How does JavaScript and JavaScript engine work in the browser and node

http://latentflip.com/loupe/?code=ZnVuY3Rpb24gcHJpbnRIZWxsbygpIHsNCiAgICBjb25zb2xlLmxvZygnSGVsbG8gZnJvbSBiYXonKTsNCn0NCg0KZnVuY3Rpb24gYmF6KCkgew0KICAgIHNldFRpbWVvdXQocHJpbnRIZWxsbywgMzAwMCk7DQp9DQoNCmZ1bmN0aW9uIGJhcigpIHsNCiAgICBiYXooKTsNCn0NCg0KZnVuY3Rpb24gZm9vKCkgew0KICAgIGJhcigpOw0KfQ0KDQpmb28oKTs%3D!!!PGJ1dHRvbj5DbGljayBtZSE8L2J1dHRvbj4%3D

# JavaScript in a nutshell

JavaScript is an **interpreted** language. This means we do not have to compile the JavaScript source code before sending it to the browser. An interpreter can take the raw JavaScript code and run it for you.

JavaScript is also a dynamically typed language, unlike C and C++. This means variables declared using var can store any type of data type like int, string, boolean and also complex data types like object and array.

# JavaScript at runtime

JavaScript is a **single-threaded** language at runtime. That means the code execution will be done one piece at a time. Since code execution is done sequentially, any code that takes a long time to execute will block anything that needs to be executed after that.

To visualize, how JavaScript executes a program, we need to understand JavaScript runtime and different components that play a part in it. So lets write a simple JavaScript program to visualize this.

function baz() {

console.log( 'Hello from baz' );

}

function bar() {

baz();

}

function foo() {

bar();

}

foo();

Here we have a simple JavaScript program that has three functions, viz. foo, bar and baz. The function foo calls the function bar and then function bar calls the function baz which logs something to the console using the console.log function provided by the runtime.

When we run this program, first the function foo gets called and then the call chain begins until the console.log() is executed. Let’s visualize this using a diagram and inspect various components of the runtime.

Graphical user interface, application

Description automatically generated

Like any other programming language, JavaScript runtime has one stack and one heap storage. A **heap** is a free memory storage unit where you can store memory in random order. Data that is going to persist in for a considerable amount of time go inside the heap. Heap is managed by the JavaScript runtime and cleaned up by the garbage collector. I am not going to explain much more about the heap, you can read it [here](https://hashnode.com/post/does-javascript-use-stack-or-heap-for-memory-allocation-or-both-cj5jl90xl01nh1twuv8ug0bjk).

What we are interested in is **stack**. A stack is **LIFO** (last in, first out) data storage that stores the current function execution **context** of a program. In the above example, when our program is loaded into the memory, it starts execution from the first function call which is foo().

Hence, the first stack entry is foo(). Since foo function calls bar function, second stack entry is bar(). Since bar function calls baz function, third stack entry is baz(). And finally, baz function calls console.log, fourth stack entry is console.log('Hello from baz').

Until a function returns something (while the function is executing), it won’t be popped out from the stack. The stack will pop entries one by one as soon as that entry (function) returns some value, and it will continue pending function executions.

Graphical user interface

Description automatically generated with medium confidence

Each entry in the stack is called a **stack frame**. A stack frame contains the information of the function call such as arguments of the function call, locals of the function, return address (where the return value will be consumed), and other information of the function.

Since JavaScript is single-threaded, it has only one stack and one heap per process. Hence, if any other program wants to execute something, it has to wait until the previous program is completely executed. This thread is commonly known as **main thread** or **main execution thread**.

So let’s think of one scenario. What if a browser sends an HTTP request to load some data over the network or to load an image to display on the web page. Will the browser freeze until that request is resolved? If it does, then it’s very bad for user experience.

A browser comes with a JavaScript engine that is responsible to execute any JavaScript contained inside a web application (web page). For example, Google Chrome uses [**V8**](https://v8.dev/) JavaScript engine.

But guess what, the browser uses more than just the JavaScript engine. This is what browser under the hood looks like.

Diagram

Description automatically generated

Looks really complex but it is very if you understand one piece at a time and they work together in harmony. JavaScript runtime actually consists of 2 more components viz. **event loop** and **callback queue**. The callback queue is also called a **message queue** or **task queue**.

Apart from JavaScript engine, browser contains different applications which can do a variety of things like send HTTP requests, listen to DOM events, delay execution using setTimeout or setInterval, caching, database storage, and much more. These features of the browser help us create rich web applications and better user experience.

But think about this, if the browser had to use the same JavaScript thread for the execution of these tasks, then user experience would be terrible. For example, if the browser had to use the same JavaScript thread to perform a task when the HTTP network response is received, then the web page would be irresponsive for seconds or even for minutes.

Hence browser implements their own logic to perform these operations such as sending HTTP requests and listening to their responses. These operations do not block the JavaScript main execution thread since they are spawned on different threads managed by the browser and JavaScript has no idea of it.

A browser may use a low-level language like C or C++ to implement these features for performance benefits and give us the clean JavaScript API to execute these operations from the JavaScript. For example, [fetch](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API) API is provided by the browser to send HTTP requests. These APIs are known as **Web APIs** since they are not part of the JavaScript specifications.

These Web APIs are **asynchronous**. That means you can instruct these APIs to do something in the background and return data once done, meanwhile we can continue further execution of JavaScript code. While instructing these APIs to do something in the background, we have to provide a **callback function**. Responsibility of a callback function is to execute some JavaScript code in the main Javascript thread once Web API is done with its work. Let’s understand how all pieces work together.

So when you call a function, it gets pushed to the stack. If that function contains a Web API call, JavaScript will delegate control of it to the Web API with a callback function and move to the next lines until the function returns something. Now the callback function is with the Web API which is performing its operation on a separate thread, separate from the main thread.

Once the function hits the return statement, that function is popped from the stack and move to the next stack entry. Meanwhile, Web API is doing its job in the background and remembers what callback function is associated with that job. Once the job is done, Web API binds the result of that job to the callback function and publishes a message to the **message queue** (AKA **callback queue**) with that callback function.

The only job of the **event loop** is to look at callback queue and once there is something pending in callback queue, push that callback to the **stack**. The event loop pushes one callback function at a time, to the stack, **once the stack is empty**. Later, the stack will execute the callback function.

Let’s see how everything works step by step using setTimeout Web API. The setTimeout Web API is mainly used to execute something after a few seconds (any time period). This execution happens once all the code in the program is done executing (when the stack is empty). The syntax for setTimeout function is as below.

setTimeout(callbackFunction, timeInMilliseconds);

The callbackFunction is a callback function which will execute after timeInMilliseconds. Let’s modify our earlier program and use this API.

function printHello() {

console.log('Hello from baz');

}

function baz() {

setTimeout(printHello, 3000);

}

function bar() {

baz();

}

function foo() {

bar();

}

foo();

The only modification done to the program is, we delayed printHello function execution by 3 seconds. In this case, the stack will keep building up like foo() => bar() => baz(). Once baz starts executing and hits setTimeout API call, JavaScript will pass the callback function to the Web API and move to the next line.

Since there is no next line, function returns and the stack will pop baz, then bar and then foo function calls. Meanwhile, Web API is waiting for 3 seconds to pass. Once 3 seconds are passed, it will push this callback to callback queue and since the stack is empty, the event loop will put this callback back on the stack where the execution of this callback will happen.

Graphical user interface, application

Description automatically generated

The event loop and callback queue are the pieces of the same puzzle. They are not part of the Javascript engine, rather they sit outside JavaScript engine and normally provided by the runtime such as a web browser or Node.js. The event loop uses JavaScript engine’s APIs to communicate with it and provide callback functions to execute.

# Even loop inside Node.js

When it comes to **Node.js**, it has to do more because the Node promises more. In the case of a browser, we are limited to what we can do in the background. But in node, we can pretty much do most of the things in the background, even it is a simple JavaScript program. But, how does that work?

Node.js uses **Google’s V8 engine** to provide JavaScript runtime and employes its own event loop using the **[libuv](https://github.com/libuv/libuv" \t "_blank)** library (written in c). Node follows the same callback approach like Web APIs and works in a similar fashion as the browser.

Diagram, schematic

Description automatically generated

If you compare the browser diagram with the above node diagram, you can see the similarities. The entire right section looks like Web API but it also contains event queue (callback queue/message queue) and the event loop. But V8, event queue, and event loop runs on the single thread while worker threads are responsible to provide asynchronous I/O operation. That’s why Node.js is said to have as **non-blocking event-driven asynchronous I/O architecture**.