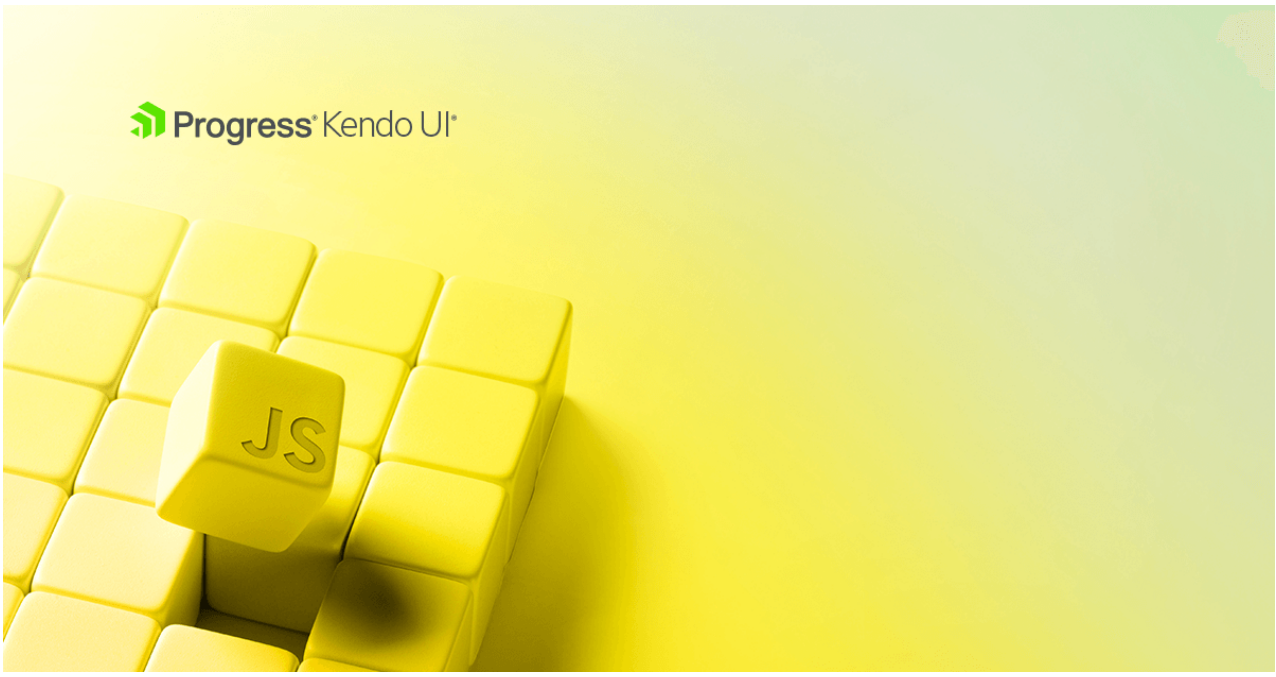
Debouncing and Throttling in JavaScript

telerik.com/blogs/debouncing-and-throttling-in-javascript

Rupesh Mishra August 22, 2019



This article talks about two important techniques, Debouncing and Throttling, to enhance your website performance. Learn about both concepts with the help of real-life examples and see how you can implement them yourself.

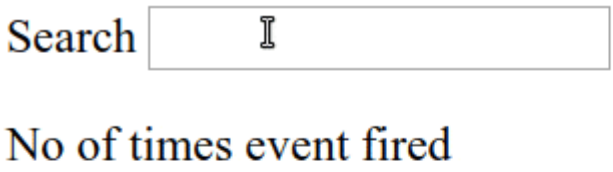
In this article, we will discuss debouncing and throttling techniques that can not only enhance the performance of your website, but also prevent unnecessary API calls and load on the server.

Debouncing and throttling techniques are used to limit the number of times a function can execute. Generally, how many times or when a function will be executed is decided by the developer. But in some cases, developers give this ability to the users. Now, it is up to the user to decide when and how many times to execute that function.

For instance, functions attached to events like `button click`, `mouse move`, and `window resize` allow the user to decide when to execute them and how many times to do so. At times, users may perform these actions more frequently than it is required. This may not be good for the performance of the website, especially if the functions attached to these events perform some heavy computation. In such cases, where users have control over function execution, developers have to design some techniques to restrict the number of times users can execute the function.

Let's consider an example of a search bar on a website. Every time you type something in the search bar, it makes an API call to fetch the data from the server on the basis of the letters typed in the search bar.

Let’s have a look at the code. For simplicity, we have not passed the parameter (that is, the word typed by the user in the search bar) to the `makeAPICall` function.



No of times event fired

Here’s, the HTML file for the search bar:

searchbar.html

```
<html>
    <body>
        <label>Search</label>
        <!-- Renders an HTML input box -->
        <input  type="text" id="search-box" />

        <p>No of times event fired</p>
        <p id='show-api-call-count'></p>
    </body>

    <script src="issueWithoutDebounceThrottling.js"></script>
</html>
```

Here’s the JavaScript file for search bar:

issueWithoutDebounceThrottling.js

```
var  searchBoxDom  =  document.getElementById('search-box');

function  makeAPICall() {
        // This represents a very heavy metho which takes a lot of time to execute
}

// Add "input" event listener on the text box or search bar
searchBoxDom.addEventListener('input', function () {
        var  showApiCallCountDom  =  document.getElementById('show-api-call-
count');
        var  apiCallCount  =  showApiCallCountDom.innerHTML  ||  0;

        // A very heavy method which takes a lot of time to execute
        makeAPICall()

        apiCallCount  =  parseInt(apiCallCount) +  1;
        // Updates the number of times makeAPICall method is called
        showApiCallCountDom.innerHTML  =  apiCallCount;
})
```

The above code renders an HTML page with a textbox. On the textbox, we have added an `oninput` event listener that calls an anonymous function. The anonymous function is called each time the user types in the textbox.

The anonymous function displays the number of times the `oninput` event has been fired by the user while typing. Inside the anonymous function, we are calling the `makeAPICall` function, which performs some heavy computation. The `makeAPICall` function calls an API that fetches data from the database and does some processing on that data and then returns the response.

Suppose the `makeAPICall` function takes 500 milliseconds to get data from the API. Now, if the user can type 1 letter per 100 milliseconds, then the `makeAPICall` function will be called 5 times in 500 milliseconds. Thus even before the `makeAPICall` has completed its task and returned the response, we are making 4 more API calls, which will put extra load on the server.

Moreover, these API calls are redundant! We could have fetched the desired data in just one API call after the user had completed typing. But, how can we decide if the user has stopped typing? In such a case, we could have assumed that if the user doesn't type for 200 milliseconds then he is done with typing. We will call the `makeAPICall` function only if the user doesn't type for 200 milliseconds. So basically, in this case, we have limited the number of times our `makeAPICall` function was called.

This technique of limiting the number of times the user can call a function attached to an event listener is debouncing and throttling. Debouncing and throttling also prevents the server from being bombarded by the API calls.

Let's try to understand both these concepts with the help of a real-life example:

Throttling

Imagine yourself as a 7-year-old toddler who loves to eat chocolate cake! Today your mom has made one, but it's not for you, it's for the guests! You, being spunky, keep on asking her for the cake. Finally, she gives you the cake. But, you keep on asking her for more. Annoyed, she agrees to give you more cake with a condition that you can have the cake only after an hour. Still, you keep on asking her for the cake, but now she ignores you. Finally, after an interval of one hour, you get more cake. If you ask for more, you will get it only after an hour, no matter how many times you ask her.

This is what throttling is!

> Throttling is a technique in which, no matter how many times the user fires the event, the attached function will be executed only once in a given time interval.

For instance, when a user clicks on a button, a `helloWorld` function is executed which prints `Hello, world` on the console. Now, when throttling is applied with 1000 milliseconds to this `helloWorld` function, no matter how many times the user clicks on the button, `Hello, world` will be printed only once in 1000 milliseconds. Throttling ensures that the function executes at a regular interval.

We're going to implement throttling later in the article.

Debouncing

Consider the same cake example. This time you kept on asking your mom for the cake so many times that she got annoyed and told you that she will give you the cake only if you remain silent for one hour. This means you won't get the cake if you keep on asking her continuously - you will only get it one hour after last time you ask, once you stop asking for the cake. This is debouncing.

> In the debouncing technique, no matter how many times the user fires the event, the attached function will be executed only after the specified time once the user stops firing the event.

For instance, suppose a user is clicking a button 5 times in 100 milliseconds. Debouncing will not let any of these clicks execute the attached function. Once the user has stopped clicking, if debouncing time is 100 milliseconds, the attached function will be executed after 100 milliseconds. Thus, to a naked eye, debouncing behaves like grouping multiple events into one single event.

When You'll Need Them

Debouncing and throttling are recommended to use on events that a user can fire more often than you need them to.

Examples include *window resizing and scrolling*. The main difference between throttling and debouncing is that throttling executes the function at a regular interval, while debouncing executes the function only after some cooling period.

Debouncing and throttling are not something provided by JavaScript itself. They're just concepts we can implement using the `setTimeout` web API. Some libraries like `underscore.js` and `loadash` provide these methods out of the box.

Both throttling and debouncing can be implemented with the help of the `setTimeout` function. So, let's try to understand the `setTimeout` function.

setTimeout

`setTimeout` is a scheduling function in JavaScript that can be used to schedule the execution of any function. It is a web API provided by the browsers and used to execute a function after a specified time. Here's the basic syntax:

```
var timerId = setTimeout(callbackFunction, timeToDelay)
```

The `setTimeout` function takes input as a `callbackFunction` that is the function which will be executed after the timer expires, and `timeToDelay` is the time in milliseconds after which the `callbackFunction` will be executed.

The `setTimeout` function returns a `timerId`, which is a positive integer value to uniquely identify the timer created by the call to `setTimeout`; this value can be passed to `clearTimeout` to cancel the timeout.

Example

```
function delayFuncExec() {
        console.log("I will be called after 100 milliseconds");
}

var timerId = setTimeout(delayFuncExec, 100)

console.log("Timer Id: " + timerId)
```

Here, `delayFuncExec` will be executed after 100 milliseconds. `timerId` will store the integer returned by the `setTimeout` function.

clearTimeout

`clearTimeout` function is used to cancel the timeout previously established by calling the `setTimeout` function. `clearTimeout` takes the `timerId` returned by the `setTimeout` function as input and cancels its execution. So, if you want to cancel the execution of any `setTimeout` function, you can use `clearTimeout` function to cancel it by passing its `timerId`.

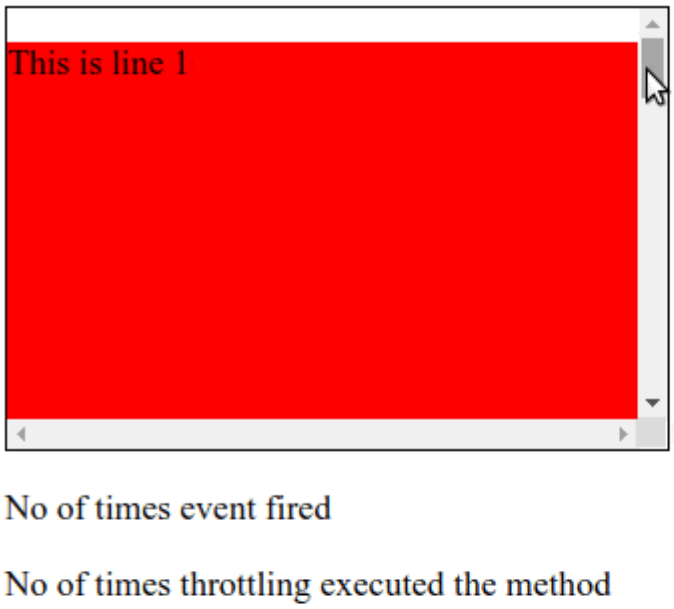
Example

```
function delayFuncExec() {
        // This statement will not be printed as it will be cancelled by
clearTimeout
        console.log("I will not be executed as I will be cancelled");
}

var timerId = setTimeout(delayFuncExec, 100)
console.log("Timer Id: " + timerId)

clearInterval(timerId)
```

Now, let try to implement throttling and debouncing using JavaScript.

Implementing Throttling in JavaScript

Throttling will change the function in such a way that it can be fired at most once in a time interval. For instance, throttling will execute the function only one time in 1000 milliseconds, no matter how many times the user clicks the button.



In the above image, we can see that when the user is scrolling, the number of `scroll` event is much larger than the number of times throttling executed the function. Also, throttling executed the function at regular intervals, irrespective of the number of time `scroll` event is fired. Let's check the code for the throttling.

Here's the HTML for throttling example:

throttling.html

```
<html>
        <style>
                div {
                        border: 1px  solid  black;
                        width: 300px;
                        height: 200px;
                        overflow: scroll;
                }
        </style>
        <body>
                <div  id="div-body">
                        <p style="background-color: red; height: 700px">This is
line 1</p>
                        <p style="background-color: blue; height: 700px">This is
line 2</p>
                        <p style="background-color: green; height: 700px">This is
line 3</p>
                        <p style="background-color: yellow; height: 700px">This is
line 4</p>
                </div>

                <p>No of times event fired</p>
                <p id='show-api-call-count'></p>

                <p>No of times throttling executed the method</p>
                <p id="debounc-count"></p>
        </body>

        <script  src="throttling.js">  </script>
</html>
```

And here's the JavaScript code for throttling example:

throttling.js

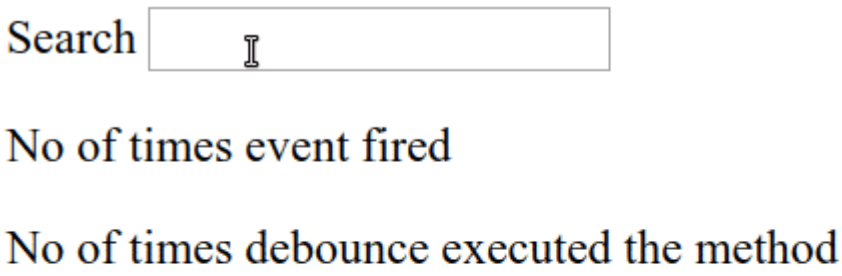
```
var  timerId;
var  divBodyDom  =  document.getElementById('div-body');

// This represents a very heavy method which takes a lot of time to execute
function  makeAPICall() {
        var  debounceDom  =  document.getElementById('debounc-count');
        var  debounceCount  =  debounceDom.innerHTML  ||  0;

        debounceDom.innerHTML  =  parseInt(debounceCount) +  1
}

// Throttle function: Input as function which needs to be throttled and delay is
the time interval in milliseconds
var  throttleFunction  =  function (func, delay) {
        // If setTimeout is already scheduled, no need to do anything
        if (timerId) {
                return
        }

        // Schedule a setTimeout after delay seconds
        timerId  =  setTimeout(function () {
                func()

                // Once setTimeout function execution is finished, timerId =
undefined so that in <br>
                // the next scroll event function execution can be scheduled by
the setTimeout
                timerId  =  undefined;
        }, delay)
}

// Event listener on the input box
divBodyDom.addEventListener('scroll', function () {
        var  apiCallCountDom  =  document.getElementById('show-api-call-count');
        var  apiCallCount  =  apiCallCountDom.innerHTML  ||  0;
        apiCallCount  =  parseInt(apiCallCount) +  1;

        // Updates the number of times makeAPICall method is called
        apiCallCountDom.innerHTML  =  apiCallCount;

        // Throttles makeAPICall method such that it is called once in every 200
milliseconds
        throttleFunction(makeAPICall, 200)
})
```

The above code renders a `div` with a scrollbar. We have added an event listener on the scroll event. Each time the user scrolls, an anonymous function is called that prints the number of times the scroll event is fired. When the user scrolls, we want to execute the `makeAPICall` method. But, as this method is heavy, attaching it to a scroll event directly will cause it to fire frequently. This may cause unnecessary load on the server. To prevent this, we have used the technique of throttling.

Let's examine the above code line by line:

1. When the first scroll event is fired, `throttleFunction` is called with the `makeAPICall` function and `delay` in milliseconds as parameters.
2. Inside `throttleFunction`, `timerId` is `undefined`, as it has not been initialized yet. So, we will go ahead and schedule the `func` that is the `makeAPICall` function using the `setTimeout` function. The `setTimeout` function will execute the `func` or the `makeAPICall` function after 200 milliseconds. Now, `timerId` will have the unique id of this `setTimeout` function.
3. When the second event for the scroll is fired, `timerId` is not `undefined` inside `throttleFunction`, so the function will return without scheduling the execution of `makeAPICall`. If `timerId` is not `undefined` it means that a `setTimeout` function has already been scheduled, hence we do not need to schedule another function.
4. Thus, unless and until `setTimeout` executes the `makeAPICall` function, we will not be able to schedule another `makeAPICall` function using `setTimeout`. This ensures that the `makeAPICall` function will be called only once in an interval.
5. The point to be noted is: inside the `setTimeout` function we have changed `timerId` to `undefined`, so once the scheduled `makeAPICall` function has been executed and the user again performs the scroll, `throttleFunction` will schedule the execution of the `makeAPICall` function with the help of the `setTimeout` function. Thus the `makeAPICall` function will be executed only once in a given interval.

Implementing Debouncing in JavaScript



In the above image, we can see that, when the user is typing, the number of `oninput` events fired is much larger than the number of times debounce executed the function. Also, debounce executed the function only after the user stopped typing in the search bar. Let's check the code for debouncing:

Here's the HTML for debounce example:

debounce.html

```
<html>
        <body>
                <label>Search</label>
                <!-- Renders an HTML input box -->
                <input  type="text"  id="search-box">

                <p>No of times event fired</p>
                <p  id='show-api-call-count'></p>

                <p>No of times debounce executed the method</p>
                <p  id="debounce-count"></p>
        </body>
        <script  src="debounce.js"></script>
</html>
```

Here's the JavaScript for debounce example:

debounce.js

```
var  timerId;
var  searchBoxDom  =  document.getElementById('search-box');

// This represents a very heavy method. Which takes a lot of time to execute
function  makeAPICall() {
        var  debounceDom  =  document.getElementById('debounce-count');
        var  debounceCount  =  debounceDom.innerHTML  ||  0;

        debounceDom.innerHTML  =  parseInt(debounceCount) +  1
}

// Debounce function: Input as function which needs to be debounced and delay is
the debounced time in milliseconds
var  debounceFunction  =  function (func, delay) {
        // Cancels the setTimeout method execution
        clearTimeout(timerId)

        // Executes the func after delay time.
        timerId  =  setTimeout(func, delay)
}

// Event listener on the input box
searchBoxDom.addEventListener('input', function () {
        var  apiCallCountDom  =  document.getElementById('show-api-call-count');
        var  apiCallCount  =  apiCallCountDom.innerHTML  ||  0;
        apiCallCount  =  parseInt(apiCallCount) +  1;

        // Updates the number of times makeAPICall method is called
        apiCallCountDom.innerHTML  =  apiCallCount;

        // Debounces makeAPICall method
        debounceFunction(makeAPICall, 200)
})
```

The above code renders an HTML page with a textbox. We have added an `oninput` event listener on the textbox that gets fired each time the user types something in the textbox.

When the user types something, we want to call the `makeAPICall` method that makes an API call to fetch the data. But, as this method is heavy, we do not want to call it each time the user inputs something in the textbox. So instead of calling it directly inside the anonymous function, we have called a `debounceFunction` function inside which `makeAPICall` method is called.

Let's understand more about the `debounceFunction`:

The `debounceFunction` is used to limit the number of times any function is executed. It takes input as the `func` that is a function whose execution has to be limited, and `delay` that is the time in milliseconds. If the user types very fast, the `debounceFunction` will allow the execution of `func` only when the user has stopped typing in the textbox.

Let's examine the above code line by line:

1. When the user types the first letter in the textbox, event handler or the anonymous function calls the `debounceFunction` with the `makeAPICall` function and 200 milliseconds as parameters.
2. Inside the `debounceFunction`, `timerId` is `undefined`, as it has not been initialized so far. Hence, `clearTimeout` function will do nothing.
3. Next, we pass `func` that is the `makeAPICall` function as a callback to the `setTimeout` function, with `delay` that is 200 milliseconds as another parameter. This means that we want the `makeAPICall` function to be executed after 200 milliseconds. The `setTimeout` function returns an integer value as its unique id, which is stored by the `timerId`.
4. Now, when the user types a second letter in the textbox, again `debounceFunction` is called. But this time `timerId` is not `undefined` and it stores the unique id of the previous `setTimeout` function. Hence, when `clearTimeout` function is called with this `timerId`, it cancels the execution of the previous `setTimeout` function.
5. Hence, all `func` or `makeAPICall` functions scheduled by `setTimeout` function due to the user typing in the textbox will be cancelled by the `clearTimeout` function. Only the `makeAPICall` function scheduled by the last letter in the textbox will execute after the specified time of 200 milliseconds.

Thus, no matter how many letters the user types in the textbox, the `debounceFunction` will execute the `makeAPICall` method only one time after 200 milliseconds - after the user types the last letter. And that's debouncing!

Use of Debouncing and Throttling in Real Life

1. We can throttle a `button click` event, if we do not want the user to spam by clicking the button frequently.
2. In the case of `window resize` event, if we want to redraw the UI only when the user has fixed the size of the window, we can use debouncing.
3. With `Mouse move` event, if we are displaying the location coordinates of the mouse pointer, it is much better to show the coordinates once the user reached the desired location. Here, we can use debouncing.

Conclusion

In this article, we understood the concepts of debouncing and throttling along with their uses in the real world. We also implemented both the concepts using HTML and JavaScript.