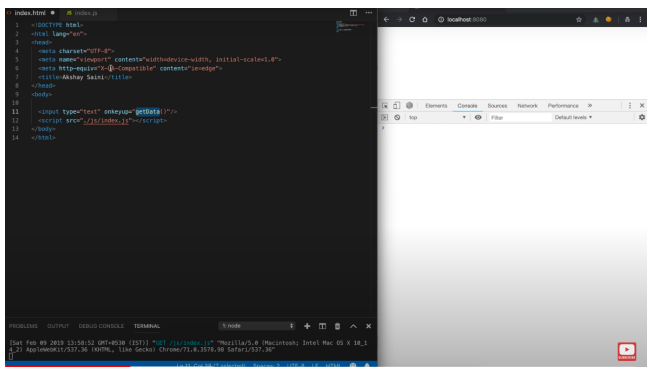
Debouncing and Throttling in JavaScript

Saturday, 26 August 2023 12:41 AM

→ Debouncing:

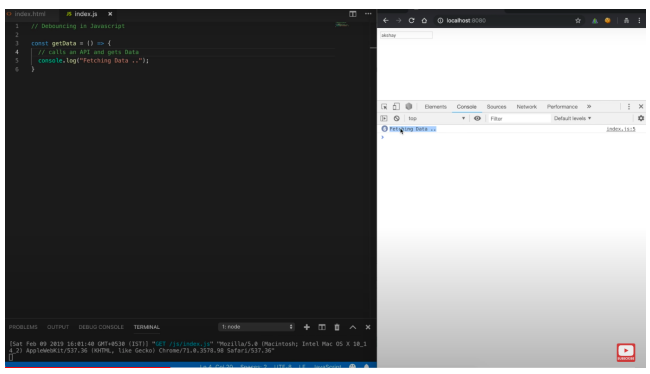
This concept can be used for example, in a search bar. So if a search bar is suggesting things on the basis of what you type then it would be making API calls on change of a value, but in large scale applications it would result in a lot of API calls.

So we can implement something so that there is a pause between different API calls. This can be achieved with debouncing.

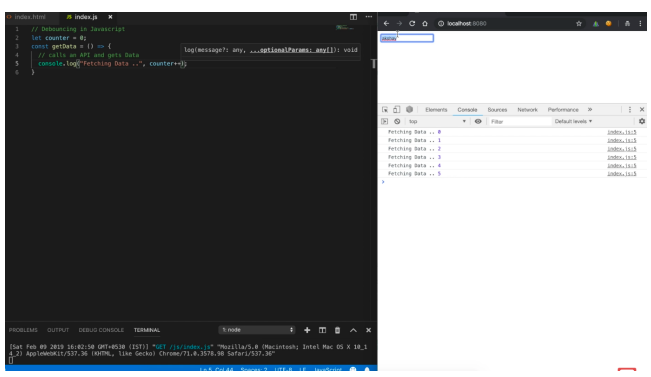


→ HTML code

→ This calls a fn getData() on the event of key up.



→ Now whenever we type in the inputfield our fn gets called.



→ So, here we can see that we have sort of called the API 6 times

→ So we want to achieve that once we pause while typing then only API call should be done. Eg: only call getData() API after user pauses for 300ms.

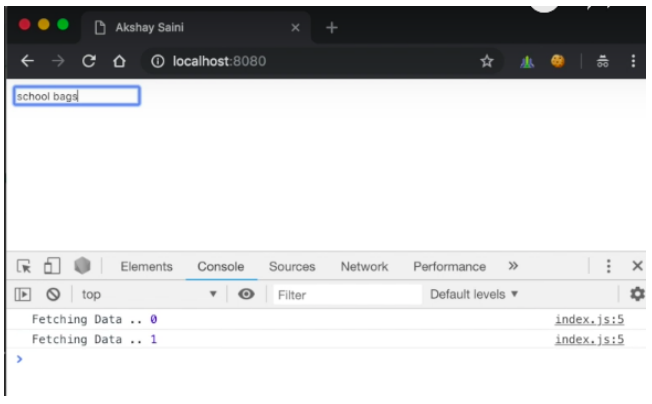
→ So, we must create a better fn.

```
index.html ● JS index.js
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="UTF-8">
5    <meta name="viewport" content="width=device-width, initial-scale=1.0">
6    <meta http-equiv="X-UA-Compatible" content="ie=edge">
7    <title>Akshay Saini</title>
8  </head>
9  <body>
10
11   <input type="text" onkeyup="betterFunction()"/>
12   <script src="./js/index.js"></script>
13 </body>
14 </html>
```

→ Changing the name of fn is HTML code.

→ betterFunction() implementation:

```js
let counter = 0;
function getData () {
    console.log("Hello ji main kya kru", ++counter);
}

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

const betterFunction = debounce(getData, 300)
```



↓

explanation:

O/p as fn will be called after a delay of 300ms so less API calls

→ We use betterFunction() which takes in the getData() callback fn (which will perform the API call)

→ This debounce() fn has 2 arguments : callback fn & delay.

→ Inside the debounce() polyfill, we receive callback fn & delay as parameters.

→ This will return us an optimized fn which will call the callback fn (here, getData) only when 300ms have been passed.

→ So, we create a setTimeout & call the callback fn inside it by using .apply() . We use .apply() to call cb() as we want to explicitly bind context (i.e, this) to cb() fn.

→ We define let context = this and pass it inside the .apply() method so, that value of 'this' is not lost. Because we may use betterFunction() by binding it to some object. Eg: obj.betterFunction(), so now 'this' will point to 'obj' which we do not want, so we save its instance in context, so that context points to inner fn always.

→ We store setTimeout inside a timerId as we need to call clearTimeout() before setTimeout & thus, declare timerId outside the inner fn.

→ We clear the timeout before we run setTimeout to clear previous instance of setTimeout (if there is one) and reset the timer basically. So, whenever user presses the timer gets cleared & a new timer is created, so API is not called for another 300 ms.

→ Also, if the betterfunction() takes in some arguments, in order to handle that we pass (...args) in the returned fn & pass the same when using .apply().

→ Debouncing vs Throttling : Both are used for optimizing the performance of a webApp. We do so by limiting the rate of function calls.

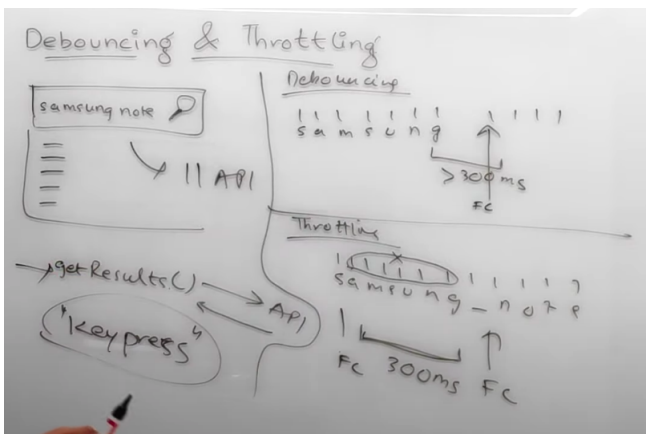
→ Where do we use which?

- Search bar of an ecommerce-website: Calling the API of search result on each key stroke event will result in a lot of API calls.

  We can optimize this by:

  Debouncing: We will make the API call only if the difference between both key stroke events is greater than the limit set. Otherwise restart the timer again if another key stroke event is fired.

  Throttling: Even if there are many events happening, only make the fn call after a certain limit of time. So, if a fn call is made and API is called, then wait until a certain period of time & ignore all API calls during that time & only a certain period of make the API call again.



→ Debouncing is more useful in above.

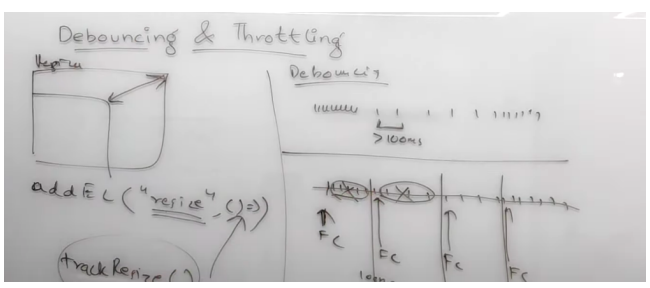
- To track how many times user is resizing the browser: In this case, an eventlistener is attached to the window which calls trackResize() on the event of `resize`:

  Once you resize the window, the resize event is triggered loooos of times.

  To solve this:

  Debouncing: If the difference b/w two resizing events is greater than a particular time, only then call the trackResized() method.

  Throttling: In this we make the fn call once the user resizes & track the resize event and automatically calls the fn next time only after a certain limit. So, in between events will be disregarded or ignored.



→ Both cases are suitable for this.

- Shooting game: If we have a shooting game wherein we have to limit the button press, we can use debouncing & throttling for different weapons. Throttling would be more beneficial here.

  Pistol: Here we will use debouncing as trigger should be clicked & ammo should be fired only after a certain time.

  Machine gun: Here the trigger will always be active & event would always be firing but after once ammo starts firing it will be fired after a certain time itself, so we will use throttling.

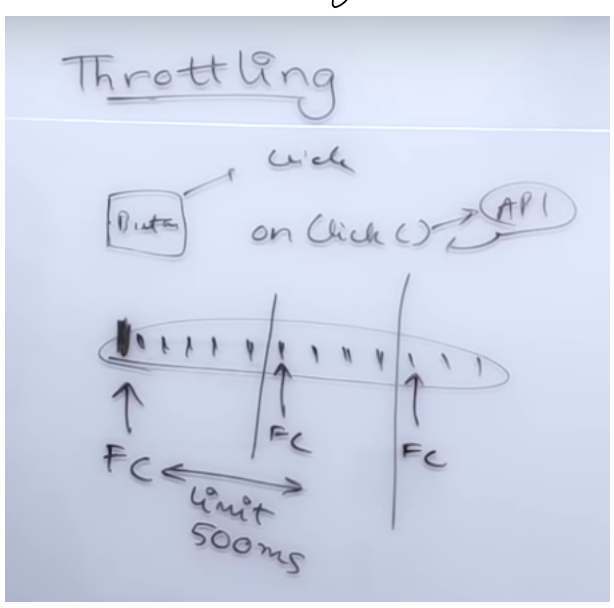
- We can use debouncing & throttling for capturing scroll event & making it optimized.

→ Which is better debouncing or throttling?

ans: Both of them are optimization techniques & both have their own use cases and benefits & we must decide which one to use according to our use case.

→ throttling: This technique is used for optimization or rate limiting the fn calls.

Let's assume that there is a button and the user keeps on clicking the btn, so API will be called that many times but we don't want that, so we use throttling wherein that fn will be called first but after that it will be called after a certain period of time automatically again & again while the in between API calls will be disregarded which are caused due to constant clicking of btn.



→ Example of throttle() fn to improve optimization:

Assuming we are capturing the resize event which makes the callback fn inside the eventlistener run loads of times. So in order to improve this we use throttling.

main.js

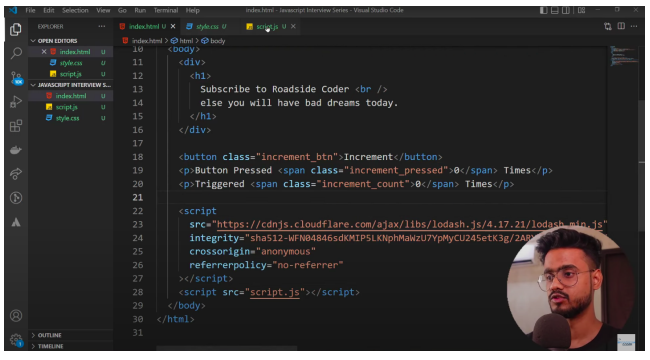
```
const expensive = (...args) => {
    console.log("Expensive Function")
}

window.addEventListener("resize", betterExpensive);

const betterExpensive = throttle(expensive, limit);

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

→ So, here we have an expensive() fn, which needs to be throttled.
→ We create a throttle() fn which will return a fn which is the throttled version of expensive()
→ We store the fn inside betterExpensive() & call it as a callback fn inside an event listener.
→ If the expensive() fn takes in any arguments then for that we've passed the arguments as ...args inside the returned fn.
→ Now, we create a flag of true, we create this outside otherwise whenever the fn will be called it will create a new instance of flag & callback fn will be called infinite times.
→ throttle() fn takes in a callback fn & a limit.
→ Now returned fn will make the callback fn run 1 time (as flag is true) we then turn flag to false and using setTimeout wait for the time limit to make the flag true again in order for the callback fn func() to run again. This is so because in throttling fn is called the first time an event is fired & then the fn is called again & again after a particular amount of time.
→ We use func.apply() method to pass argument as a list.
→ We also store 'this' in 'context' & pass it to apply. Returned fn makes closure with its parent so it has access to 'context' and 'flag'.
→ We store 'this' in 'context' so that we don't lose access to 'this', because we can lose 'this' if we bind it to some object. Eg: during explicit binding like obj.betterExpensive(), so 'this' will now point to 'obj' which we don't want so we store its instance in 'context'.

eg: Flipkart search uses debouncing & Twitter infinite scroll uses throttling.

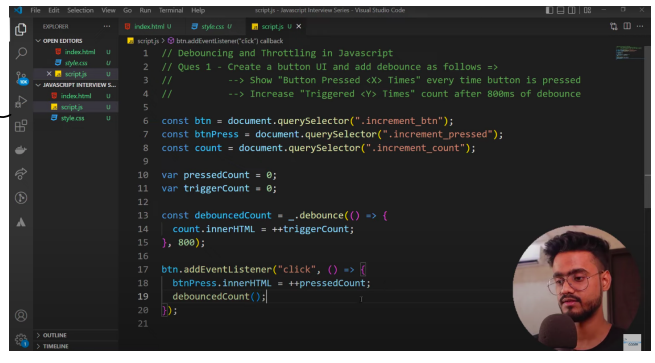
Note: ⇒ We can use lodash library also for implementing debouncing & throttling.

eg: Example for lodash library usage for debouncing:



HTML file

JS file



(added CDN for lodash)

→ We can also use lodash for throttle similarly.