Why, What & How

Debounce in JavaScript

M. SHAHZEB RAZA
REACT WEB DEVELOPER

Swipe →

The Setup

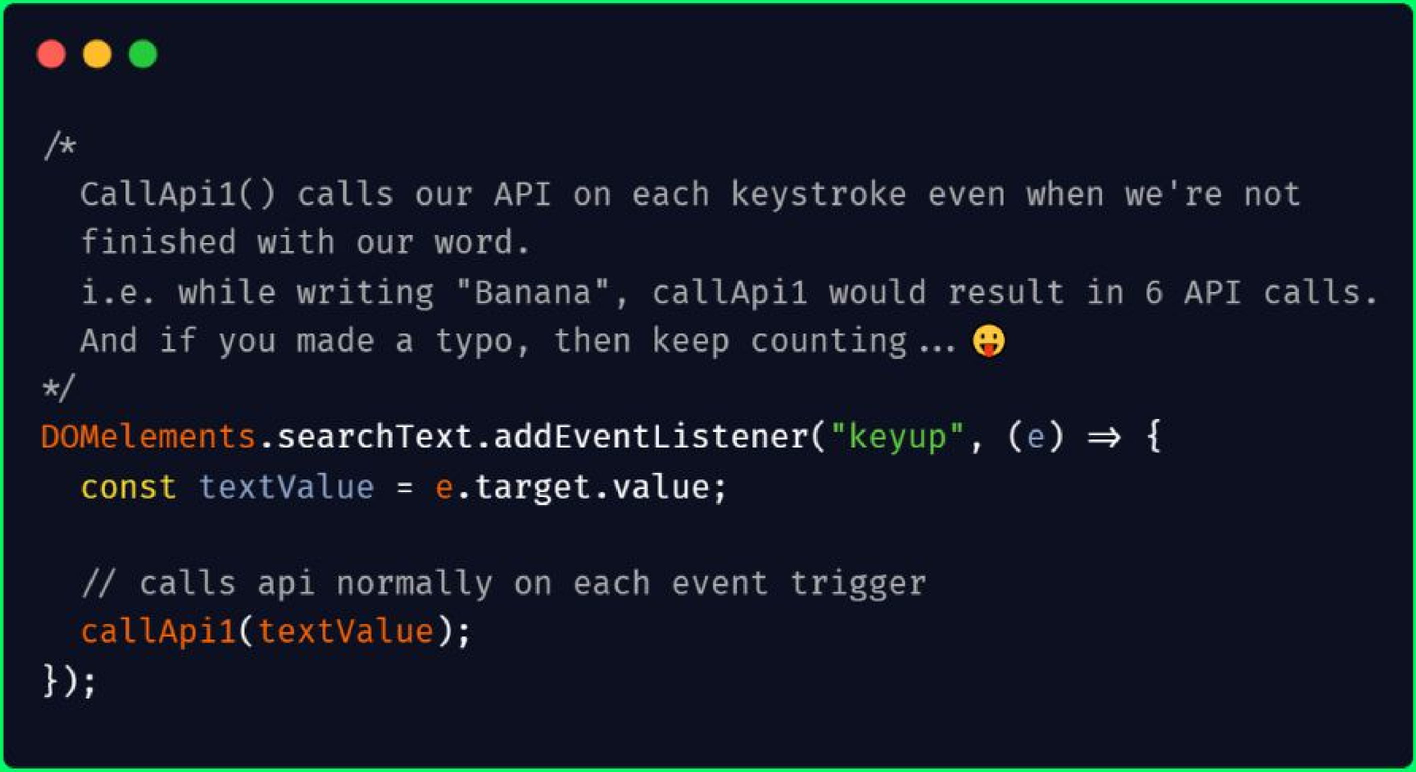
Suppose we want to use the value of searchbar input to fetch data from an API. (Usually used to show suggestions)

We do this by using **event-listeners** in JavaScript to look for changing value and then **use new value to send requests to api-endpoints**.

In the following example, event listener is used to run **callApi1()** on each **"keypress"** event.

Swipe →

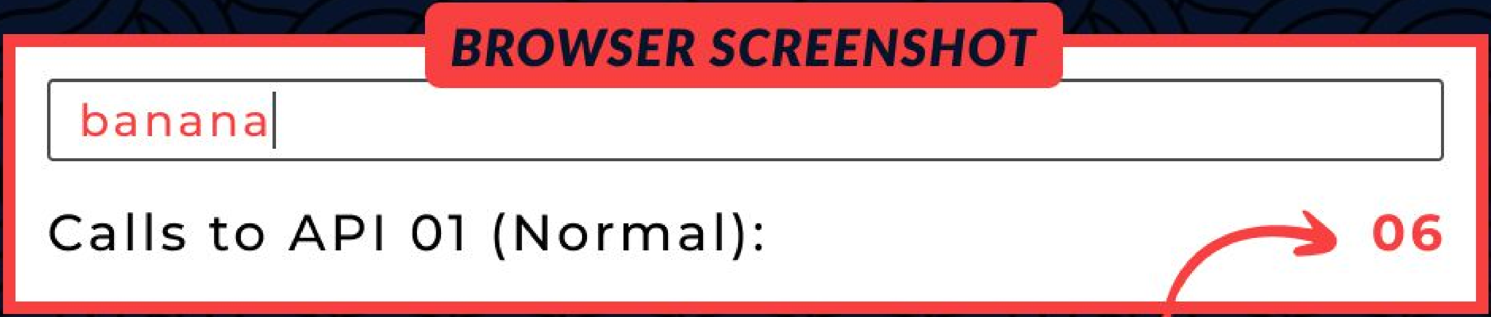
The Setup



```
/*
  CallApi1() calls our API on each keystroke even when we're not
  finished with our word.
  i.e. while writing "Banana", callApi1 would result in 6 API calls.
  And if you made a typo, then keep counting... 😛
*/
DOMelements.searchText.addEventListener("keyup", (e) => {
  const textValue = e.target.value;

  // calls api normally on each event trigger
  callApi1(textValue);
});
```

Swipe →

The Problem


BROWSER SCREENSHOT

banana

Calls to API 01 (Normal): 06

You might've noticed that callApi1() is called on each **"keypress"** event

This is **NOT optimal** as we don't want to search while we are typing.

Swipe →

The Problem

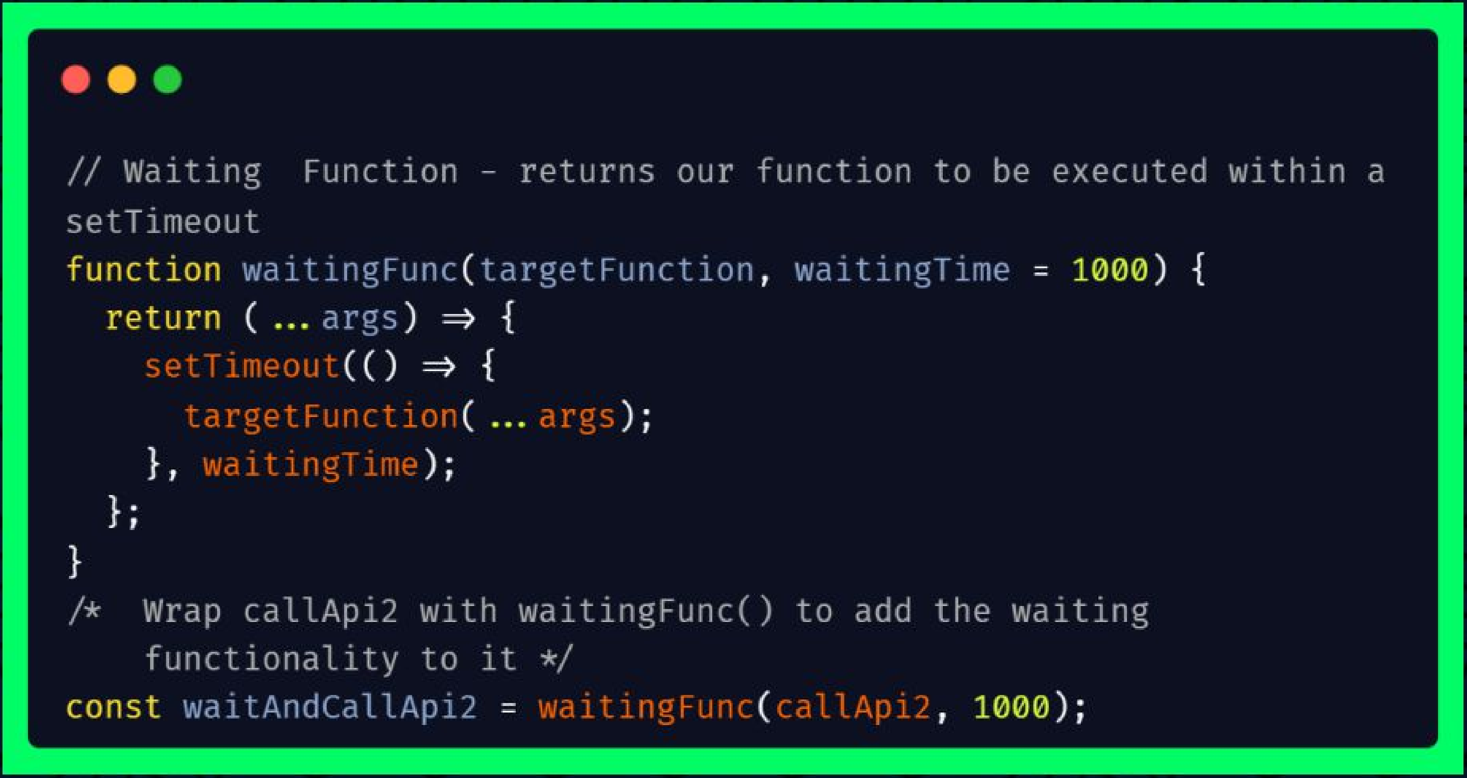
Sending the API request to server too many times can be **performance-expensive**. Also, paid APIs can cost a lot too.

What can we do then?

Let’s create a function that waits a bit before sending any requests to server, so we can complete our “SearchText”

Swipe →

Find a Solution: *Waiting Fn*

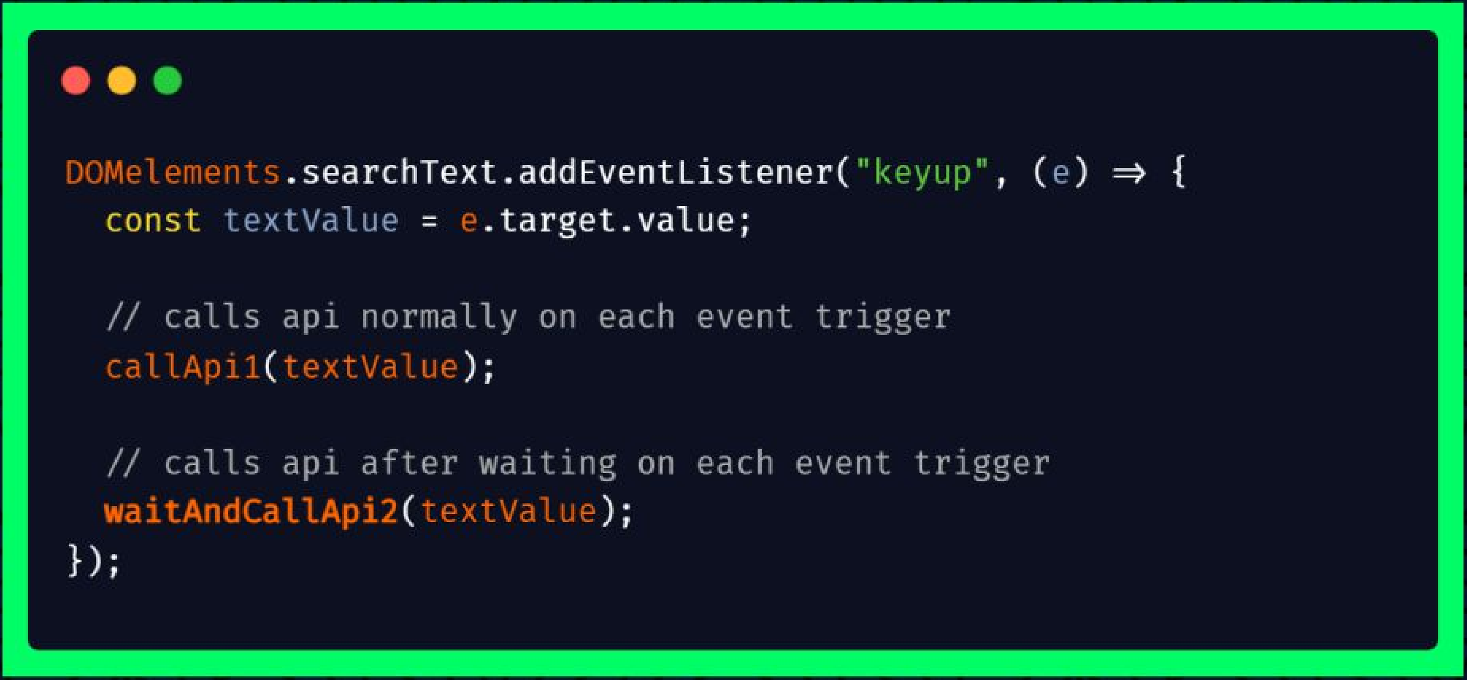


```javascript
// Waiting  Function - returns our function to be executed within a
setTimeout
function waitingFunc(targetFunction, waitingTime = 1000) {
  return (...args) => {
    setTimeout(() => {
      targetFunction(...args);
    }, waitingTime);
  };
}
/*  Wrap callApi2 with waitingFunc() to add the waiting
    functionality to it */
const waitAndCallApi2 = waitingFunc(callApi2, 1000);
```

Let’s use waitAndCallApi2 to call Api 2 in the our event listener

Swipe →

Find a Solution: *Waiting Fn*



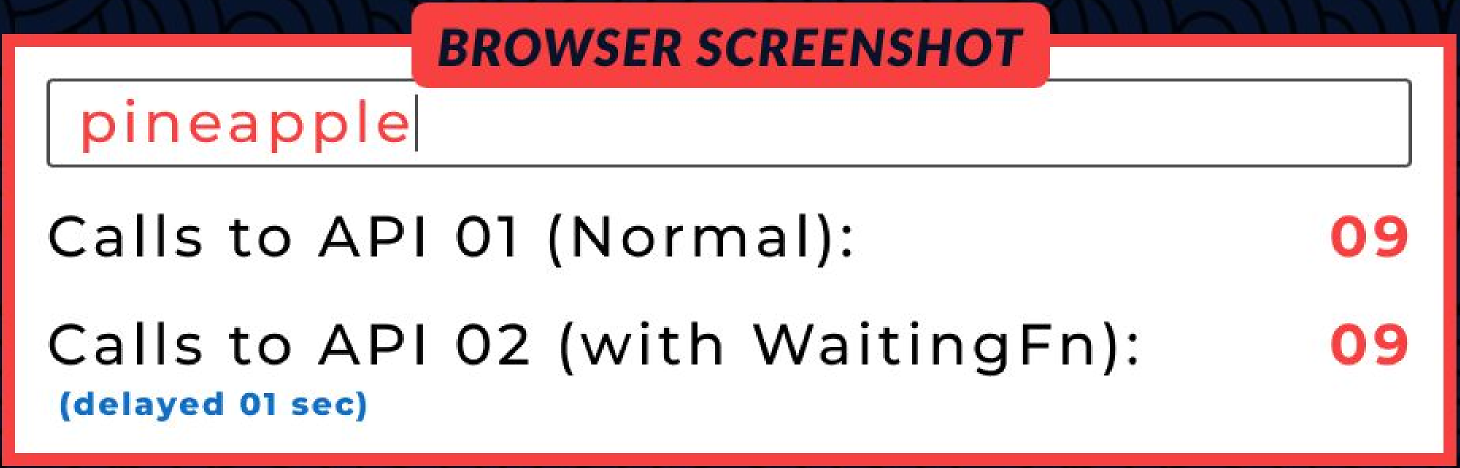
```
DOMelements.searchText.addEventListener("keyup", (e) => {
  const textValue = e.target.value;

  // calls api normally on each event trigger
  callApi1(textValue);

  // calls api after waiting on each event trigger
  waitAndCallApi2(textValue);
});
```

Now, on each **"keyup"** event, both **callApi1()** & **waitAndCallApi2()** are called.

But **waitAndCallApi2()** calls our **callApi2()** with a delay.

Swipe →

Check our Solution: *Waiting Fn*



pineapple

Calls to API 01 (Normal): **09**

Calls to API 02 (with WaitingFn): **09**
(delayed 01 sec)

Now, upon searching for **"pineapple"** **(9 letters)**, we see that **09 calls** are made to both APIs.

The only difference is that **waitAndCallApi2()** calls the API-2 with a delay of 1000ms and unlike **callApi1()** which calls API-1 instantly on each event.

Swipe →

Inspect the problem again!

So, this is what we wanted, right?

Well, Not Exactly!

VERY IMPORTANT

We did want to call our API with a delay **(schedule it)**.
But in addition to that **if a new API-call** is made during that **waitingTime,** we wanted to **cancel any previously scheduled API-calls**, AND **schedule this new API-call instead.**

You might want to read that again 👆

Swipe →

Fix our solution!

We're halfway there! *(kinda)*

Okay! So what do we do now!

Step 01:

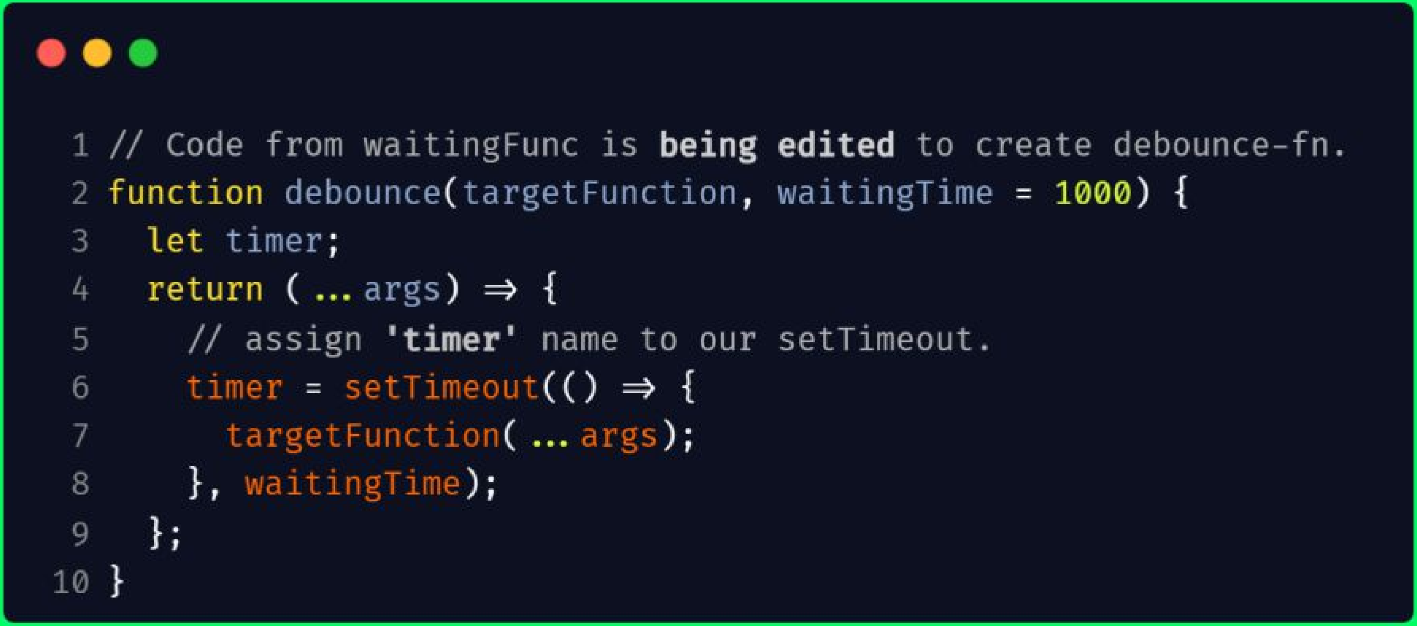
Add the cancel-setTimeout functionality.

To cancel a timeout, we can use **clearTimeout(nameOfTimer)** to clear the scheduled execution for the timeout.

So let's assign our timeout to a name. We've named our timeout **"timer"** in this example.

Swipe →

Fix our solution!



```javascript
// Code from waitingFunc is being edited to create debounce-fn.
function debounce(targetFunction, waitingTime = 1000) {
  let timer;
  return (...args) => {
    // assign 'timer' name to our setTimeout.
    timer = setTimeout(() => {
      targetFunction(...args);
    }, waitingTime);
  };
}
```

*Now that we've named our **"timer"** function, we can use **clearTimeout(timer)** to cancel it.*

Swipe →

Fix our solution!

But when do we want to clear it?

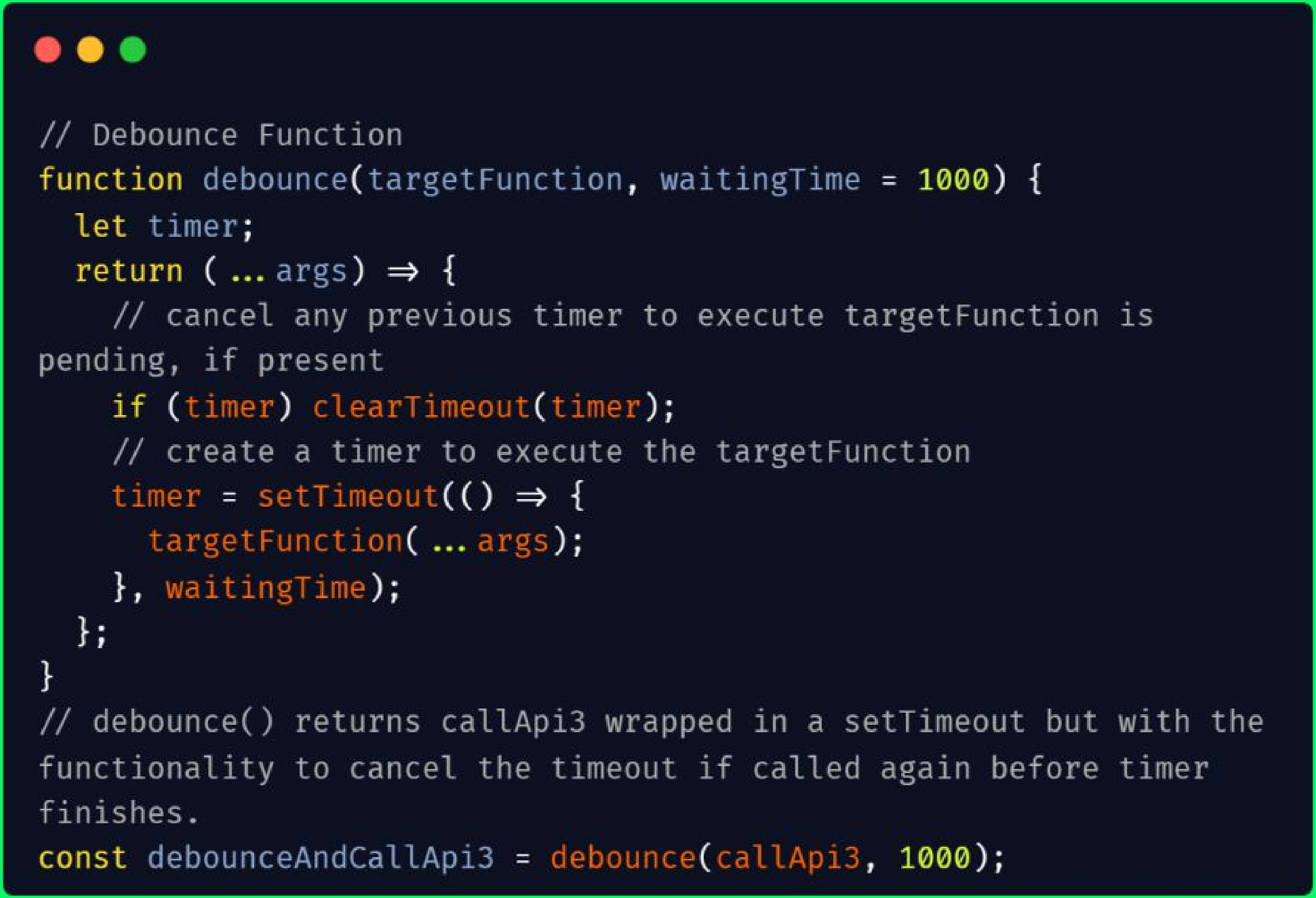
We want to cancel the previous timeout before setting up a new timeout.

This means, we must **check if a timer is scheduled already**.
If **YES**, we must clear it before proceeding further!

Let's **implement the code** for it!

Swipe →

The New Solution

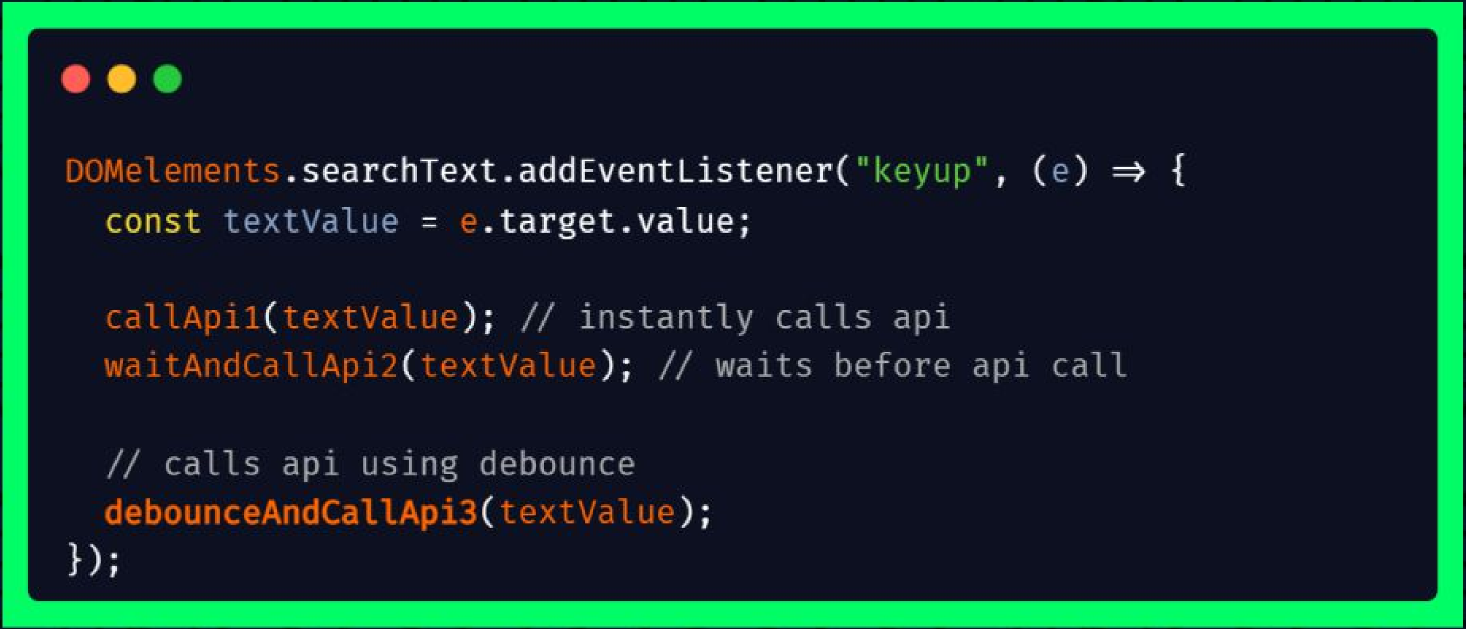


```
// Debounce Function
function debounce(targetFunction, waitingTime = 1000) {
  let timer;
  return (...args) => {
    // cancel any previous timer to execute targetFunction is
pending, if present
    if (timer) clearTimeout(timer);
    // create a timer to execute the targetFunction
    timer = setTimeout(() => {
      targetFunction(...args);
    }, waitingTime);
  };
}
// debounce() returns callApi3 wrapped in a setTimeout but with the
functionality to cancel the timeout if called again before timer
finishes.
const debounceAndCallApi3 = debounce(callApi3, 1000);
```

Swipe →

The New Solution

Let’s use, we can use **debounceAndCallApi3()** in our event listener.

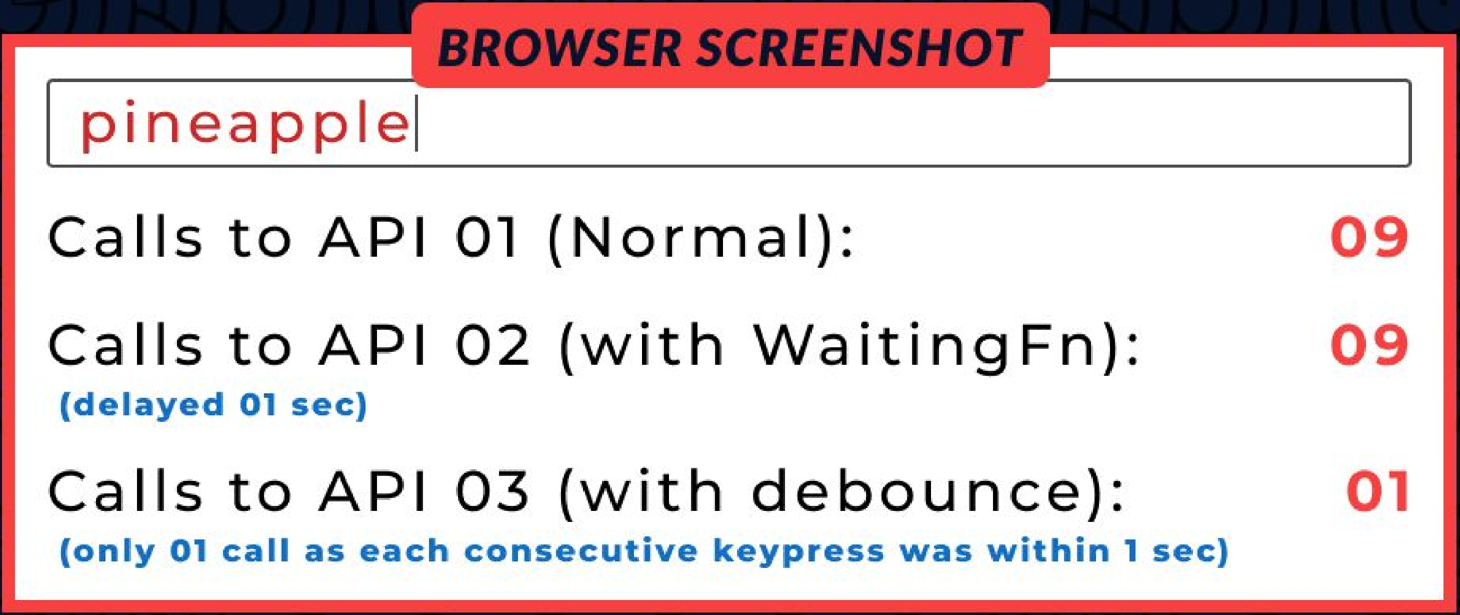


```
DOMelements.searchText.addEventListener("keyup", (e) ⇒ {
  const textValue = e.target.value;

  callApi1(textValue); // instantly calls api
  waitAndCallApi2(textValue); // waits before api call

  // calls api using debounce
  debounceAndCallApi3(textValue);
});
```

Now Let’s **go to browser** to compare the working of all 3 functions..

Swipe →

Check our New Solution: Success!



pineapple

Calls to API 01 (Normal): 09

Calls to API 02 (with WaitingFn): 09
(delayed 01 sec)

Calls to API 03 (with debounce): 01
(only 01 call as each consecutive keypress was within 1 sec)

Note that even when the “keyup” event runs **callApi1()**, **waitAndCallApi2()** and **debounceAndCallApi3()**.
The API request from **debounceAndCallApi3()** is only **01.**

Swipe →

What is exactly happening!

This is because all the subsequent letters of pineapple after **‘P’ (1st Letter)** were typed quickly (less than 1000ms delay).

It implies that the **API-call scheduled by ‘P’ (1st Letter) was cancelled when API-call of ‘I’ (2nd Letter) was scheduled.**

Similarly, **API-call of ‘N’ (3rd letter) resets the ‘timer’ scheduled by ‘I’ (2nd letter)** and so on...

Swipe →

What is exactly happening!

In this way, **all the letters cancel the API calls be letters before them except the last letter ‘E’**.

Why? Because **there is no letter after ‘E’** to cancel the API call scheduled by ‘E’.

🔰 ***CHALLENGE QUESTION !*** 🔰

Now, go to slide 12 and think about why was the ‘timer’ name initialized in the parent scope.

Swipe →

Did you found it Helpful?

M. SHAHZEB RAZA

REACT WEB DEVELOPER

Like ❤️ if you found it helpful! 😺

Comment if your answers 😺 for the 🔰**Challege Question!**🔰

FOLLOW FOR MORE! 🔥