Mastering JavaScript Dates: Part 1 - Understanding Time Zones, UTC

Time Zones

UTC - Coordinated Universal Time. It is a time zone that is used to coordinate the time across the world. It is also known as GMT (Greenwich Mean Time). Whenever, astronauts travel outside earth, they refer this time.

UTC time is micro coordinated, i.e. it is maintained very strictly according to the earth's rotation speed.

ISO - International Standard Organization. It is a standard for the representation of dates.

IST - Indian Standard Time. It is the time zone in which the Indian people live. It is also known as UTC +5:30 or GMT +5:30

ISO Date and Time Representation : `YYYY-MM-DDTHH:MM:SS.MMM+05:30`

We can also write `Z` instead of +05:30 at last which means it is UTC 00:00

YYYY - Year MM - Month DD - Day

T - Time starts from Here HH - Hour MM - Minute SS - Second MMM - Millisecond +05:30 - Time Zone (UTC +5:30) Z - UTC Time (00:00)

How computers keep track of time?

For computers, to keep track of time, they decided to keep `1970-01-01T00:00:00.000Z` as the starting point.

This was set by UNIX operating system developers.

So there is an RTC chip in our system which feeds the program with the electricity which keeps it vibrating and keep track of time, even when we shut down our system. When we start our computer, it also cross checks time via NTP (Network Time Protocol) which is a protocol which keeps track of time across the world via an Atomic clock.

Unix Time and Unix Epoch

Unix Time - Total Time in milliseconds since 1970-01-01T00:00:00.000Z till current time.

Unix Epoch - 1970-01-01T00:00:00.000Z (A remarkable event which is this day in history).

Unix Time Glitch - Unix devices uses a 32 bit signed integer to keep track of time. But on Jan 19 2038, the 32 bit signed integer will overflow and make the time negative. This is called Unix Time Glitch. To avoid this, we use 64 bit unsigned integer.

Date Object in JavaScript

When we call `Date()` it returns us a string, but `Date()` is a constructor function, so we always call it with `new` keyword as `new Date()`. This will return us an Object with same data that we got earlier in string format.

```
const date = new Date();

console.log(date.getTime());
```

Gives the time in milliseconds from 1970-01-01T00:00:00.000Z (this is also called unix timestamp).

```
const date = new Date();

console.log(date.toLocaleString());
```

Shows the date and time in a representable format.

But it shows in `MM-DD-YYYY`, so to get `DD-MM-YYYY` (Great Britain (GB) style) we can do:

`console.log(date.toLocaleString('en-GB'));`

Now time is shown in 24 hour format, so to make it into 12 hour format, we can do:

```
const date = new Date();

console.log(date.toLocaleString("en-GB", { hour12: true }));
```

To get only the date in GB style, we can do:

`console.log(date.toLocaleDateString('en-GB'));`

Let us see some more date methods:

```
const date = new Date();

console.log(date.toLocaleDateString("en-GB"));
// gets only date
// 20/12/2024

console.log(date.getDate());
// gets current DD
// 20

console.log(date.getFullYear());
// gets current YYYY
// 2024

console.log(date.getMonth());
```

```
// gets current MM
// 11 (Month starts from 0, hence 11 means month of December)

console.log(date.getUTCFullYear());
// gets current YYYY
// 2024

console.log(date.getDay());
// gets current Day no.
// 5 -> Friday
// (starts from monday (1) to sunday (7))

console.log(date.getMilliseconds());
// gets current milliseconds
// 0 to 999

console.log(date.getSeconds());
// gets current seconds
// 0 to 59

console.log(date.getMinutes());
// gets current minutes
// 0 to 59

console.log(date.getHours());
// gets current hours
// 0 to 23

console.log(date.getTimezoneOffset());
// gets current timezone offset in minutes
// 0 to 1440

// for hours, we divide it by 60
console.log(date.getTimezoneOffset() / 60);
// it shows the number in negative
// eg.. -5.5 for +5:30

console.log(date.toISOString());
// gives us the date in ISO format (timestamp format )
// 2024-12-20T00:00:00.000Z (we will see time of UTC 00:00)

console.log(date.toLocaleDateString("en-GB"));
// gives us the date in GB format

console.log(date.toLocaleTimeString("en-GB", { hour12: true }));
// gives us the time in 12 hour format

console.log(
  date.toLocaleTimeString("en-GB", { hour12: false, timestyle: "short" })
);
// gives us the time in 24 hour format in short format

console.log(date.toLocaleTimeString("en-GB", { timestyle: "medium" }));
```

```js
console.log(
  date.toLocaleTimeString("en-GB", { hour12: false, timestyle: "long" })
);

console.log(
  date.toLocaleTimeString("en-GB", { hour12: false, timestyle: "full" })
);

console.log(date.toLocaleDateString("en-GB", { datestyle: "full" }));
```

This Date object relies on our computer's time, so if we change the time zone in our computer, we will get the output here accordingly.

Seeing some more date methods:

```js
const date = new Date();

console.log(date.toJSON());
// 2024-12-19T23:55:36.736Z
// output is same as ISO string

console.log(date.toString());
// Fri Dec 20 2024 05:25:36 GMT+0530 (India Standard Time)
// also converts date from object type to string type

console.log(date.toUTCString());
// string datatype of this data -> Thu, 19 Dec 2024 23:57:20 GMT
```

Similarly all these methods like getFullYear, getMonth,...date. have their UTC versions like getUTCFullYear, getUTCMonth, etc.....

Local Seconds and milliseconds and UTC Seconds and milliseconds are same. But UTC hours and local hours & local minutes and UTC minutes may be different from each other.

```js
const date = new Date();

console.log(date.getTime());
```

When we do this, we get the current timestamp (in milliseconds) from 1970-01-01T00:00:00.000Z. This keeps on changing and updating continuously every millisecond.

But when we do this,

```js
const date = new Date(1734682425235);
// provided a timestamp in the params

console.log(date.getTime());
```

We get the fixed timestamp of `1734682425235` and it remains the same even after reloading the page i.e. it does not update.

```js
const date = new Date(0);
// 1970-01-01T00:00:00.000Z (i.e. 00:00 in england on jan 1 1970)
// This is UNIX Epoch

console.log(date.toString());
// as this gives local converted time, it will give
// Thu Jan 01 1970 05:30:00 GMT+0530 (India Standard Time)
```

`new Date().getTime() and Date.now(), both give us the same result i.e. timestamp in milliseconds of UTC timezone.`

To represent time before Jan 1 1970, we can enter negative time in the Date constructor.

```js
const date = new Date(-60000);
// 1min before unix epoch i.e. Dec 31, 1969, 11:59:00 PM Z

console.log(date.toISOString());
// 1969-12-31T23:59:00.000Z
```

Let us see an example where we are comparing ages of 2 people and telling who is more older and by how much

```js
const user1dob = "2022-02-18";
const user2dob = "2002-08-29";

const user1dobms = new Date(user1dob).getTime();
const user2dobms = new Date(user2dob).getTime();

user1dobms < user2dobms
  ? console.log(`User 1 is older ${user1dobms}`)
  : console.log(`User 2 is older ${user2dobms}`);
// Wrong way (or un-ideal way)

// Correct way
const user1agems = Date.now() - user1dobms;
const user2agems = Date.now() - user2dobms;
// making their age in milliseconds

if (user1agems > user2agems) {
  console.log(
    `User 1 is older by ${
      user1agems / 1000 / 60 / 60 / 24 / 365 -
      user2agems / 1000 / 60 / 60 / 24 / 365
    } years`
  );
} else {
  console.log(
    `User 2 is older by ${
      user2agems / 1000 / 60 / 60 / 24 / 365 -
      user1agems / 1000 / 60 / 60 / 24 / 365
    } years`
  );
}
```

Different way of writing dates:

```
const user1dob = "14-01-2000";
const user2dob = "05/01/1996";
// here date is in DD-MM-YYYY

// but in JS DD-MM-YYYY is not accepted
// only MM-DD-YYYY or YYYY-MM-DD is accepted

// conversion to YYYY-MM-DD

const user1Date = new Date(user1dob.split("-").reverse().join("-"));
const user2Date = new Date(user2dob.split("/").reverse().join("/"));

console.log(user1Date.toString());
// Fri Jan 14 2000 05:30:00 GMT+0530 (India Standard Time)

console.log(user2Date.toString());
// Fri Jan 05 1996 00:00:00 GMT+0530 (India Standard Time)

// or
const [day, month, year] = user1dob.split("-").map((el) => +el);
const user1newDate = new Date(year, month - 1, day);
console.log(user1newDate.toString());
// Fri Jan 14 2000 00:00:00 GMT+0530 (India Standard Time)
// here month starts from 0, hence Jan = 1 - 1 = 0
// Also to do this multiplication, we converted the day, month and year data types
from string to number (to do month - 1)

// We cannot pass time in this method

// Some other ways to input and write dates with time are
const isoString = "2008-11-24T10:00:54.125Z";
const user3Date = new Date(isoString);
console.log(user3Date.toString());
//  Mon Nov 24 2008 15:30:54 GMT+0530 (India Standard Time)

const user4Dob = new Date(2014, 8, 25, 17, 50, 10, 478); // month = month - 1,
hence 8 = september
// (YYYY, M, DD, HH, MM, SS, MS)
console.log(user4Dob.toString());
```

```
// Thu Sep 25 2014 17:50:10 GMT+0530 (India Standard Time)
console.log(user4Dob.getMilliseconds());
// 478
```

Mastering JavaScript Dates: Part 2 - Understanding Set Methods

Date(2015, 1, 14) = > 14th Feb, 2015

Date(2015, 0, 14) = > 14th Jan, 2015

`const date = new Date(2015, -1, 14)` => -1 means Jan - 1 month, hence it is 14th Dec 2014

`const date = new Date(2015, -1, -5)` => Jan - 1 = Dec and -5 means 1 Dec - 5 => November 30 - 5 = November 25, hence ans => 25th Nov 2014 00:00:00

`const date = new Date(2015, -1, -5, -1)` => 24th Nov 2014 23:00:00

But when we enter date with `-` separating them, then our time will be adjusted to local time wrt utc time

eg. `const date = new Date('2015-7-14')` => Aug 14, 2015, 05:30:00 GMT+0530

here time changes from 00:00:00 (UTC) and gets set to 05:30:00 (local time)

But when we use `/` to separate, then we get 00:00:00 in time.

Some more Date methods

```
const date = new Date("2017-03-25 07:55:12.004 +0530");

console.log(date); // Sat Mar 25 2017 07:55:12 GMT+0530 (India Standard Time)
console.log(date.getYear()); // current year - 1900 => 2017 - 1900 = 117
console.log(date.getFullYear()); // gives the current year => 2017

console.log(date.toDateString()); // Sat Mar 25 2017
console.log(date.toTimeString()); // 07:55:12 GMT+0530 (India Standard Time)
```

Task -> Create 2 functions, one which gives us the Day name of the Week and the other which gives us the month name of the year

- Long Way

```
const date = new Date();

function getMonthName(date) {
  const monthNum = date.getMonth();
  switch (monthNum) {
    case 0:
      console.log("January");
      break;
    case 1:
      console.log("February");
      break;
    case 2:
      console.log("March");
      break;
    case 3:
      console.log("April");
      break;
    case 4:
      console.log("May");
      break;
    case 5:
      console.log("June");
      break;
    case 6:
      console.log("July");
      break;
    case 7:
      console.log("August");
      break;
    case 8:
      console.log("September");
      break;
    case 9:
      console.log("October");
      break;
    case 10:
      console.log("November");
      break;
    case 11:
      console.log("December");
      break;
    default:
      console.log("Invalid month");
  }
}

function getDayName(date) {
  const dayNum = date.getDayName();

  switch (dayNum) {
    case 1:
      console.log("Monday");
      break;
    case 2:
      console.log("Tuesday");
      break;
    case 3:
      console.log("Wednesday");
      break;
    case 4:
      console.log("Thursday");
      break;
    case 5:
      console.log("Friday");
      break;
    case 6:
      console.log("Saturday");
      break;
    case 7:
      console.log("Sunday");
      break;
    default:
      console.log("Invalid day");
  }
}

getDayName("2012-10-04");
getMonthName(date);
```

- Shorter Way

```
const date = new Date();

function getDayAndMonthName(date) {
  const dateString = date.toLocaleDateString("en-GB", { dateStyle: "full" });
  const dayName = dateString.split(" ")[0];
  const monthName = dateString.split(" ")[2];

  return `day -> ${dayName}, month -> ${monthName}`;
}

console.log(getDayAndMonthName(date)); // day -> Thursday, month -> December
console.log(getDayAndMonthName(new Date("2014-08-05"))); // day -> Tuesday, month
-> August
console.log(getDayAndMonthName(new Date(2047, 5, 30))); // day -> Sunday, month ->
June
```

- Shorter way (without using split)

```
const date = new Date();

function getDayAndMonthName(date) {
  const dayString = date.toLocaleDateString("en-GB", { weekday: "long" });
  const monthString = date.toLocaleDateString("en-GB", { month: "long" });

  return `day -> ${dayString}, month -> ${monthString}`;
}

console.log(getDayAndMonthName(date)); // day -> Thursday, month -> December
console.log(getDayAndMonthName(new Date("2014-08-05"))); // day -> Tuesday, month
-> August
console.log(getDayAndMonthName(new Date(2047, 5, 30))); // day -> Sunday, month ->
June
```

Set methods with Date object

These methods override the current date and time

```
const date = new Date();

console.log(date);
// Thu Dec 26 2024 13:03:52 GMT+0530 (India Standard Time)

// date.setFullYear(2010); // override the current year i.e. 2024 with 2010
// console.log(date)

date.setYear(10);
console.log(date);
// Mon Dec 26 1910 13:03:45 GMT+0530 (India Standard Time)

date.setYear(100);
console.log(date);
// Sun Dec 26 0100 13:03:37 GMT+0553 (India Standard Time)

date.setYear(2000);
console.log(date);
// Tue Dec 26 2000 13:03:19 GMT+0530 (India Standard Time)

date.setFullYear(2047);
console.log(date);
// Thu Dec 26 2047 13:04:56 GMT+0530 (India Standard Time)

date.setMonth(0);
console.log(date);
// Sat Jan 26 2047 13:06:12 GMT+0530 (India Standard Time)

date.setMonth(12);
console.log(date);
// Sun Jan 26 2048 13:05:36 GMT+0530 (India Standard Time)
// Next year's january 26th
```

```
const date = new Date();

console.log(date);
// Thu Dec 26 2024 13:03:52 GMT+0530 (India Standard Time)
```

```
date.setDate(14);
console.log(date);
// Sat Dec 14 2024 13:08:42 GMT+0530 (India Standard Time)

date.setHours(4);
console.log(date);
// Sat Dec 14 2024 04:10:27 GMT+0530 (India Standard Time)

date.setMinutes(57);
console.log(date);
// Sat Dec 14 2024 04:57:44 GMT+0530 (India Standard Time)

date.setSeconds(41);
console.log(date);
// Sat Dec 14 2024 04:57:41 GMT+0530 (India Standard Time)

date.setMilliseconds(800);
console.log(date.getMilliseconds());
// 800
```

```
const date = new Date();

console.log(date);
// Thu Dec 26 2024 13:03:52 GMT+0530 (India Standard Time)

date.setTime(0); // in milliseconds
console.log(date);
// Thu Jan 01 1970 05:30:00 GMT+0530 (India Standard Time)

date.setTime(1234657894530); // in milliseconds
console.log(date);
// Sun Feb 15 2009 06:01:34 GMT+0530 (India Standard Time)
```

Similarly, we have UTC set methods:

- setUTCFullYear()
- setUTCMonth()
- setUTCDate()
- setUTCHours()
- setUTCMinutes()
- setUTCSeconds()
- setUTCMilliseconds()

It takes this all from our local timezone, so if we change the time zone in our computer, we will get the output here accordingly.

static methods, (these do not need `new` keyword to be called. Call them directly)

- Date.now()
- Date.parse()
- Date.UTC()

Date.parse()

```
const dateParse = Date.parse(
  "Sun Feb 15 2009 06:01:34 GMT+0530 (India Standard Time)"
);
console.log(dateParse);
// 1234657894000 (gives the millisecond value of this date and time string)

// We can also input ISO strings in the argument too (HHHH-MM-DDTHH:MM:SS.MSSZ)!

// This is a fixed value, so we can send it to others via API in this format.
Person in the receiving end can extract the date from this number
// eg.
const recoverDate = new Date(1234657894000);
console.log(recoverDate);
// Sun Feb 15 2009 06:01:34 GMT+0530 (India Standard Time)
```

Date.UTC()

String inputs not allowed

```
const dateUTC = Date.UTC(2022, 1, 4, 10, 25, 4);
console.log(dateUTC);
// 1643970304000

const recoverDate = new Date(1643970304000);
console.log(recoverDate);
// Fri Feb 04 2022 15:55:04 GMT+0530 (India Standard Time)
// time is set keeping 05:30:00 as base
```

Year is compulsory, others are optional

Date.now()

Gives the current date and time in milliseconds

```
const nowDate = Date.now();
console.log(nowDate);
// 1735200266492

const recoverDate = new Date(1735200266492);
console.log(recoverDate);
// Thu Dec 26 2024 13:34:26 GMT+0530 (India Standard Time)
```

date.valueOf()

`date.valueOf()` is same as `date.getTime()`, both return the date in `ms` value and both give the same value, so they are `===` to each other.

List of all date methods in JS

```js
const myDate = new Date();
console.log(myDate);

// Local get methods
console.log("Local Year", myDate.getFullYear());
console.log("Local Month", myDate.getMonth());
console.log("Local Date", myDate.getDate());
console.log("Local Day", myDate.getDay());
console.log("Local Hours", myDate.getHours());
console.log("Local Minutes", myDate.getMinutes());
console.log("Local Seconds", myDate.getSeconds());
console.log("Local Milliseconds", myDate.getMilliseconds());
console.log("Local Year - 1900", myDate.getYear()); // Deprecated

// UTC get methods
console.log("UTC Year", myDate.getUTCFullYear());
console.log("UTC Month", myDate.getUTCMonth());
console.log("UTC Date", myDate.getUTCDate());
console.log("UTC Day", myDate.getUTCDay());
console.log("UTC Hours", myDate.getUTCHours());
console.log("UTC Minutes", myDate.getUTCMinutes());
console.log("UTC Seconds", myDate.getUTCSeconds());
console.log("UTC Milliseconds", myDate.getUTCMilliseconds());

// Other get methods
console.log("Timestamp in Milliseconds", myDate.getTime());
console.log("Time Zone Offset in Minutes", myDate.getTimezoneOffset());

// to methods without arguments
console.log("toString: ", myDate.toString());
console.log("toUTCString: ", myDate.toUTCString());
console.log("toISOString: ", myDate.toISOString());
console.log("toJSON: ", myDate.toJSON());
console.log("toDateString: ", myDate.toDateString());
console.log("toTimeString: ", myDate.toTimeString());

// to methods with arguments
console.log("toLocaleString: ", myDate.toLocaleString());
console.log("toLocaleDateString: ", myDate.toLocaleDateString());
console.log("toLocaleTimeString: ", myDate.toLocaleTimeString());

// Local set methods
myDate.setFullYear(2014);
myDate.setMonth(10);
myDate.setDate(5);
myDate.setHours(20);
myDate.setMinutes(12);
myDate.setSeconds(10);
myDate.setMilliseconds(60);
myDate.setTime(1709802054158);
myDate.setYear(2024); // Deprecated

// UTC set methods
myDate.setUTCFullYear(2014);
myDate.setUTCMonth(10);
myDate.setUTCDate(5);
myDate.setUTCHours(20);
myDate.setUTCMinutes(12);
myDate.setUTCSeconds(10);
myDate.setUTCMilliseconds(60);

// static methods
console.log(Date.now());
console.log(Date.parse("04 Dec 1995 00:12:00 GMT"));
console.log(Date.UTC());

// other methods
console.log(myDate.valueOf());
```

Quiz

Memoization in JavaScript Explained in Hindi

Let us first look at this example where we are finding out the square root of a number but intentionally making the function heavy i.e. it takes a lot of time to execute.

```
function doHeavyCalculation(x) {
  // function to calculate square root of a number
  const startTime = Date.now();
  let currentTime = startTime;

  // blocking the code execution for 1 second i.e. 1000 ms and then giving the
answer (square root)
  while (startTime + 1000 > currentTime) {
    currentTime = Date.now();
    console.log("Calculating...", currentTime - startTime);
  }

  const result = +Math.sqrt(x).toFixed(3);
  return result;
}
```

Now in general there will be some functions where it will take a lot of time to get the answer and it may be called multiple times, due to which the code gets stuck during the function execution.

To solve this issue, we can do memoization i.e. caching the answer of the function upon its first execution and then storing it in a temporary memory (cache). We can now refer to this cache whenever we call the function. It will save us a lot of time.

```
const cache = {};

function doHeavyCalculation(x) {
  // function to calculate square root of a number

  if (cache[x]) return cache[x];
  const startTime = Date.now();
  let currentTime = startTime;

  // blocking the code execution for 1 second i.e. 1000 ms and then giving the
answer (square root)
  while (startTime + 1000 > currentTime) {
    currentTime = Date.now();
    console.log("Calculating...", currentTime - startTime);
  }

  const result = +Math.sqrt(x).toFixed(3);
  cache[x] = result;
  // x is key, result is value
  return result;
}
```

In this function, we have stored the result of the function in a cache object. If the function is called again with the same parameter, it will return the result from the cache object and it need not run the heavy function again.

But there is a problem here, the cache is declared globally, i.e. anyone can modify these cache functions and can access them. So we need to make it local to the function. This can be done using closures.

```
function getYourMemoizationFunction() {
  const cache = {};

  function doHeavyCalculation(x) {
    // function to calculate square root of a number

    if (cache[x]) return cache[x];
    const startTime = Date.now();
    let currentTime = startTime;

    // blocking the code execution for 1 second i.e. 1000 ms and then giving the
answer (square root)
    while (startTime + 1000 > currentTime) {
      currentTime = Date.now();
      console.log("Calculating...", currentTime - startTime);
    }

    const result = +Math.sqrt(x).toFixed(3);
    cache[x] = result;
    // x is key, result is value
    return result;
  }

  return doHeavyCalculation;
}

// getYourMemoizationFunction() will give us the code inside the inner function,
to access the inner function, see the below code

const memoized = getYourMemoizationFunction();

console.log(memoized(17));

console.dir(memoized); // look inside the scopes to see the closure formed
```

Now we have made sure that the inner function forms a closure with the variables of the outer function so that the outer cache variable can not be accessed globally and altered now.

Currying in JavaScript Explained in Hindi

Currying is a technique in JavaScript where a function with multiple arguments is transformed into a `sequence of functions, each taking a single argument`. For example, instead of calling `add(2, 3)`, you can call it like `add(2)(3)`. It helps in creating reusable and more modular code by allowing partial application of functions.

```
function multiplyNormal(x, y, z) {
  return x * y * z;
}

multiplyNormal(2, 3, 5);

function curryMultiply(x) {
  return function (y) {
    return function (z) {
      return x * y * z;
    };
  };
}

curryMultiply(2)(3)(5);
```

Now what if we take infinite arguments in currying??

Infinite Currying

Uses recursion

```
function infiniteCurryMultiply(x) {
  return function (y) {
    if (y) {
      return infiniteCurryMultiply(x * y);
      // recursion happening here
    }
    return x;
  };
}

console.log(infiniteCurryMultiply(2)(3)(5)(7)());
// an empty () is necessary to exit the recursive call statement

// 210 -> answer
```

Now there is no practical application of currying in real life in general but we can use it in some cases where we apply currying:

1. Using closures

Make one generalized function and keep calling its variants

```
function multiplyByN(x) {
  return function (y) {
    return x * y;
  };
}

// Now by using this, we can implement, multiply M * N where M will the the number
we want N to be multiplied with

const mulBy3 = multiplyByN(3);

console.log(mulBy3(7));
// multiply 7 with 3
// 21

const mulBy8 = multiplyByN(8);

console.log(mulBy8(9));
// multiply 9 by 8
// 72

console.dir(mulBy3);
```

2. Using Bind() method

```
function multiply(a, b) {
  return a * b;
}

const multiplyBy5 = multiply.bind(this, 5);
// here bind method helps the function multiply to retain the a argument which is
5 here, this will be fixed
// we can also use null here

console.log(multiplyBy5(15));
// we can now use this to give different b argument values to make the number
multiply by 5
```

In short, Currying and Memoization are applications of closures.

What is Debouncing in JavaScript? | JavaScript for ProCodrrs

Debouncing is a technique in JavaScript where a function is delayed for a certain amount of time before being executed. This is useful when we want to avoid multiple calls to the same function.

eg. when we enter input to search something, it makes a call to the backend for every keystroke, this is called `spamming`. So we want to delay the call to the backend for a certain amount of time. So that we don't make too many calls to the backend. This is where we will use debouncing.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Debouncing in JS</title>
    <script src="./script.js" defer></script>
  </head>
  <body>
    <h1>Debouncing in JS</h1>
    <input type="text" id="input" />
  </body>
</html>
```

```
const inputElement = document.querySelector("#input");
const debounce = (functionReceived, waitTime) => {
  let timerId;
  return (...args) => {
    clearTimeout(timerId);
    timerId = setTimeout(() => functionReceived(...args), waitTime);
  };
};

// Now we know that in each setTimeout, there is a timer id,
// so we have made initially a variable called timerId to keep track of it
// then we do ...args to accept multiple arguments (not just e)
// when we input something, the debounce function is called wih wait time of 1000
ms i.e. 1s
// inside it we first clear any ongoing set Timeout, this will make sure that when
we are typing or giving input, set timeout will not run and the callApi function
will not execute, which gives us our output
// When we stop typing, or the gap reaches 11 sec, clear timeout cannot run here
and hence the settimeout will be called and callApi is called after 1s (wait time
=> 1000ms) when we stop typing. This will now give us our output

const callApi = (e) => {
  console.log(`Calling API:`, e.target.value);
};

const debouncedCallApi = debounce(callApi, 1000);

inputElement.addEventListener("input", debouncedCallApi);

// event listener calls the debounced call api function which tells us which
function to receive and what wait time should be given
```

Please do check out debouncing in React at `23.39` from this video, there we have used useState for state change and useMemo for debouncing.

I have not included it here!!

What is Throttling in JavaScript? | JavaScript for ProCodrrs

Throttling is mostly used in mousemove, and window resizing.

It works mostly like debouncing, we can also use it in input fields too though we mostly use debouncing for it!!

If debouncing is like a traffic signal which allows a certain number of chars to be printed at a time. Then throttling is like a speed limiter which slows down the printing of chars at a time.

In both cases, the number of events fired, reduces significantly.

Throttle with mousemove

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Throttling in JS</title>
    <script src="./script.js" defer></script>
  </head>
  <body>
    <h1>Throttling in JS</h1>
    <input type="text" id="input" autocomplete="off" />
    <h2>Count <span>0</span></h2>
  </body>
</html>
```

```js
const inputElement = document.querySelector("#input");
const span = document.querySelector("h2 span");

const callApi = function (e) {
  console.log("Calling API", e.target.value);
};

inputElement.addEventListener("input", callApi);

const updateNumber = throttle(() => {
  span.innerText = ++span.innerText;
}, 100);

document.addEventListener("mousemove", () => {
  updateNumber();
});

function throttle(func, delay = 1000) {
  let timerId = null;
  return (...args) => {
    if (timerId) return;
    timerId = setTimeout(() => {
      timerId = null; // so that other timer ids get cancelled when one is running
      func(...args);
    }, delay);
  };
}
```

Throttle with input field

```
const inputElement = document.querySelector("#input");
const span = document.querySelector("h2 span");

const callApi = function (value) {
  console.log("Calling API", value);
};

const logInput = throttle(callApi, 1000);

inputElement.addEventListener("input", (e) => {
  logInput.call({ name: "KSD", age: 23 }, e.target.value);
  // taking care of 'this' keyword context
});

function throttle(func, delay = 1000) {
  let firstCall = true;
  let timerId = null;
  let lastArgs = [];
  return (...args) => {
    console.log(this);
    lastArgs = args;
    if (firstCall) {
      // making sure that initial call does not give any delay, but calls after
that can give delay
      func.apply(this, lastArgs); // lastArgs getting spread automatically
      firstCall = false;
      return
    }
    if (timerId) return; // if the delay is happening, cancel the other delays
    timerId = setTimeout(() => {
      timerId = null; // so that other timer ids get cancelled when one is running
      func.apply(this, lastArgs); // lastArgs getting spread automatically giving
us the result
    }, delay);
  };
}
```

Master Debugging and Become a Pro Developer | Catch Bugs Instantly | Advanced JavaScript

Shortcut to detect any event and debug it:

DevTools -> Sources -> Event Listener Breakpoints -> Mouse (here items are getting removed on click) -> click

Then perform that event, this will highlight that code bit which is performing that event in debugger.

Use `watch` in sources to track the variable value in devtools.

Tip -> Do debugging in incognito mode because in normal mode, chrome extensions may hamper the debugging process.

Practice some debugging problems!!