**API**

Application Programming Interfaces (APIs) are constructs made available in programming languages to allow developers to create complex functionality more easily. They abstract more complex code away from you, providing some easier syntax to use in its place.

* **APIs for manipulating documents** loaded into the browser. The most obvious example is the [DOM (Document Object Model) API](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model), which allows you to manipulate HTML and CSS — creating, removing and changing HTML, dynamically applying new styles to your page, etc. Every time you see a popup window appear on a page or some new content displayed, for example, that's the DOM in action.

.

* **APIs that fetch data from the server** to update small sections of a webpage on their own are very commonly used. This seemingly small detail has had a huge impact on the performance and behavior of sites — if you just need to update a stock listing or list of available new stories, doing it instantly without having to reload the whole entire page from the server can make the site or app feel much more responsive and "snappy". The main API used for this is the [Fetch API](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API), although older code might still use the [XMLHttpRequest](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest) API. You may also come across the term **Ajax**, which describes this technique
  + - * [**How do APIs work?**](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Introduction#how_do_apis_work)

Your code interacts with APIs using one or more [JavaScript objects](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects), which serve as containers for the data the API uses (contained in object properties), and the functionality the API makes available (contained in object methods).

# The event loop

JavaScript has a runtime model based on an **event loop**, which is responsible for executing the code, collecting and processing events, and executing queued sub-tasks. This model is quite different from models in other languages like C and Java.

The **event loop** got its name because of how it's usually implemented, which usually resembles:

while (queue.waitForMessage()) {

queue.processNextMessage();

}

## How do Event loops work?

1. **Call Stack:**
   * JavaScript uses a call stack to keep track of the currently executing function (where the program is in its execution).
2. **Callback Queue:**
   * Asynchronous operations, such as I/O operations or timers, are handled by the browser or Node.js runtime. When these operations are complete, corresponding functions (callbacks) are placed in the callback queue.
3. **Event Loop:**
   * The event loop continuously checks the call stack and the callback queue. If the call stack is empty, it takes the first function from the callback queue and pushes it onto the call stack for execution.
4. **Execution:**
   * The function on top of the call stack is executed. If this function contains asynchronous code, it might initiate further asynchronous operations.
5. **Callback Execution:**
   * When an asynchronous operation is complete, its callback is placed in the callback queue.
6. **Repeat:**
   * The event loop continues this process, ensuring that the call stack is always empty before taking the next function from the callback queue.

## ****Memory allocation in JavaScript****

### ****Heap memory****

This is where JavaScript stores objects and data that are dynamically allocated. Memory in the heap is allocated in an unordered way, which means data can be stored at any available location.

### ****Stack memory****

Stack memory is used to keep track of function calls. It works in a last-in, first-out (LIFO) manner, meaning the last function that gets pushed onto the stack is the first one to be popped off when it’s done.

### ****Function call stack****

The call stack specifically tracks which functions are being executed at any moment. If a function calls another function, the new function is added to the

top of the stack. Once a function finishes, it’s removed from the stack, and execution resumes where it left off.

## ****Event loop****

An event loop is something that pulls stuff out of the queue and places it onto the function execution stack whenever the function stack becomes empty.

The event loop is the secret by which JavaScript gives us an illusion of being multithreaded even though it is single-threaded. The below illusion demonstrates the functioning of the event loop well: