**EVENT LOOP**

JavaScript is **single-threaded**: only one task can run at a time. Usually that’s no big deal, but now imagine you’re running a task which takes 30 seconds. During that task we’re waiting for 30 seconds before anything else can happen. (JavaScript runs on the browser’s main thread by default, so the entire UI is stuck) It’s 2019, no one wants a slow, unresponsive website.

Luckily, the browser gives us some features that the JavaScript engine itself doesn’t provide: a Web API. This includes the DOM API, setTimeout, HTTP requests, and so on. This can help us create some async, non-blocking behavior.

When we invoke a function, it gets added to something called the call stack. The call stack is part of the JS engine, this isn’t browser specific. It’s a stack, meaning that it’s first in, last out. When a function returns a value, it gets popped off the stack.

Graphical user interface, application

Description automatically generated

Let’s look some example;

const foo = () => console.log("First");

const bar = () => setTimeout(() => console.log("Second"), 500);

const baz = () => console.log("Third");

bar();

foo();

baz();

1. We invoke bar. bar returns a setTimeout function.
2. The callback we passed to setTimeout gets added to the Web API, the setTimeout function and bar get popped off the callstack.
3. The timer runs, in the meantime foo gets invoked and logs First. foo returns (undefined),baz gets invoked, and the callback gets added to the queue.
4. baz logs Third. The event loop sees the callstack is empty after baz returned, after which the callback gets added to the call stack.
5. The callback logs Second.