5 JULY 2014 / ENGINEERING

# What is the JavaScript event loop?

## Introduction

If you're like me, you love JavaScript. Yea, it's not the perfect language but seriously, is there even such a thing as a perfect language?! So despite it's flaws and all, I love programming on the web and how JavaScript enables me to build applications that connect with the world.

But JavaScript is deep—it's got a complex interior that takes a while to really understand. One of these deeper aspects of JavaScript is it's Event Loop. It's possible to program in JavaScript for years and not really understand exactly how the event loop works in JavaScript. However, through this blog post I'm hoping to shed some light into what the event loop is and how it's really not that complicated.

## JavaScript in the browser

When we think of JavaScript, we commonly think of it in the context of a web browser—this makes sense since most of us code our JavaScript for the client-side. However, it's important to realize that running any web application actually involves a collection of technologies, such as a JavaScript Engine (such as Chrome's V8), a collection of Web API's (such as the DOM), and the Event Loop and Event Queue.

Having seen the above list of items, you might be thinking, "Holy moly, this seems super complicated …" and you'd be right—but as we'll soon see, the basic idea of how this all works is actually not too complicated even if the actual implementation might be over our heads!

Before diving into the event loop, we need a basic understanding of the JavaScript Engine and what it does.

## JavaScript engine

There are actually several different implementations of JavaScript engines but by far the most popular version is Google Chrome's V8 engine (which is not limited to the browser but also exists in the server via NodeJS). But what exactly does the JavaScript Engine do? Well, it's actually quite simple—it's job is to go through all the lines of JavaScript in an application and process them one at a time. That's right—one at a time, meaning that JavaScript is *single-threaded*. The main repercussion of this is that if you're running a line of JavaScript that happens to take a longgggggg time to return, then all the code after that will be *blocked*. And we don't want to write code that's blocking—especially on the browser. Imagine that you're on a web site and click on a button … and

(assuming no bugs) is the button click triggered some JavaScript code to execute but it's blocking.

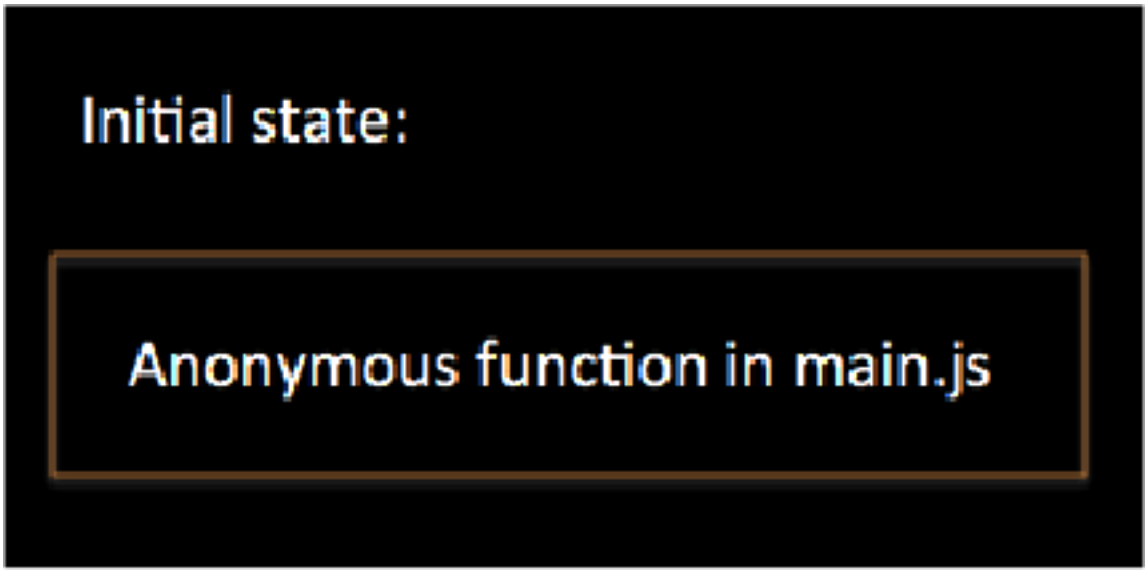
So how does the JavaScript engine know how to process a single line of JavaScript at a time? It uses a call stack. You can think of a call stack like entering an elevator—the first person who enters the elevator is the last person to exit the elevator, whereas the last person to enter is the first to exit.

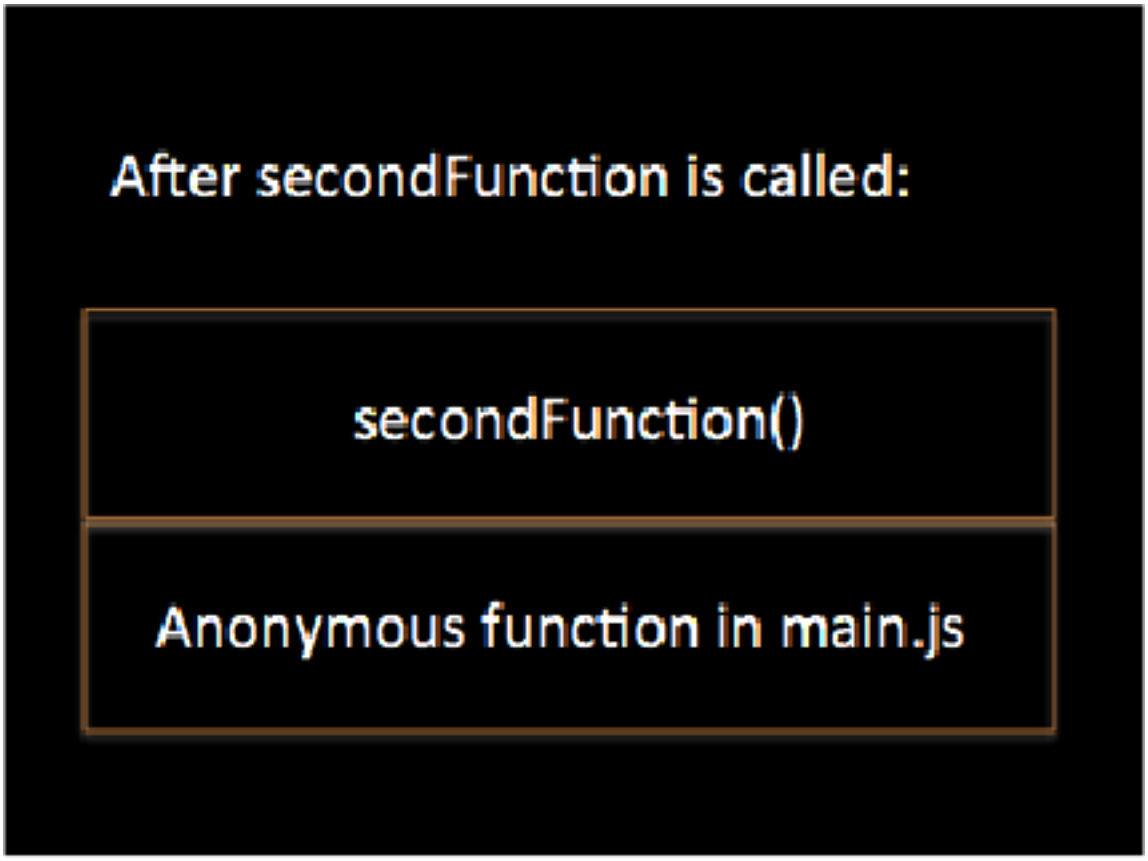
Let's see an example!

```
/* Within main.js */

var firstFunction = function () {
  console.log("I'm first!");
};

var secondFunction = function () {
  firstFunction();
  console.log("I'm second!");
};

secondFunction();

/* Results:
 * => I'm first!
 * => I'm second!
 */
```

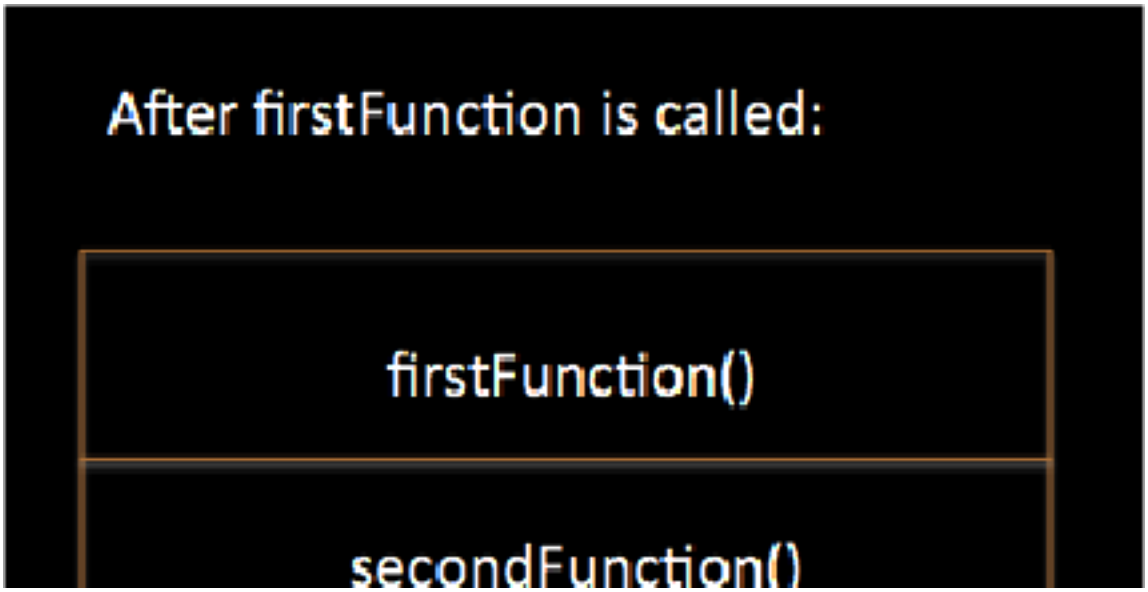
And here's a sequence of what's going on with the call stack:

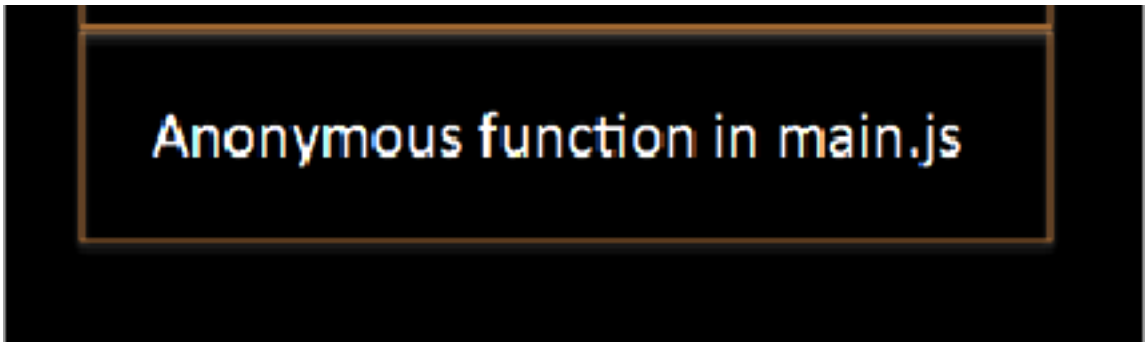
- Main.js is first executed:



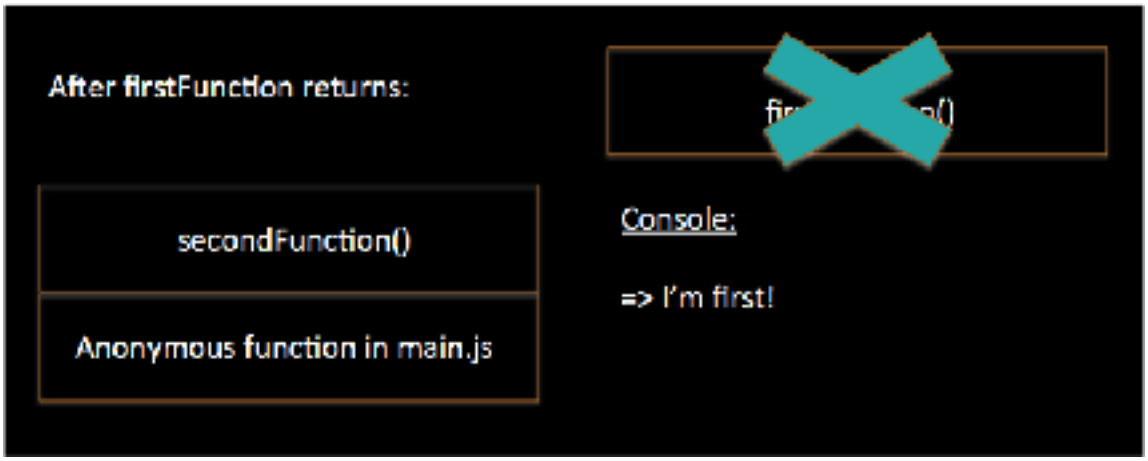



- Invoking secondFunction causes firstFunction to be invoked:

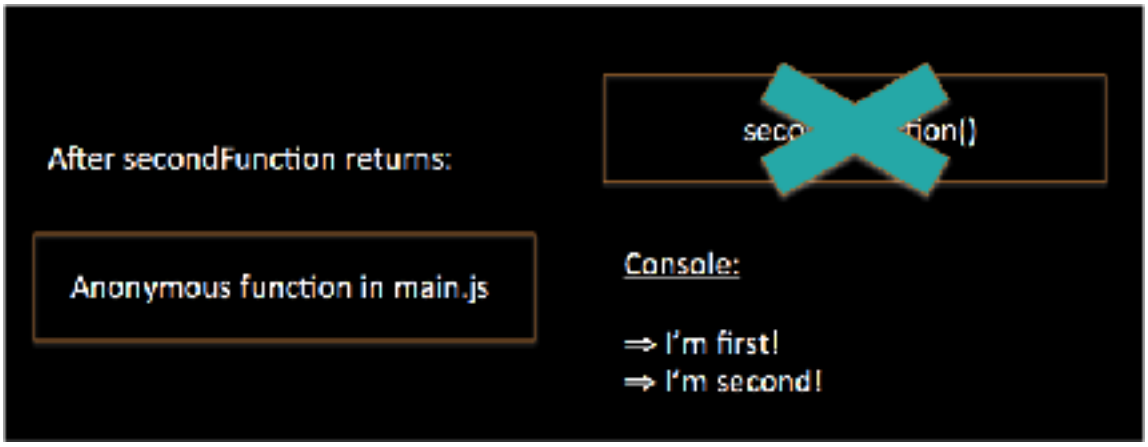




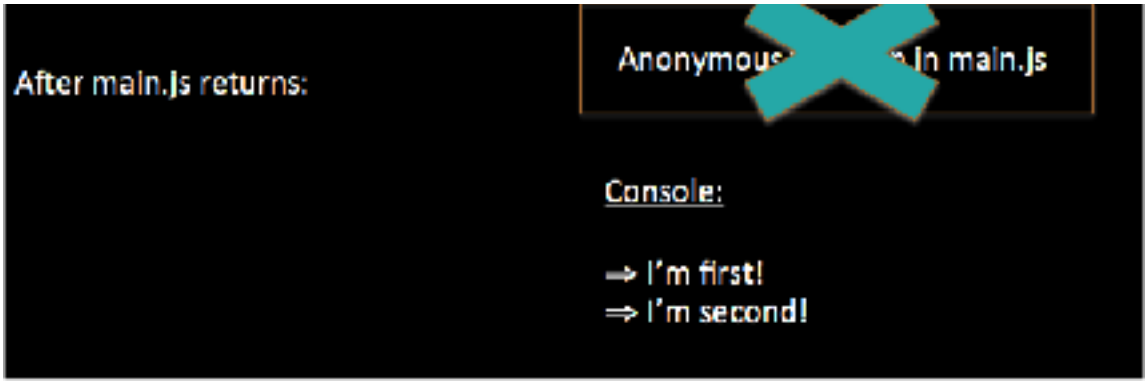

- Executing firstFunction causes the string “I’m first!” to be logged out first and since there are no more lines to execute in firstFunction, firstFunction is removed from the call stack:




- Execution continues in secondFunction and “I’m second!” is logged out. Once the string is logged out though, there are no more lines to execute in secondFunction and secondFunction is removed from the call stack:




- Finally, since there are no more lines of code to execute in main.js, main.js is removed from the call stack as well:



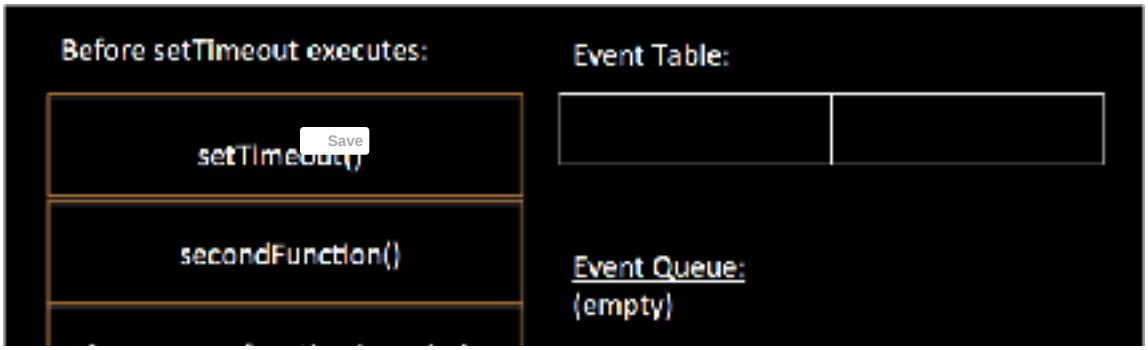

## Um ok, but can we talk about the Event Loop??

Now that we get how the call stack works in the JavaScript engine, let's get back to the idea of blocking code. We know we should avoid it, but how? Luckily for us, JavaScript provides a mechanism and it's via *asynchronous callback functions*. Whoa big words, right? No worries—an asynchronous callback function is just like any other function you're used to writing in JavaScript, with the **added caveat that it doesn't get executed till later**. If you've used JavaScript's setTimeout function, then you're already familiar with asynchronous callback functions! Let's take a look at an example:

```
/* Within main.js */

var firstFunction = function () {
 console.log("I'm first!");
};

var secondFunction = function () {
 setTimeout(firstFunction, 5000);
 console.log("I'm second!");
};

secondFunction();

/* Results:
 * => I'm second!
 * (And 5 seconds later)
 * => I'm first!
 */
```

And here's a sequence of what's going on with the call stack (we'll fast forward a little bit to where the action starts):

- After secondFunction is placed on the call stack, the setTimeout function is invoked and also placed on the call stack:

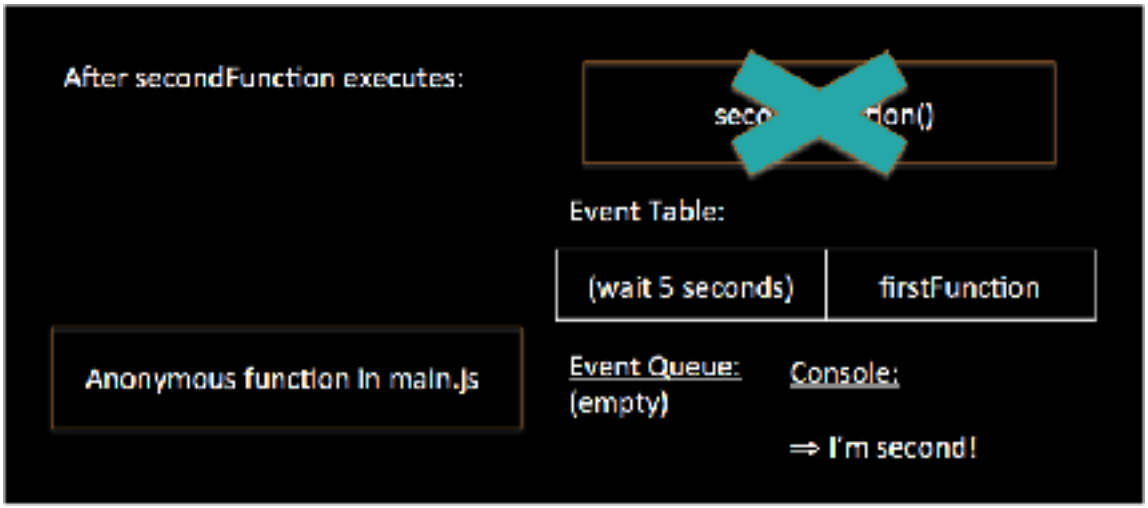


- Right after the setTimeout function is executed, something special happens here—the browser places setTimeout's callback function (in this case, firstFunction) into an **Event Table**. Think of the event table as a registration booth: the call stack tells the event table to register a particular function to be executed only when a specific event happens. And when the event does happen, the event table will simply move the function over to the **Event Queue**. The beauty of this event queue is that its simply a staging area for functions waiting to be invoked and moved over to the call stack.

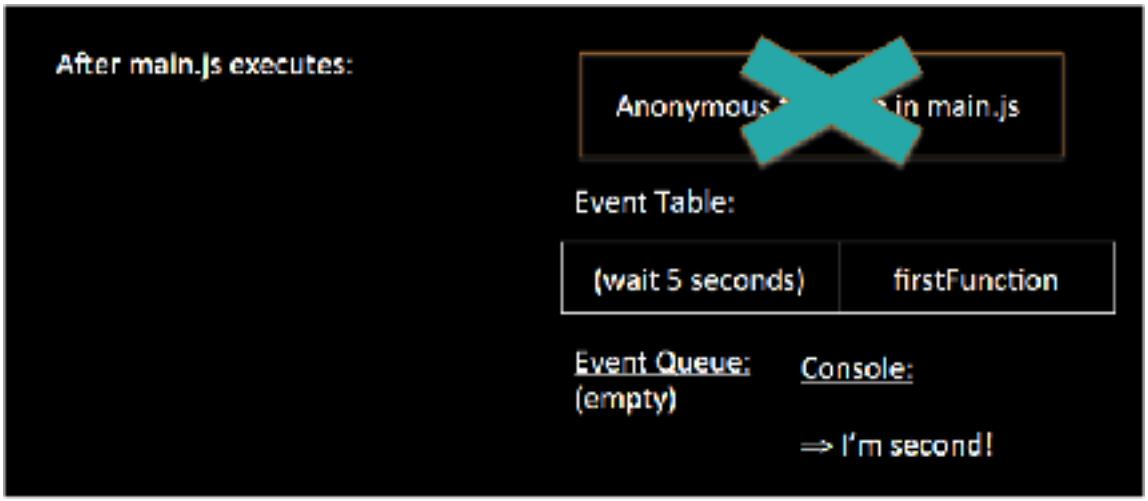
You might ask, "So when exactly can functions in the event queue move over to the call stack?" Well, the JavaScript engine follows a very simple rule: there's a process that constantly checks whether the call stack is empty, and whenever it's empty, it checks if the event queue has any functions waiting to be invoked. If it does, then the first function in the queue gets invoked and moved over into the call stack. If the event queue is empty, then this monitoring process just keeps on running indefinitely. And voila—what I just described is the infamous **Event Loop**!

- Now going back to our example, executing the setTimeout function moves the callback function (in this case, firstFunction) to the event table and registers it to with a time delay of 5 seconds:


- Here's another "Ah ha!" moment—notice that once the callback function is moved to the event table, nothing is blocked! The browser doesn't wait 5 seconds before it continues doing anything else—instead it proceeds to executing the next line in secondFunction, which is the console.log.

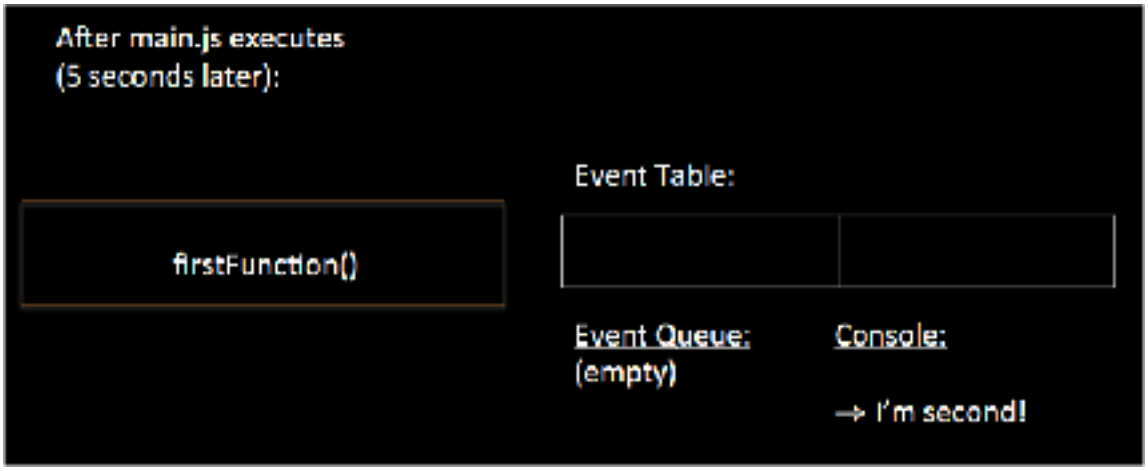


execution and main.js has now completed its execution as well.




- At some point, 5 seconds since the callback function was placed in the event table will have elapsed. And when that happens, the event table will move firstFunction into the event queue.


- And since the event loop constantly monitors whether the call stack is empty, it now notices that the call stack is indeed empty and invokes firstFunction which creates a new call stack.

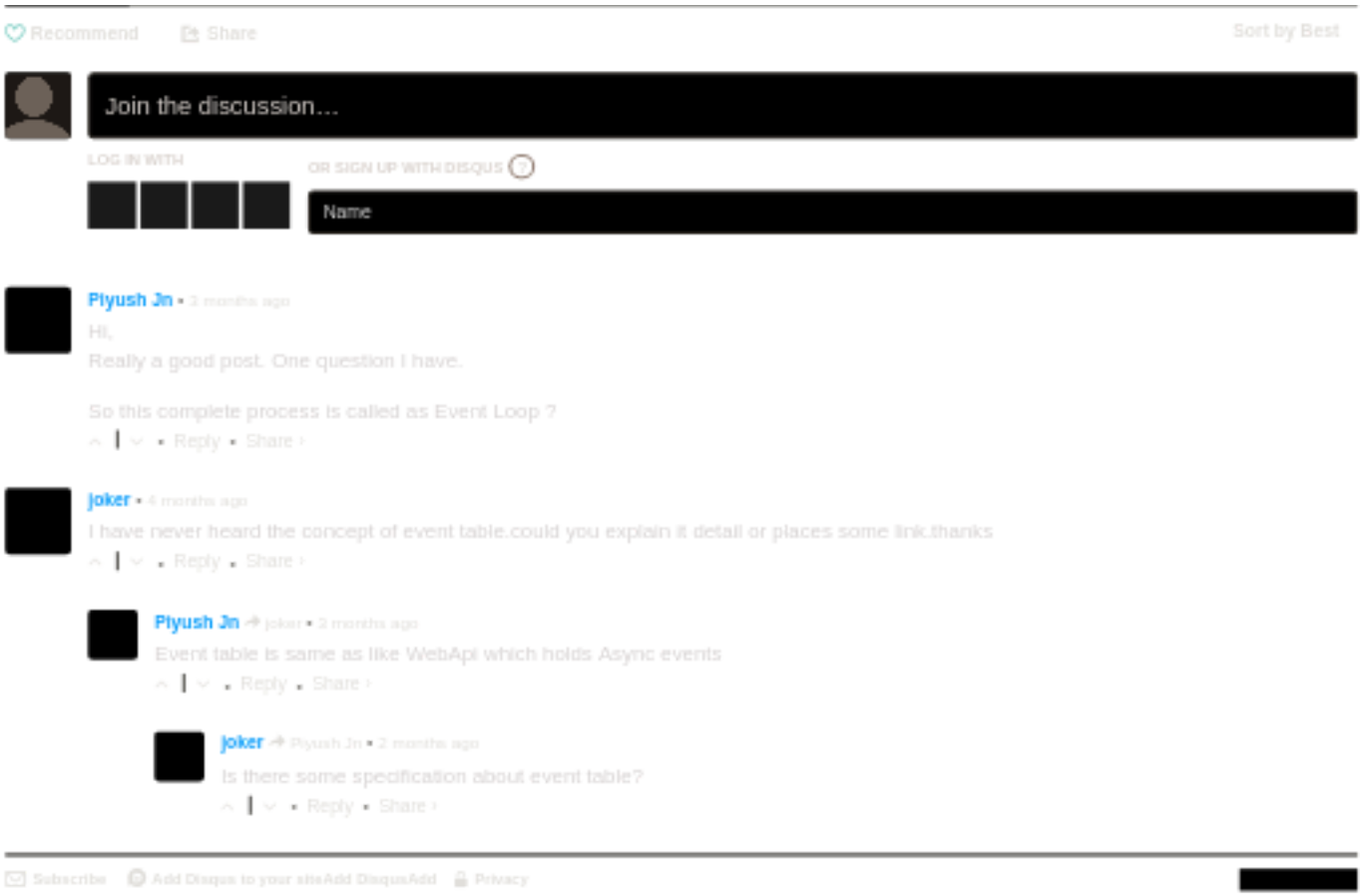



- Once firstFunction has completed its execution, we return to a state where there's nothing in the call stack, the event table doesn't have any events to listen for, and the event queue is empty.


# Conclusion

I'll be the first to admit that my explanation glosses over a TON of the actual implementation detail behind the JavaScript engine, event table, event queue, and event loop. However, for the vast majority of us, we simply need to have a solid foundational understanding of what's happening when JavaScript executes an asynchronous function. And I'm hopeful the explanation above will provide you with some clarity into what's happening under the hood that'll suffice for the majority of our work as web developers.

Disrupt by Altitude Labs




— Disrupt by Altitude Labs —

## Engineering

Infographic: How to create an app

4 things to consider when hiring a mobile app developer in Hong Kong

How to personalize your website in minutes with Metisa

See all 27 posts →

ENGINEERING

### Ghost blog images not working after move to subdirectory?

As search engines mostly regard subdomains as a separate domain in ranking algorithms, we moved our Ghost blog from a subdomain to a subdirectory to optimise our search engine performance. Ghost v0.4

JUSTIN YEK

ENGINEERING