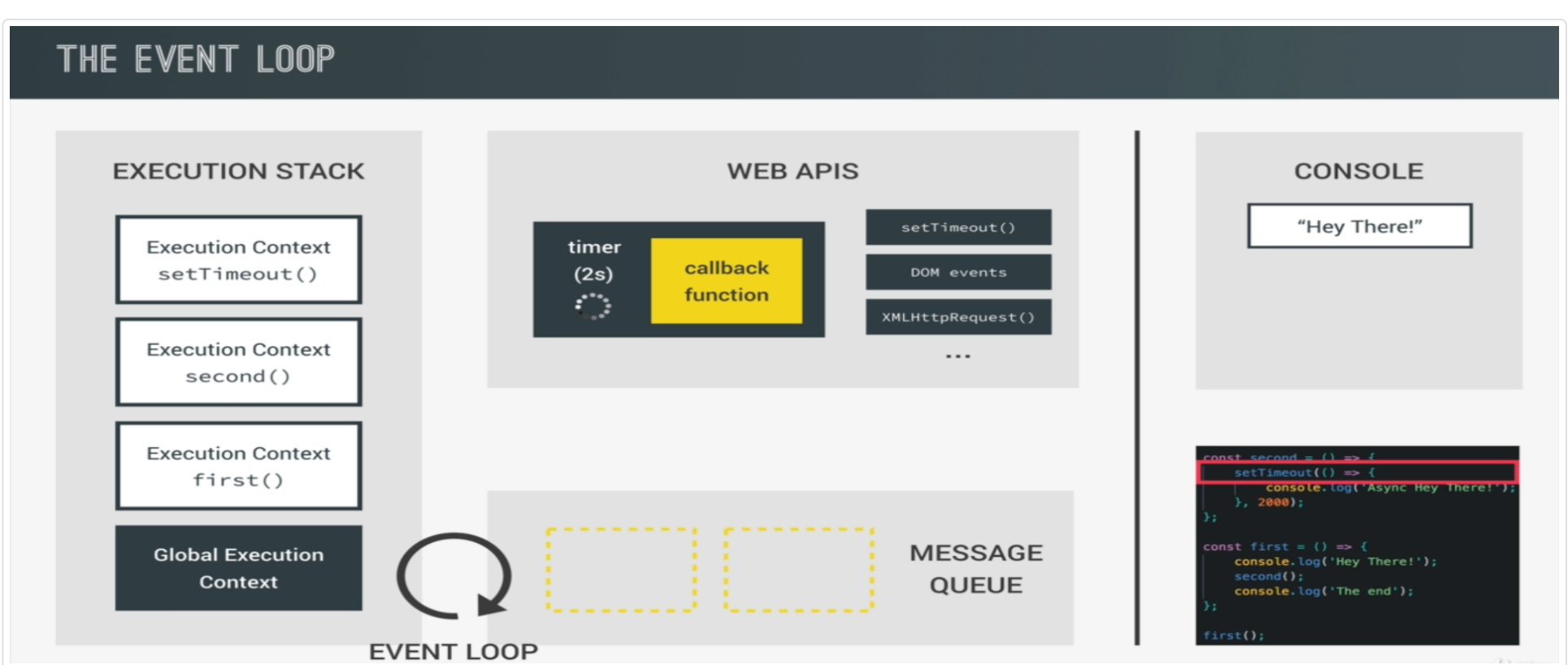
In `second()`'s next line, `setTimeout()` is called and its Execution Context is pushed on top of the Execution Stack as shown below.



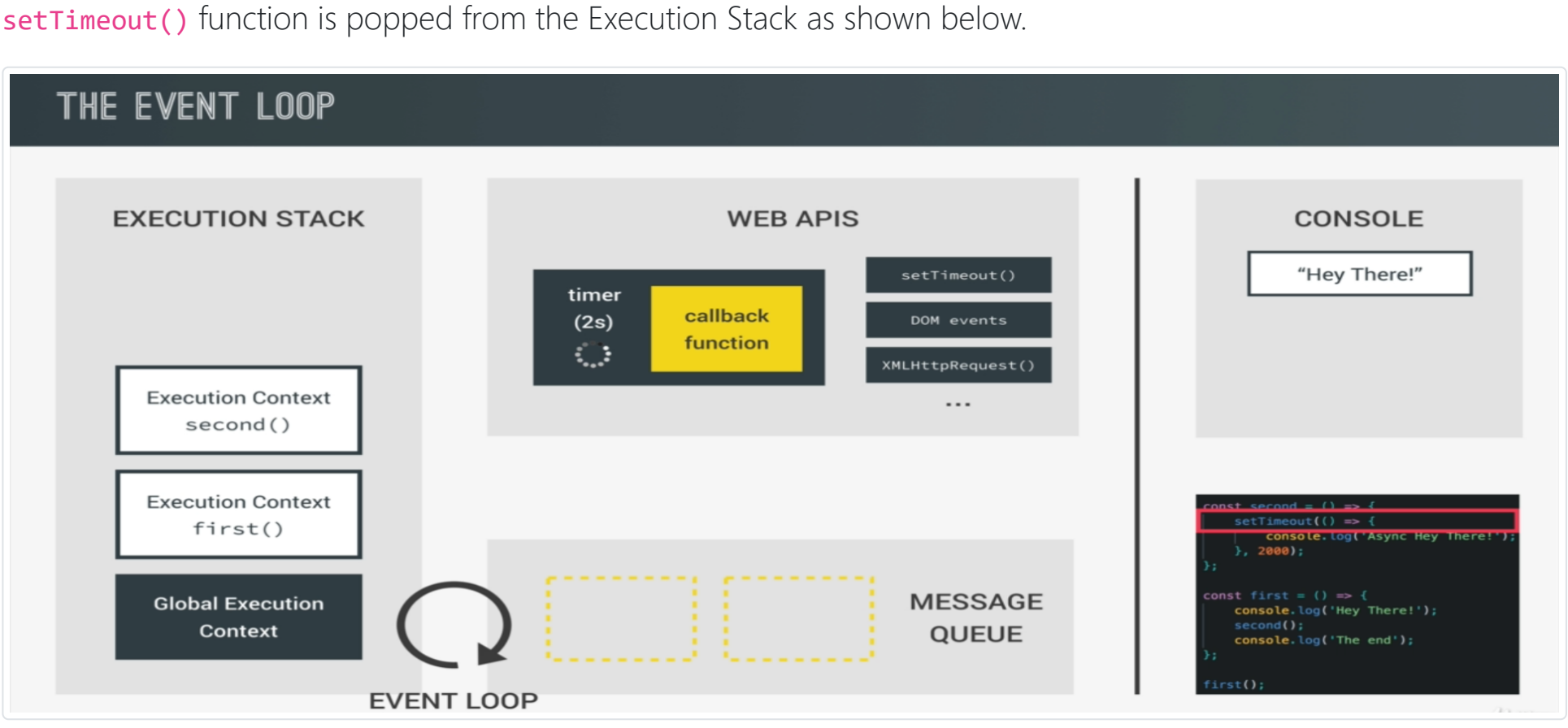
Before we move on any further, we have to know a little bit about the `setTimeout()` function. Where does this `setTimeout()` function actually come from? The `setTimeout()` function is a part of something called the Web APIs, which actually live outside the JavaScript Engine itself. So, stuff like DOM Manipulation, Set Timeouts, HTTP Requests for AJAX, Geo Location, Local Storage and tons of other things actually live outside of the JavaScript Engine as Web APIs. We just have access to these Web APIs because they're also the part of the JavaScript Runtime. And this is exactly where the timer will keep running for 2000ms (i.e., 2 seconds), asynchronously, so that our code can keep running without being blocked.

So, when we call the `setTimeout()` function, the timer is created together with the callback function right inside the Web API's environment as shown below.

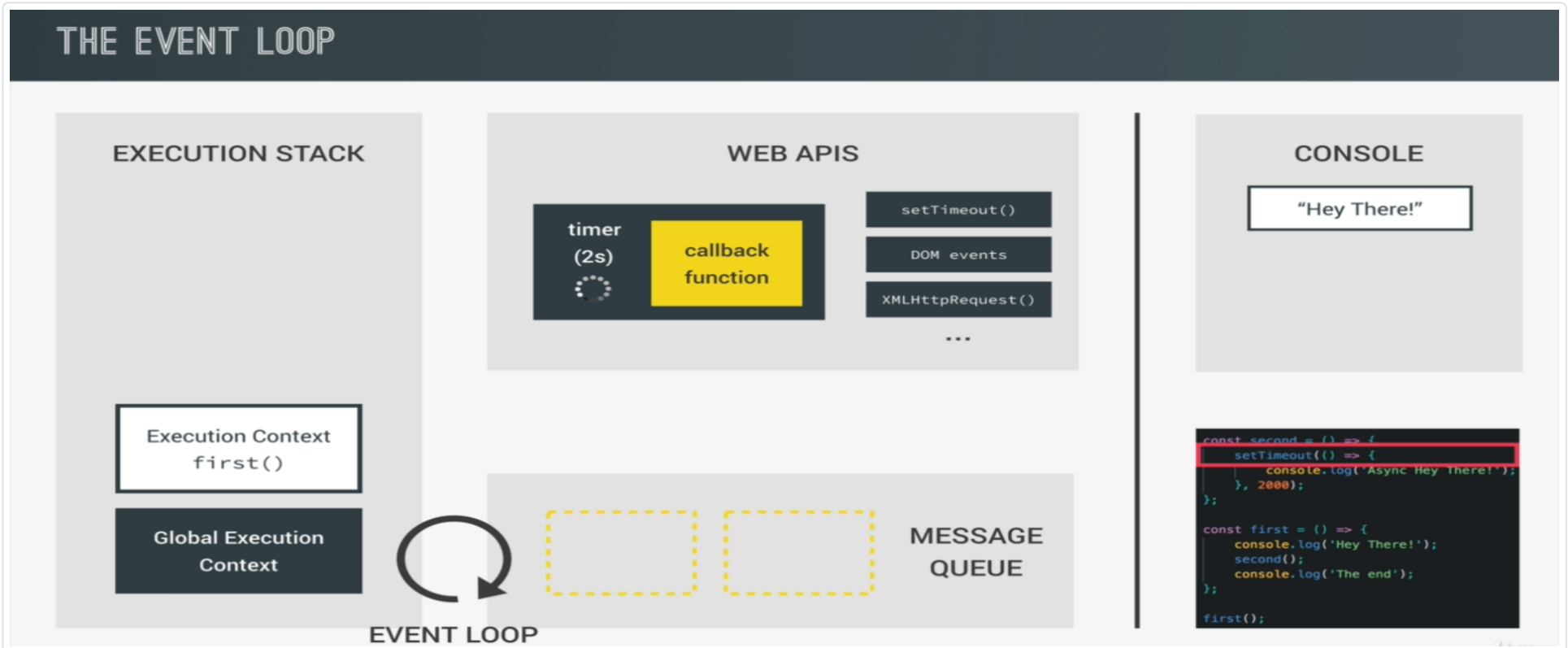




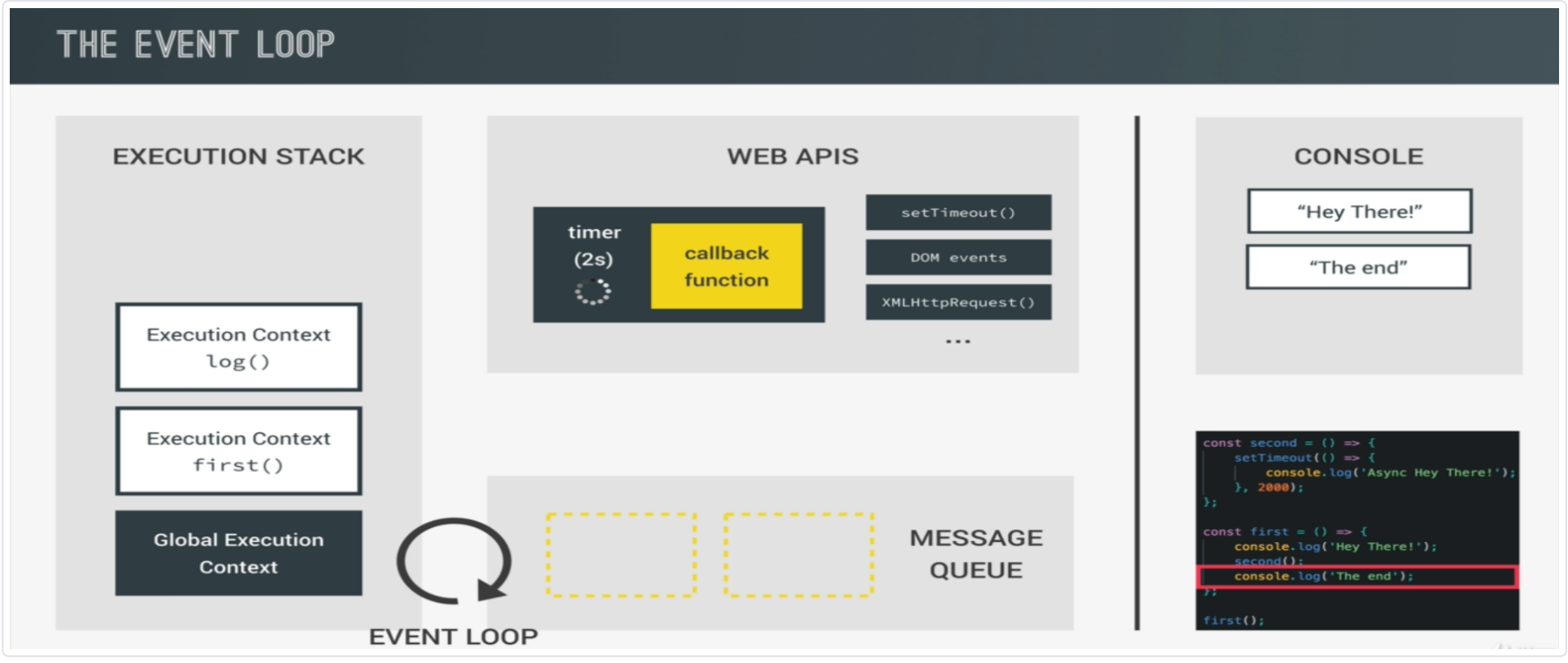
The callback function along with the timer keeps sitting in the Web API's environment until the timer runs out of the specified time (in our case, it is 2000ms), all ofcourse in an asynchronous way. Therefore, the callback function stays attached to the timer in the Web API's environment, until the timer runs out. And so, since the timer keeps working, basically in the background, we don't have to wait for the timer to finish, and can keep executing our code. And because of this asynchronous nature of JavaScript, the `setTimeout()` function returns and the Execution Context of `setTimeout()` function is popped from the Execution Stack as shown below.



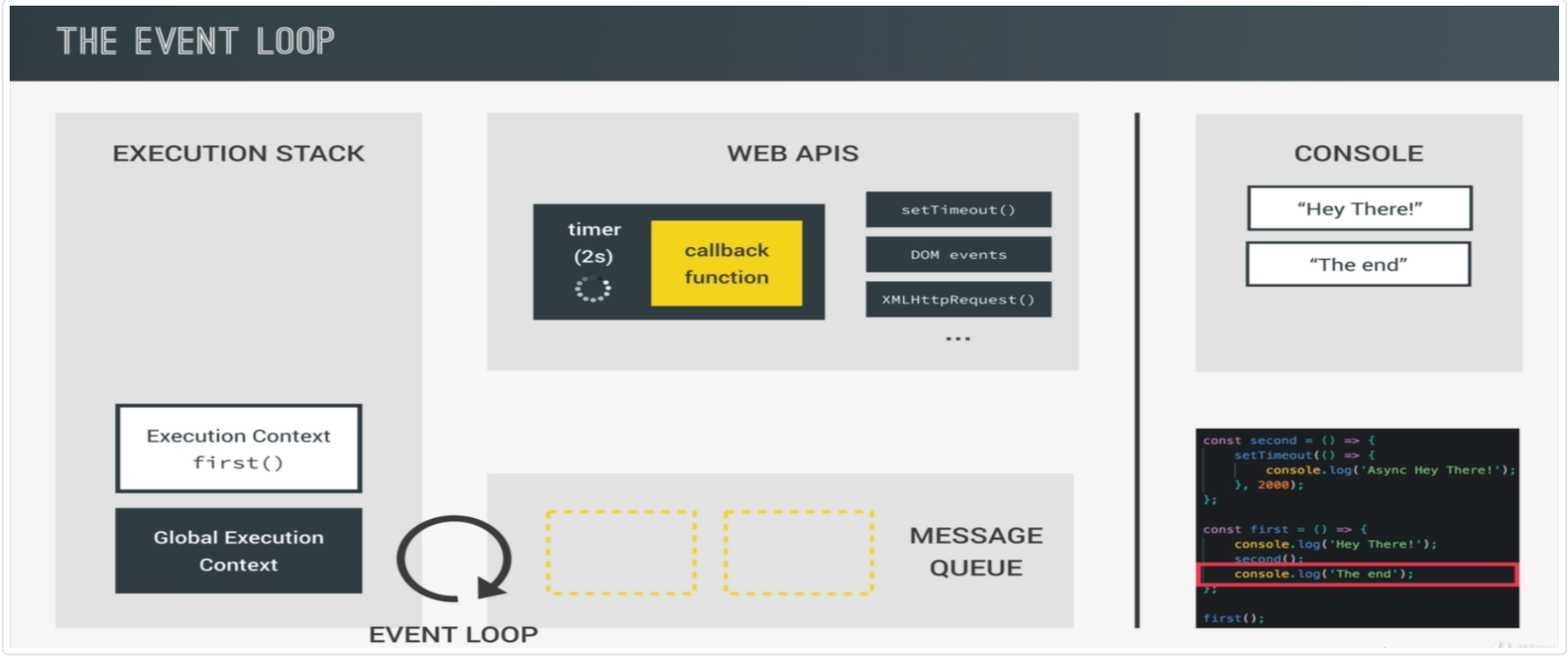
After that, `second()` function's execution is completed, so the Execution Context of the `second()` function pops off the Execution Stack as shown below.



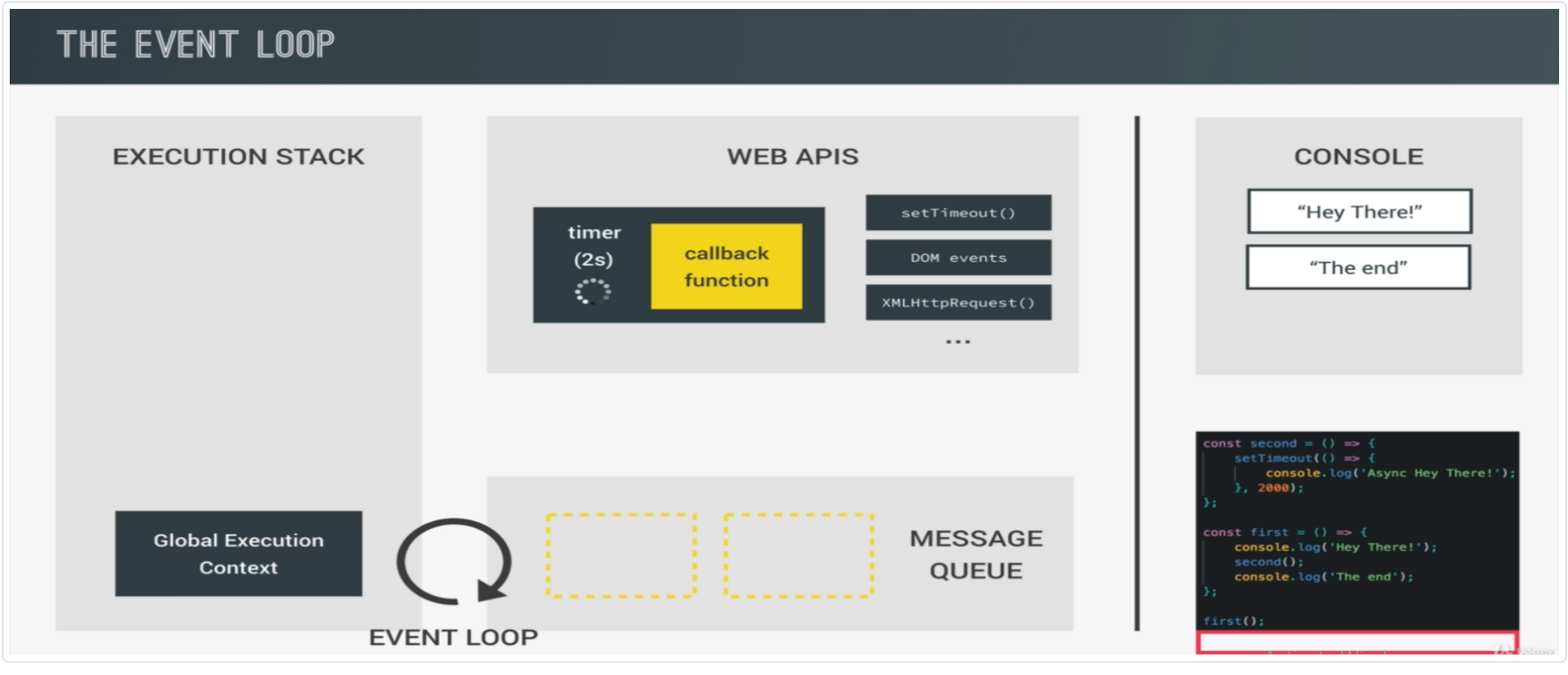
Now we're back to the Execution Context of the `first()` function, where `console.log("The end");` is executed and for that, `log()`'s Execution Context is pushed on top of the Execution Stack and "The end" is logged onto the console as shown below.



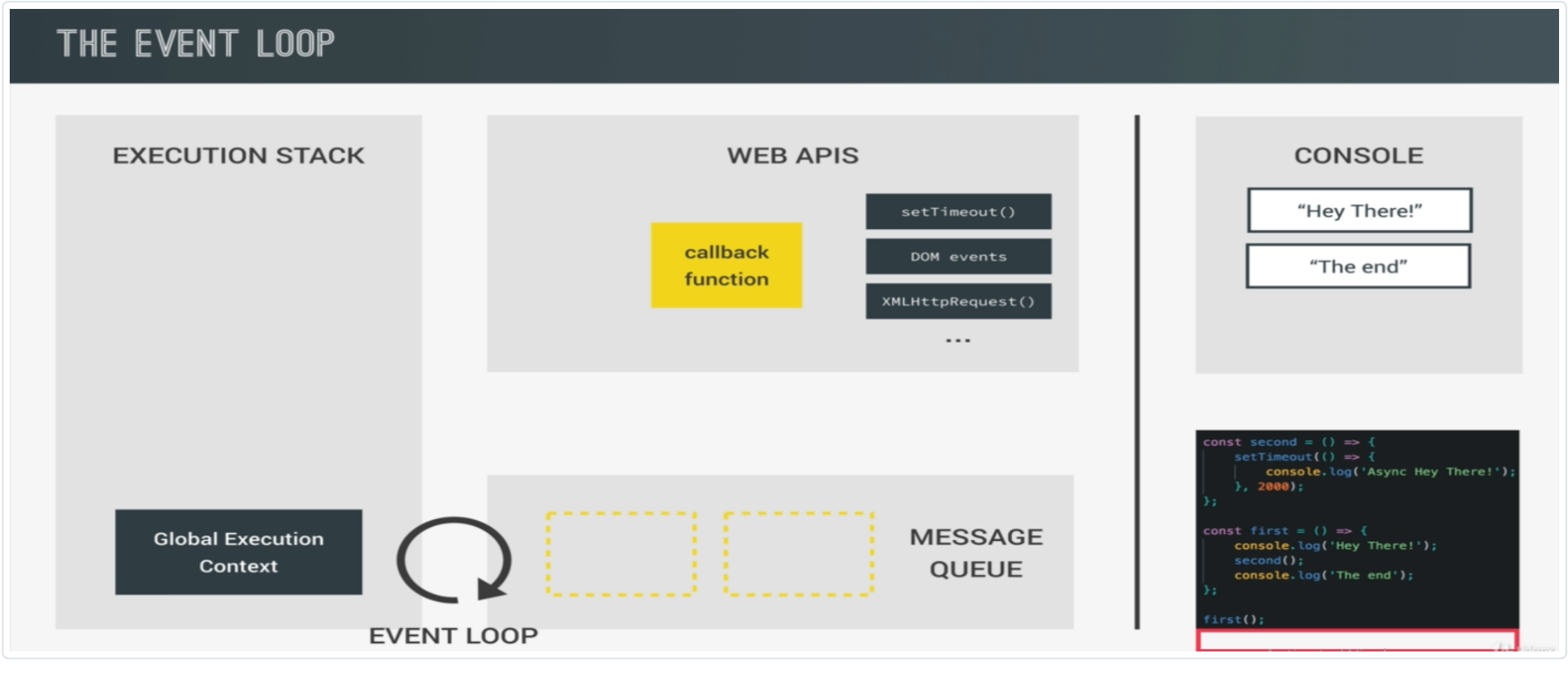
The `log()`'s Execution Context is popped off from the Execution Stack and then the control returns to the `first()` again as shown below.



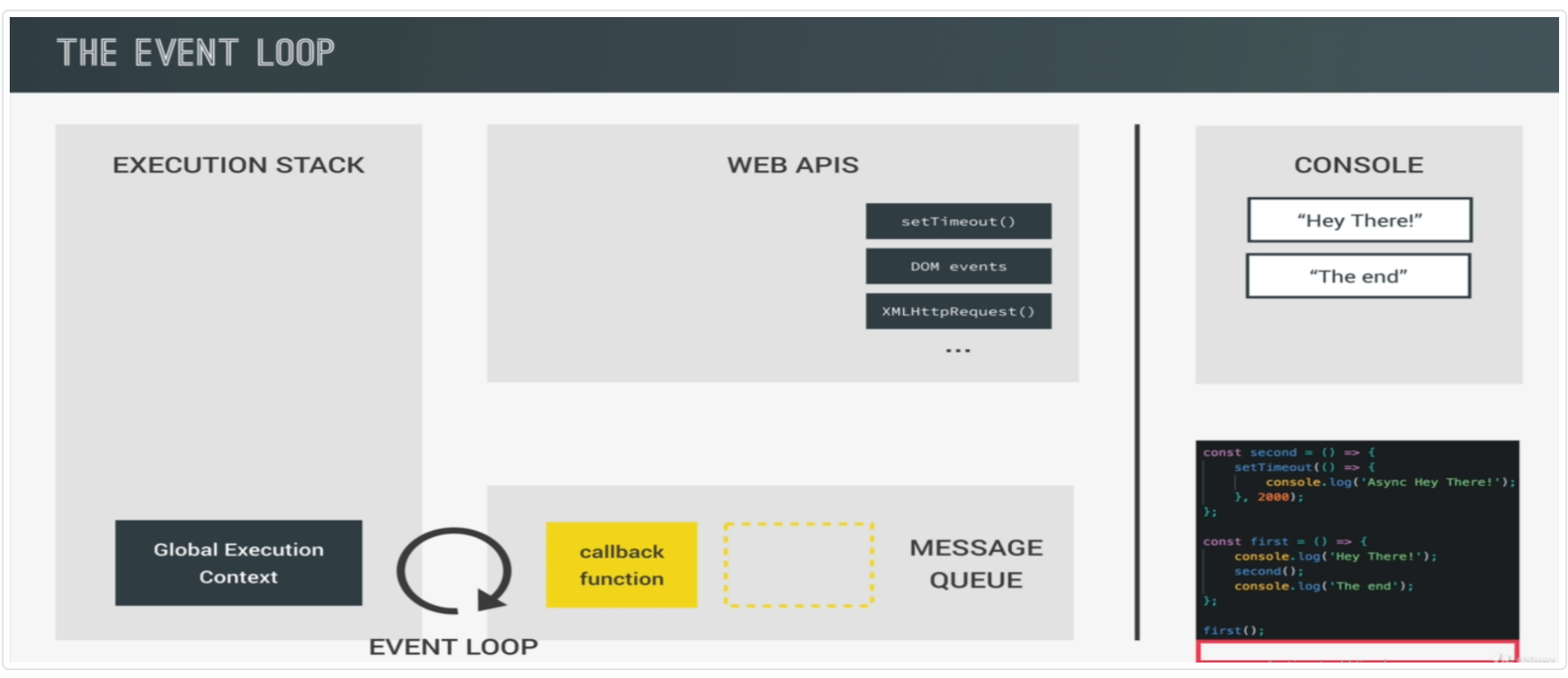
In `first()`'s Execution Context, there's no other statement to be executed, so `first()`'s Execution Context is now popped off from the Execution Stack and we'll be back to the initial state, as shown below.



Right now we've executed all our code in a synchronous way and we have the timer running asynchronously in the background. Now, after timer runs out (in our case it is 2000ms i.e., 2 seconds), what will happen to the callback function now?

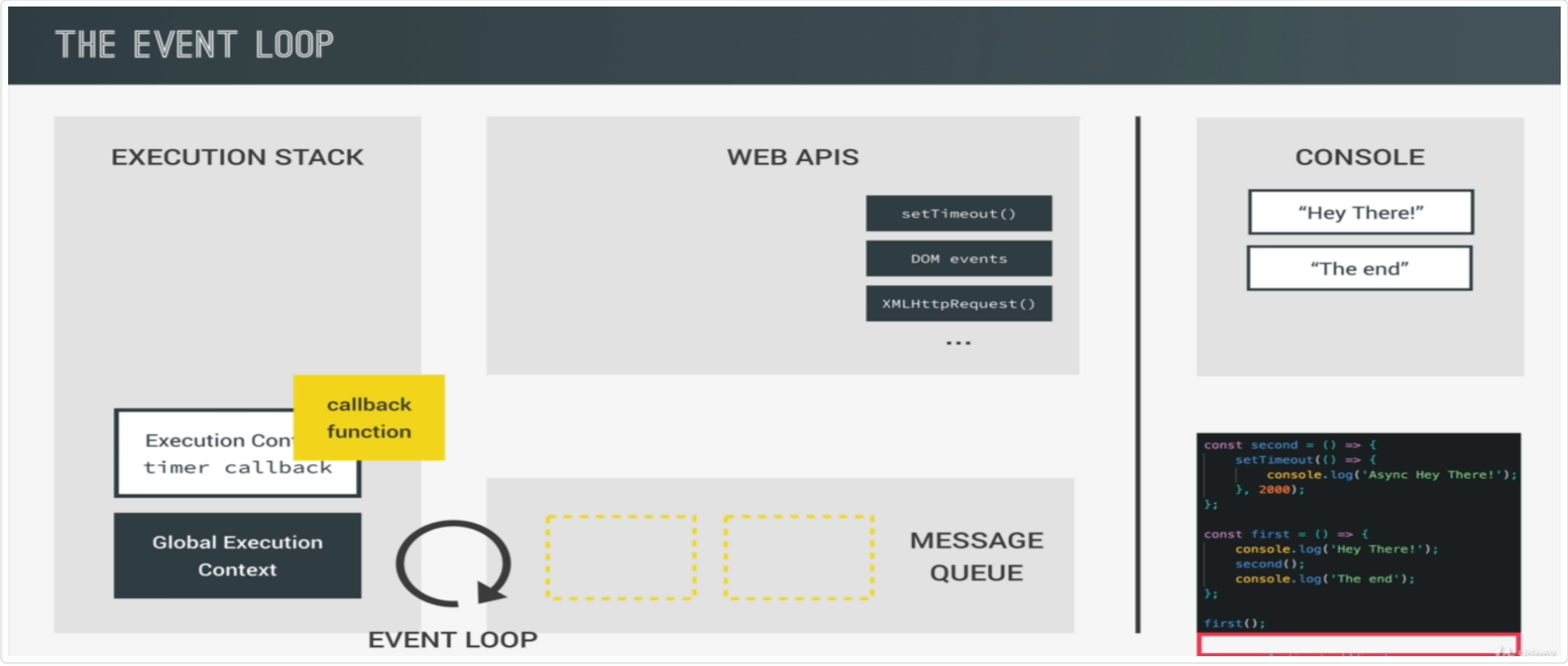


The callback function moves to the Message Queue and it waits to be executed as soon as the Execution Stack is empty (i.e., there's only Global Execution Context presiding on top of the Execution Stack) as shown below.

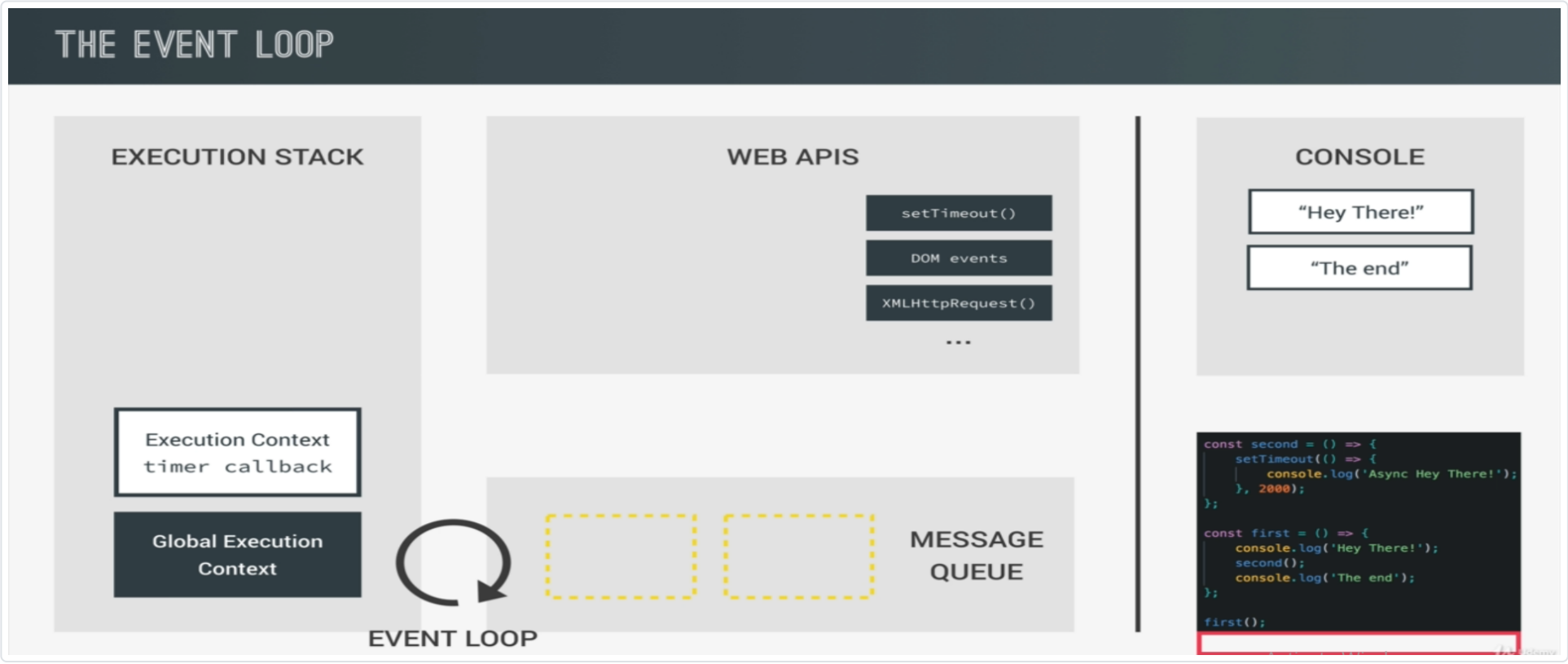


And this is what exactly happens with DOM Events as well, because it works the exact same way. In the case of DOM Events, our Event Listeners sit in the Web API's Environment waiting for a certain event to happen. As soon as the required Event occurs, the callback function related to that Event is placed in the Message Queue, where the callback waits for itself to be executed. The callback function in the Message Queue always waits till the Execution Stack has no other Execution Context(s) other than Global Execution Context.

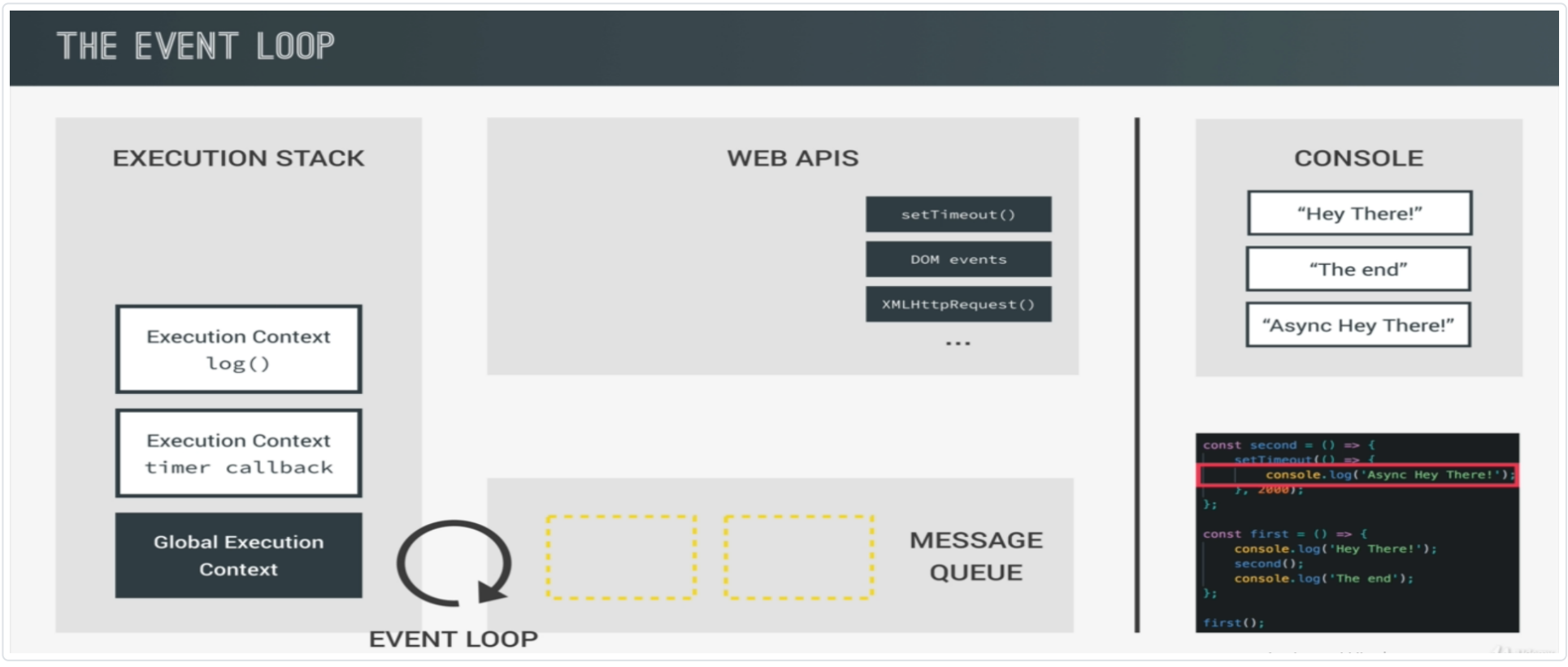
How are these callback functions in the Message Queue executed? That's where finally, the `Event Loop` comes in. The job of the Event Loop is to constantly monitor the Message Queue and the Execution Stack, and to push the first callback function in-line (in the Message Queue) on to the Execution Stack as soon as the Execution Stack is empty (i.e., the stack only has GEC). In our example, the Execution Stack right now is actually empty and also, we have one callback function waiting to be executed, and so the Event Loop takes that callback function from the Message Queue and pushes it on top of the Execution Stack as shown below.



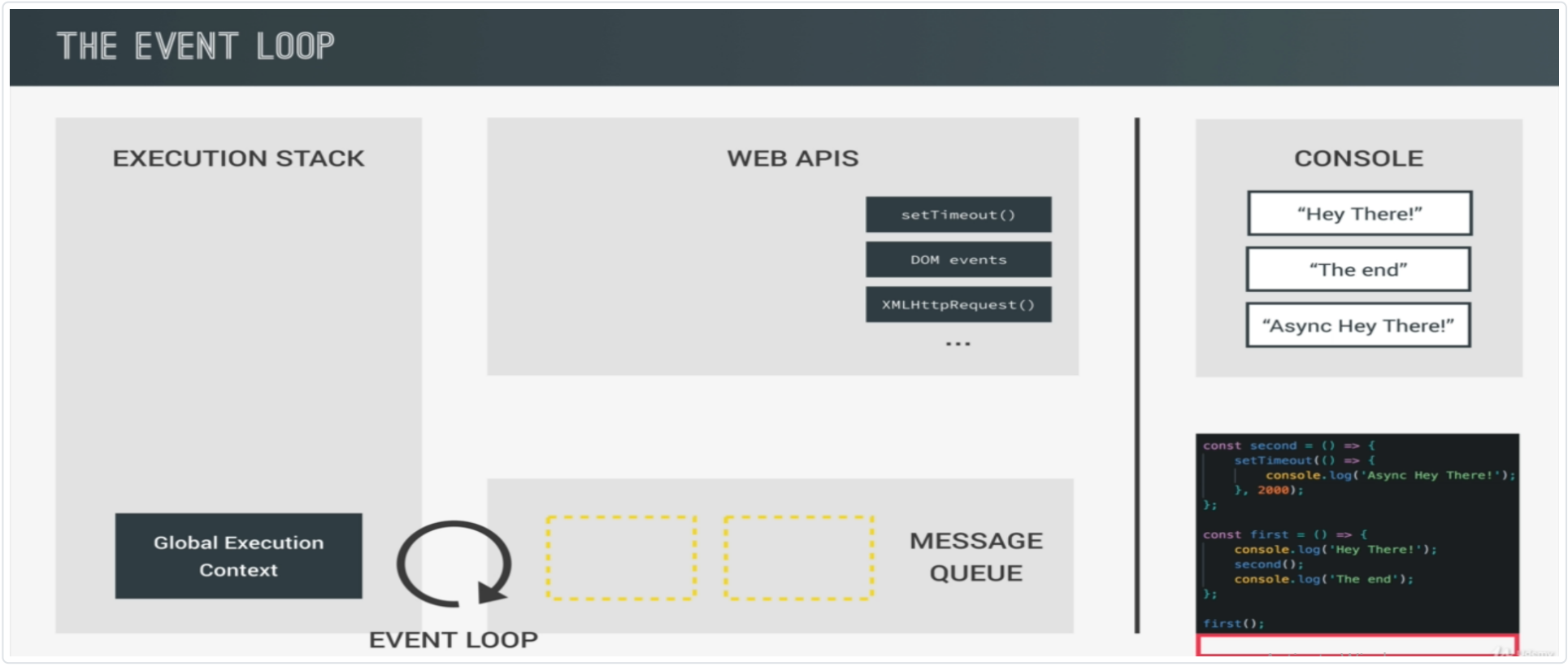
For that callback function, a new Execution Context is created on top of the Execution Stack as shown below.



Inside the callback function now, `console.log("Async Hey There!");` executes, for which `log()`'s Execution Context is pushed on top of the Execution Stack which logs "Async Hey There!" on to the Console, as shown below.



And now, the Execution Context of `log()` function is popped off of the Execution Stack, and also, the Execution Context of the callback function is also popped off of the Execution Stack, and the control returns back to the initial state as shown in the images below.



Now, if there were some more callbacks waiting right now, like data coming back from an AJAX request or the Handler of a DOM Event, then the Event Loop will continue pushing those callbacks from the Message Queue onto the Execution Stack until all of those callbacks in the Message Queue are processed. And this is how the Event Loop works.