# Node.JS Overview

In this article we will take a look at the browser’s JavaScript runtime environment. We will learn how Chrome’s V8 JS Engine parses code and see how the event loop helps that code run on a single thread, both synchronously, and asynchronously, sort of. Finally, we will look at a common example that more clearly explains how this process works.

## Let’s start from the beginning…

When you visit a website you do so within a web browser, like Chrome, Firefox, Edge, or Safari. Each browser has a **JS Runtime Environment**. In the environment are Web API’s that a developer can access to build a program.

*AJAX, the DOM tree, and other API’s, are not part of JavaScript, they are just objects with properties and methods, provided by the browser and made available in the browser’s JS Runtime Environment.*

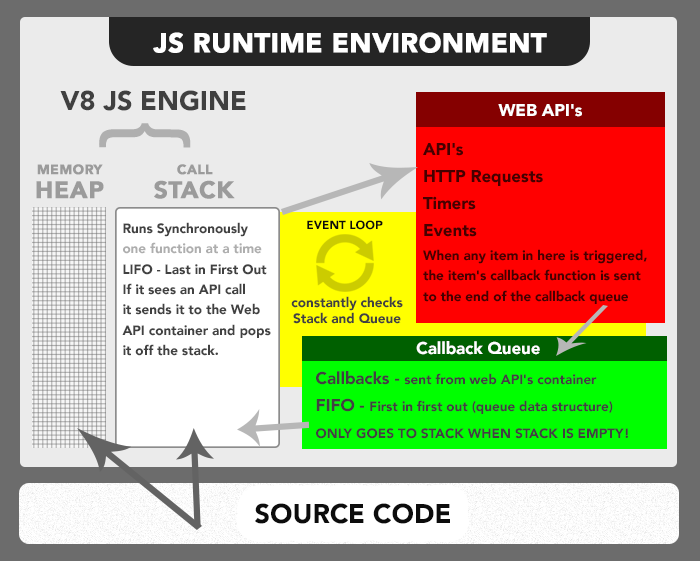
Also in the runtime environment is a **JavaScript Engine** that parses the code. Each browser has its own version of a JS engine. Chrome uses what it calls its **V8 JS Engine** and that is what we will analyse now.

## The V8 JS Engine

Once Chrome receives JavaScript code or scripts on the web page the V8 JS Engine starts parsing it. First, it will partially parse the code checking for syntax errors. If it finds none, it starts reading the code from top to bottom. Its ultimate goal is to turn the JavaScript code into machine code that the computer can understand. But before we understand what exactly it does with the code we must understand the environment in which it is parsed.

## The JavaScript Runtime Environment

Think of the JS runtime environment as a big container. Within the big container are other smaller containers. As the JS engine parses the code it starts putting pieces of it into different containers.



### The Heap

The first container in the environment, which is also part of the V8 JS Engine, is called the ‘memory heap.’ As the V8 JS Engine comes across variables and function declarations in the code it stores them in the **Heap**.

### The Stack

The second container in the environment is called the ‘call stack.’ It is also part of the V8 JS Engine. As the JS Engine comes across an actionable item, like a function call, it adds it to the **Stack**.

Once a function is added to the Stack the JS engine jumps right in and starts parsing its code, adding variables to the Heap, adding new function calls to the top of the stack, or sending itself to the third container where Web API calls go.

When a function returns a value, or is sent to the **Web API container**, it is popped off the stack and moves to the next function in the stack. If the JS Engine gets to the end of the function and no return value is explicitly written, the JS Engine returns undefined and pops off the function from the stack. This process of parsing a function and popping it off the stack is what they mean when they say JavaScript runs **synchronously**. It does one thing at a time on a single thread.

Note - the Stack is a data structure that runs LIFO — last in first out. No function other than the one at the top of the stack will ever be in focus, and the engine will not move to the next function unless the one above it is popped off.

### The Web API Container

The Web API calls that were sent to the Web API container from the Stack, like event listeners, HTTP/AJAX requests, or timing functions, sit there until an action is triggered. Either a ‘click’ happens, or the HTTP request finishes getting its data from its source, or a timer reaches its set time. In each instance, once an action is triggered, a ‘callback function’ is sent to the fourth and final container, the ‘callback queue.’

### The Callback Queue

The **Callback Queue** will store all the callback functions in the order in which they were added. It will ‘wait’ until the Stack is completely empty. When the Stack is empty it will send the callback function at the beginning of the queue to the Stack. When the Stack is clear again, it will send over its next callback function.

Note - the Queue is a data structure that runs FIFO — first in first out. Whereas the Stack uses a push and pop (add to end take from end), the Queue uses push and shift (add to end take from beginning).

### The Event Loop

The **Event Loop** can be thought of as a ‘thing’ inside the JavaScript runtime environment. Its job is to constantly look at the Stack and the Queue. If it sees the Stack is empty, it will notify the Queue to send over its next callback function. The Queue and the Stack might be empty for a period of time, but the event loop never stops checking both. At any time a callback function can be added to the Queue after an action is triggered from the Web API container.

This is what they mean when they say JavaScript can run **asynchronously**. It isn’t actually true, it just seems true. JavaScript can only ever execute one function at a time, whatever is at top of the stack, it is a synchronous language. But because the Web API container can forever add callbacks to the queue, and the queue can forever add those callbacks to the stack, we think of JavaScript as being asynchronous. This is really the great power of the language. Its ability to be synchronous, yet run in an asynchronous manner, like magic!

### Blocking vs Non-Blocking I/O

When we talk about blocking I/O, think of an infinite loop, where a function just keeps running. If the function never stops running then it will never get popped off the stack, thus ‘blocking’ the next function in the stack from ever running. Another possibility is running a function that has so much complex logic and calculations, it thus takes so much time to run that it ‘blocks’ the next function from running. These are things to be aware of when creating code, but these are more programming errors and poorly written code that block i/o, rather than it being the fault of the language.

One thing that can be a ‘blocking i/o’ is an HTTP request. Say you make a request to some external site data and you have to wait for that site’s network. The network may never respond and your code will ultimately be stuck. But JavaScript handles this in the runtime environment. It sends the HTTP request to the Web API and pops it off the stack so that the next function can run while the Web API waits for its data to return. If the HTTP request never gets its data back the rest of the program will continue running. This is what we mean when we say JS is a **Non-Blocking**language.

### A Typical Example

Many instructional videos and articles use an example similar to this to explain how the JS runtime environment works…

setTimeout(function(){  
 console.log('Hey, why am I last?');  
}, 0);

function sayHi(){  
 console.log('Hello');  
}

function sayBye(){  
 console.log('Goodbye');  
}

sayHi();

sayBye();

If you copy and paste this code into your console you will see it prints ‘Hello’, then ‘Goodbye’, then undefined, then ‘Hey, why am I last?’. Even though the setTimeout function is called first and is supposed to run in zero seconds, it outputs last. Go through each line and try to understand the process of the JS Engine as it parses this code. Try and think why the setTimeout() call prints to the console after sayHi() and sayBye().

When you’re done thinking it through, let’s take a look at the exact way in which the V8 JS Engine handles this code…

1. The JS Engine parses the entire script checking for syntax errors. It sees none so it starts more fully parsing from the top.

2. It sees a setTimeout function call and pushes it to the top of the Stack.

3. It jumps right into the function call, sees it is part of the Web API and thus sends it over to the Web API container and pops it off the Stack.

4. Because the timer is set to 0 seconds, the Web API container immediately pushes its anonymous function to the callback queue. The event loop checks the Stack to see if it’s empty, but it is not, because…

5–6. …as soon as the setTimeout function was sent over to the Web API container, the V8 JS Engine saw two function declarations, stored them in the Heap and then saw a function call of sayHi() and pushed it to the Stack.

7. That sayHi() function calls console.log(), which is pushed to the top of the stack.

8. The JS Engine jumps right into parsing that function. It returns a message to the console - ‘Hi’ - and is popped off.

9. The JS Engine moves back into the sayHi() function, gets to its closing bracket and pops it off the Stack.

10–12. As soon it’s popped off the Stack the next function call, sayBye(), is pushed to the Stack. It is parsed, calls console.log(), which gets pushed to the top of the Stack, it returns a message, is popped off the stack, then its original function is popped off the Stack.

13. The Event Loop sees that the Stack is finally empty. It lets the Queue know and the Queue pushes its anonymous function to the Stack.

14. The anonymous function is parsed and it calls console.log(), which is pushed on the stack.

15. The console.log() function is executed and is popped off the Stack.

16. The anonymous function is popped off the Stack and the program is finally finished.

PS - If you copy and pasted the code to the console you may have noticed an ‘undefined’ result. This is because none of the main functions are returning a value. Instead, they are calling console.log(), which is being executed by the parser and getting popped off, then the parser is getting to the end of the main function without seeing a returned result, so it returns ‘undefined’, then it pops the function off the stack.

**Now You Try!**

To visually see the above code in action, you should use one of the greatest tools ever created for understanding the JS engine and the event loop. You can find that [here](http://latentflip.com/loupe/?code=Ly8gZmlyc3QsIGNhbGwgdGltZW91dCBmdW5jdGlvbiwgc2V0IHRvIHplcm8gc2Vjb25kcwpzZXRUaW1lb3V0KGZ1bmN0aW9uKCl7CiAgY29uc29sZS5sb2coJ0hleSwgd2h5IGFtIEkgbGFzdD8nKTsKfSwgMCk7Ci8vIGRlY2xhcmUgdHdvIGZ1bmN0aW9ucwpmdW5jdGlvbiBzYXlIaSgpewogIGNvbnNvbGUubG9nKCdIZWxsbycpOwp9CmZ1bmN0aW9uIHNheUJ5ZSgpewogIGNvbnNvbGUubG9nKCdHb29kYnllJyk7Cn0KLy8gY2FsbCB0aGUgdHdvIGZ1bmN0aW9ucyBsYXN0CnNheUhpKCk7CnNheUJ5ZSgpOw%3D%3D!!!PGJ1dHRvbj5DbGljayBtZSE8L2J1dHRvbj4%3D). After you view the above example, reload the page and look at the provided default example. Watch the [video presentation](https://www.youtube.com/watch?v=8aGhZQkoFbQ) by Philip Roberts. Then try to explain this process to someone, anyone, out loud. Rubber ducky it if you have to. Then go back to [the loupe tool](http://latentflip.com/loupe/?code=Ly8gZmlyc3QsIGNhbGwgdGltZW91dCBmdW5jdGlvbiwgc2V0IHRvIHplcm8gc2Vjb25kcwpzZXRUaW1lb3V0KGZ1bmN0aW9uKCl7CiAgY29uc29sZS5sb2coJ0hleSwgd2h5IGFtIEkgbGFzdD8nKTsKfSwgMCk7Ci8vIGRlY2xhcmUgdHdvIGZ1bmN0aW9ucwpmdW5jdGlvbiBzYXlIaSgpewogIGNvbnNvbGUubG9nKCdIZWxsbycpOwp9CmZ1bmN0aW9uIHNheUJ5ZSgpewogIGNvbnNvbGUubG9nKCdHb29kYnllJyk7Cn0KLy8gY2FsbCB0aGUgdHdvIGZ1bmN0aW9ucyBsYXN0CnNheUhpKCk7CnNheUJ5ZSgpOw%3D%3D!!!PGJ1dHRvbj5DbGljayBtZSE8L2J1dHRvbj4%3D) and try writing your own examples. Predict what will happen, then run it to see if you were right. Repeat until you got this all intrinsically.

# Browser Environment vs Node.js Environment

It is important to note that the environment described in this article is provided by the browser. Node.js provides a completely different runtime environment, though it is also powered by Google Chrome’s V8 JS Engine. This means Node.js will not provide for you the DOM tree, AJAX, or other Web API’s. However, you can install your preferred packages into the Node environment as needed for the program you are creating.

**Other Considerations**

The browser does much more than just provide a JavaScript runtime environment. It is also processing HTML and CSS and rendering graphics on the screen. We will take a look at how the browser handles all that in another article.

# The Node Command

When Node.js is installed on a system, it provides a program called node, which is used to run JavaScript files. Say you have a file hello.js, containing this code:

var message = “Hello world”;

console.log(message)

You can then run node from the command line like this to execute the program:

node hello.js

The console.log method in Node does something similar to what it does in the browser. It prints out a piece of text. But in Node, the text will go to the process’ standard output stream, rather than to a browser’s JavaScript console.

If you run node without giving it a file, it provides you with a prompt at which you can type JavaScript code and immediately see the result.

The process variable, just like the console variable, is available globally in Node. It provides various ways to inspect and manipulate the current program. The exit method ends the process and can be given an exit status code, which tells the program that started node (in this case, the command-line shell) whether the program completed successfully (code zero) or encountered an error (any other code).

All the standard JavaScript global variables, such as Array, Math, and JSON, are also present in Node’s environment. Browser-related functionality, such as document and alert, is absent.

The global scope object, which is called window in the browser, has the more sensible name global in Node.