**Javascript APIs**

JavaScript provides several powerful APIs that extend its functionality and enable developers to create richer and more interactive web applications. These APIs are part of the browser environment and allow JavaScript to interact with various system features such as file handling, clipboard management, and running tasks in the background.

This documentation explores some of the commonly used APIs that enhance JavaScript's capabilities:

1. **Clipboard API**
2. **File API**
3. **Web Workers API**

**Clipboard API**

The Clipboard API allows web applications to interact with the user's clipboard (cut, copy, and paste operations). It provides more control over clipboard actions compared to the traditional document.execCommand() method and enables programmatic access to copy and paste operations.

**Features:**

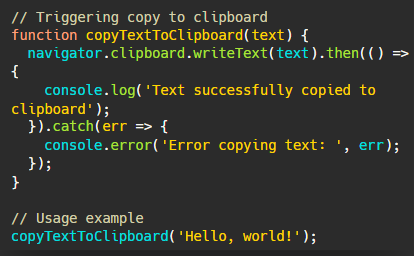
* Supports copying and pasting text and images.
* Offers a more secure and modern alternative to document.execCommand().
* Allows access to clipboard content using JavaScript (with user permission).

**Use Cases**

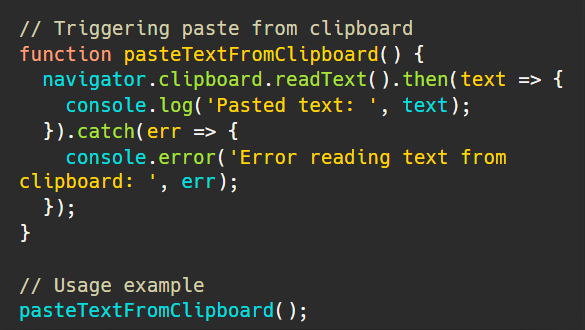
1. **Copying text to clipboard:** Custom copy-to-clipboard buttons.
2. **Clipboard management:** Implementing features like copying code snippets, URLs, or other content.

**Example**

**Copying Text to Clipboard**



**Pasting Text from Clipboard**



**Security Considerations**

* Clipboard operations require user interaction (e.g., clicking a button) due to security and privacy concerns.
* Access to the clipboard might be restricted in some browsers, especially when not triggered by a user action.

**File API**

The File API provides the ability to read and manipulate files on the client-side. It enables web applications to interact with files selected by the user, such as images, documents, and more. This API is essential for handling file uploads, allowing the application to preview, read, and validate files before they are sent to a server.

**Features:**

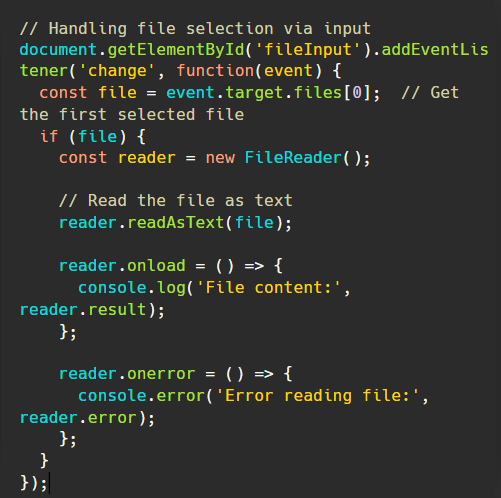
* Allows reading file metadata (e.g., name, size, type).
* Can read file contents using FileReader, Blob, and File objects.
* Supports drag-and-drop file uploads and file selection via <input type="file">.

**Use Cases**

1. **File upload functionality:** Handle user uploads of images, documents, or other files.
2. **File preview:** Display a preview of selected images or other files before uploading them.
3. **File validation:** Ensure uploaded files meet the required formats and sizes.

**Example**

**Reading a File Using FileReader**



**File Upload Example with Form**



**Security Considerations**

* The File API allows reading files from the user's local machine, but access is restricted to files selected by the user through an <input> element or drag-and-drop.
* Files should not be stored or transmitted without proper validation to ensure that the content is safe (e.g., checking for malicious scripts in uploaded files).

**Web Workers API**

The Web Workers API enables JavaScript to run scripts in the background, in separate threads, without blocking the main UI thread. This allows long-running tasks (e.g., data processing, heavy computations) to be executed without freezing or slowing down the user interface.

**Features:**

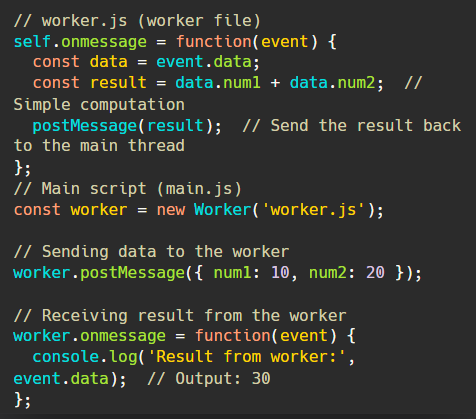
* Offloads computations to background threads.
* Can be used to run tasks asynchronously while keeping the UI responsive.
* Supports communication between the main thread and workers using messages.
* Web workers run in a separate global context, so they don’t have access to the DOM or the window object.

**Use Cases**

1. **Complex data processing:** Run large calculations without blocking the UI.
2. **Background tasks:** Perform background tasks like data fetching, video processing, or machine learning model inference.
3. **Multithreading:** Take advantage of multi-core processors to speed up computations.

**Example**

**Creating a Web Worker**



**Terminating a Worker**



**Security Considerations**

* Web Workers run in a separate global context and cannot access the DOM, window object, or most web APIs. This is a security feature to isolate the worker from the main thread.
* Web Workers do not have access to certain browser features (e.g., localStorage), but they can communicate with the main thread via the postMessage() API.

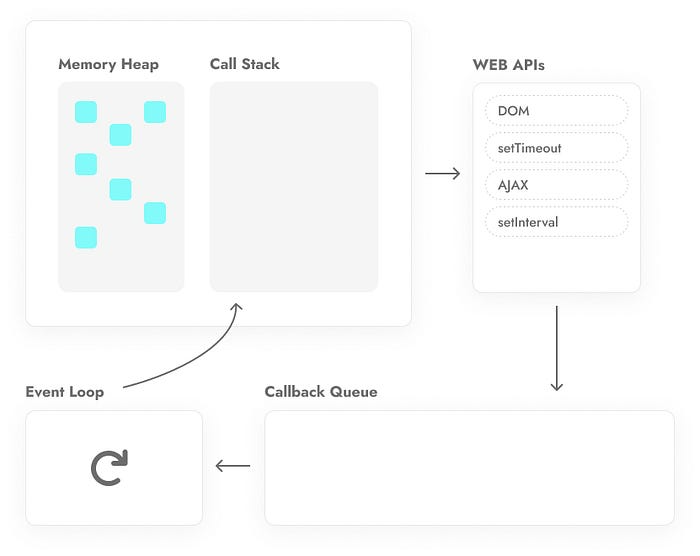
**Further Reading and Resources:**

* [**https://developer.mozilla.org/en-US/docs/Web/API/Clipboard\_API**](https://developer.mozilla.org/en-US/docs/Web/API/Clipboard_API)
* [**https://developer.mozilla.org/en-US/docs/Web/API/File\_API**](https://developer.mozilla.org/en-US/docs/Web/API/File_API)
* [**https://developer.mozilla.org/en-US/docs/Web/API/Web\_Workers\_API**](https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API)

**JavaScript Event Loop and Call Stack**

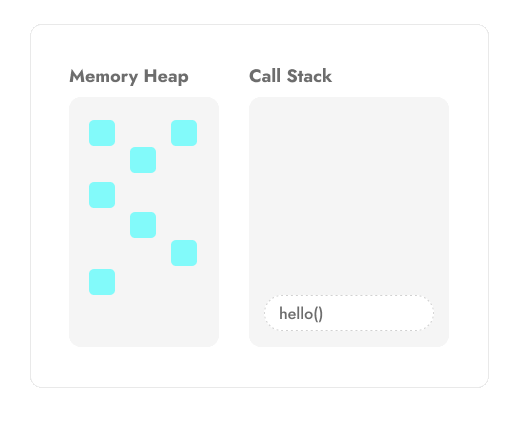
JavaScript works on a **single** thread, so it can only do one thing at a time using one **call stack**. JavaScript engines (like Chrome V8) use this stack to manage what is running. The stack works like a “**last in, first out**” (LIFO) system, meaning the last thing added is the first to be handled.

* **Memory Heap**: This is where the variables, objects, and functions we create are stored.
* **Web APIs**: We said JavaScript is**single-threaded**, but if you wanna do a few things at once, it’s possible thanks to **Web APIs** provided by browsers. So, when we wanna do something **async**, like an API request, it’s handled in its own thread and moved to the**Callback Queue**. For example, if we want to log “**hello**” to the console after **1** second using setTimeout, first, the setTimeout() function is added to the stack. Since there's a **1-second delay**, the callback function gets moved to **Web APIs** and setTimeout is removed from the stack. So, the stack is **empty** and the code isn't blocked. After 1 second, the callback function from Web APIs is added to the **Callback Queue**, and if the stack is empty, the **Event Loop** will remove this callback into the stack.
* **Callback Queue**: When the async task in Web APIs is done, the callback lands here.
* **Event Loop**: As soon as the Call Stack is empty, the Event Loop takes the first call from the Callback Queue and throws it into the stack.



function foo(){  
 console.log("foo");  
}  
function bar(){  
 console.log("bar");  
}  
function hello(){  
 foo();  
 bar();  
 console.log ("hello");  
}  
hello();

In the code above, there are three functions. The first one, foo, will log "**foo**" to the console. The second one, bar, will log "**bar**" to the console. The third one, hello, calls the foo and bar functions and then logs "**hello**" to the console. Finally, I called the hello() function.



The code runs from top to bottom.

* First, the hello() function goes into the Call Stack. Inside that, the foo() function gets called, so it’s **added** to the stack too. The foo function needs to log "**foo"** to the console, so console.log is added to the stack. Since the stack works on a "last in, first out" (**LIFO**) basis, the console.log runs first and then gets **removed**. After that, foo is done, so it’s also taken off the stack.
* The hello function is still in the stack 'cause it’s not finished yet. Now, the bar function gets added to the Call Stack. It needs to log "bar" to the console, so it goes into the stack, runs, and then gets removed, just like foo.
* Finally, the hello function needs to log "hello" to the console. It gets added to the stack, runs, and then gets removed. Since the hello function has nothing left to do, it’s also removed from the stack.

**Async**

Let’s check out an example with setTimeout.

console.log("hello");  
function world(){  
 console.log("world")  
}  
setTimeout(function cbFunction(){  
 console.log("Callback Function")  
}, 1000);  
world();

* In the code above, the first line gets **added** to the Call Stack and logs “**hello**” to the console, then it’s **removed** from the stack.
* Next, we have a **function** named world that logs "**world**" to the console. This function is called after setTimeout.
* In this case, the setTimeout function is **added** to the stack. It checks the **1000 millisecond** delay and moves the callback function, cbFunction(), to Web APIs while removing setTimeout from the stack.
* Then, the world() function and the console log are **added** to the Call Stack one by one, executed, and **removed** from the stack. During this time, **1000 millisecond** pass, and cbFunction() is **added** to the **Callback Queue**.
* The Event Loop checks if the Call Stack is **empty**, and if it is, it adds the callback function to the stack, logs “**Callback Function”** to the console, and removes it from the stack. Since this function has no other tasks, it’s also removed, leaving the Call Stack empty again.

**Order of Execution in Asynchronous Functions in the Event Loop**

**1. Synchronous Code Execution:**

All synchronous code is executed first. This includes function calls, variable assignments, and other straightforward JavaScript statements. This code runs in the call stack.

**2. Microtasks Execution:**

Once the synchronous code has finished executing, the event loop checks the microtask queue. All tasks in the microtask queue are executed before moving on to the macro task queue. This is why Promise callbacks are executed before setTimeout callbacks.

Examples include:

* **Promise callbacks (.then(), .catch(), .finally())**
* **MutationObserver callbacks**
* **process.nextTick()** (Node.js only)

**3. Macro Task Execution:**

After the microtasks are completed, the event loop checks the macro task queue and executes the first task. Once a task from the macro task queue is executed, the event loop goes back to check the microtask queue again.

Examples include:

* **setTimeout**
* **setInterval**
* **I/O operations**
* **UI rendering events**

**4. Repeating the Process:**

The event loop continues to alternate between executing tasks from the microtask queue and the macro task queue, ensuring that the microtask queue is always checked after executing any macro task.

**Example 1 : Combining Multiple Microtasks and Macro Tasks**

console.log('Script start');  
  
Promise.resolve().then(() => {  
 console.log('Promise 1');  
}).then(() => {  
 console.log('Promise 2');  
});  
  
console.log('Script end');

**Execution Flow:**

1. **Synchronous code:** The code executes console.log('Script start'), printing "Script start".
2. **Promise resolved:** The promise is resolved immediately, and the .then() callbacks are scheduled in the microtask queue.
3. **Synchronous code:** The code executes console.log('Script end'), printing "Script end".
4. **Microtasks execution:**

* The first .then() callback is executed, printing "Promise 1".
* The second .then() callback is executed, printing "Promise 2".

**Final Output:**

Script start  
Script end  
Promise 1  
Promise 2

**Example 2: Combining Multiple Microtasks and Macro Tasks**

console.log('Start');  
  
setTimeout(() => {  
 console.log('Timeout 1');  
}, 0);  
  
Promise.resolve().then(() => {  
 console.log('Promise 1');  
}).then(() => {  
 console.log('Promise 2');  
});  
  
setTimeout(() => {  
 console.log('Timeout 2');  
}, 0);  
  
console.log('End');

**Execution Flow:**

1. **Synchronous code:** “Start” is logged first.
2. **First setTimeout:** The first setTimeout callback is scheduled in the macro task queue.
3. **Promise resolved:** The promise is resolved, and its .then() callbacks are scheduled in the microtask queue.
4. **Second setTimeout:** The second setTimeout callback is scheduled in the macro task queue.
5. **Synchronous code:** “End” is logged.
6. **Microtasks executed:**

* “Promise 1” is logged.
* “Promise 2” is logged.

7. **Macro tasks executed:**

* “Timeout 1” is logged.
* “Timeout 2” is logged.

**Final Output:**

Start  
End  
Promise 1  
Promise 2  
Timeout 1  
Timeout 2

**Example 3 : Microtasks Inside Macro Tasks**

console.log('Start');  
  
setTimeout(() => {  
 console.log('Timeout 1');  
 Promise.resolve().then(() => {  
 console.log('Promise inside Timeout 1');  
 });  
}, 0);  
  
Promise.resolve().then(() => {  
 console.log('Promise 1');  
});  
  
console.log('End');

**Execution Flow:**

1. **Synchronous code:** “Start” is logged.
2. **First setTimeout:** The setTimeout callback is scheduled in the macro task queue.
3. **Promise resolved:** The promise is resolved, and its .then() callback is scheduled in the microtask queue.
4. **Synchronous code:** “End” is logged.
5. **Microtasks executed:**

* “Promise 1” is logged.

**6. Macro tasks executed:**

* “Timeout 1” is logged.
* **New microtask added:** The promise inside the setTimeout callback schedules a new microtask.
* **New microtask executed:** “Promise inside Timeout 1” is logged.

**Final Output:**

Start  
End  
Promise 1  
Timeout 1  
Promise inside Timeout 1

**Debouncing & Throttling**

**Understanding Debouncing:**

Debouncing is a programming practice used to ensure that time-consuming tasks do not fire so often, which can cause performance issues. In essence, it limits the execution of a function until a certain amount of idle time has passed without it being triggered again.

**Use Case: Search Input Field**

Consider a search input field that fetches suggestions from a server as the user types. Without debouncing, every keystroke would send a request, potentially leading to hundreds of requests per minute. Debouncing allows us to delay the function call until the user has stopped typing for a predefined time.

**Code Example:**

function debounce(func, delay) {  
 let debounceTimer;  
 return function() {  
 const context = this;  
 const args = arguments;  
 clearTimeout(debounceTimer);  
 debounceTimer = setTimeout(() => func.apply(context, args), delay);  
 };  
}

// Usage  
const fetchSuggestions = debounce(() => {  
 // Fetch suggestions from the server  
}, 250);

**Understanding Throttling:**

Throttling, on the other hand, ensures that a function is executed at most once every specified period. This is particularly useful for managing events that occur more frequently than you need to handle them.

**Use Case: Scroll Event Listener**

An example use case is attaching a listener to the scroll event of a webpage. Since the scroll event can fire dozens of times per second, throttling can be used to limit the number of times your callback function executes, improving performance.

**Code Example:**

function throttle(func, limit) {  
 let inThrottle;  
 return function() {  
 const args = arguments;  
 const context = this;  
 if (!inThrottle) {  
 func.apply(context, args);  
 inThrottle = true;  
 setTimeout(() => (inThrottle = false), limit);  
 }  
 };  
}

// Usage  
window.addEventListener('scroll', throttle(() => {  
 // Handle the scroll event  
}, 1000));

**Debouncing vs Throttling: Key Differences**

| **Feature** | **Throttling** | **Debouncing** |
| --- | --- | --- |
| Execution Control | Ensures function runs at most once per interval | Delays function execution until event stops |
| Best Use Cases | Scroll events, API calls, keypress handling | Search input, auto-save, form validation |
| Frequency of Execution | Runs at regular intervals | Runs only once after a delay |

**Implementing Debouncing and Throttling in JavaScript**

Both techniques can be implemented using vanilla JavaScript, as shown in the examples above. Additionally, libraries such as Lodash offer \_.debounce and \_.throttle functions, providing a more robust and cross-browser compatible solution.

**Conclusion**

Debouncing and throttling are powerful techniques for optimizing JavaScript applications, preventing unnecessary code executions, and improving user experience. By understanding their differences and use cases, developers can choose the right approach for their specific needs.