50 drops of JavaScript

50 useful, powerful, joyful JavaScript functions

The free, opensource book

ROBERTO BUTTI

Table of Contents

Welcome to 50 drops of JavaScript

The reason why

This book collects 50 useful, unknown, underrated JavaScript functions or stuff discovered, used, and learned during JavaScript daily use.

Using JavaScript frameworks/libraries daily (like React, Vue, Angular), sometimes the perception of the power of the language and the basic functionalities provided by the JavaScript core could be lost. I see that usually, I used to look at the framework documentation or look for a package in Npm for the system, array, and string functions instead of using core functionalities provided by the language.

While I wrote this book, I also wrote some scripts to better understand the behavior of the functions. You can find these examples here: https://github.com/roberto-butti/50-drops-of-javascript in the *examples* directory.

Requirements

The code used in this book is tested with **NodeJs version 18 (LTS)**. Node.js is an open-source, cross-platform JavaScript runtime environment, and you can obtain Node.js on the official https://nodejs.org/ website.

Continuous release

I was thinking to print this book, but I think that is not so eco-friendly and a book about development could be improved daily in terms of spellchecking and the content and examples. So, I expect to adopt the same approach in the software with the CI/CD, with a continuous release of the book.

Releases:

- (WIP) release 1.0.0 (WIP not yet released): writing... ;

So, if you have any feedback, or you want to suggest some corrections, feel free to open an issue here: https://github.com/roberto-butti/50-drops-of-javascript/issues

Where to find this book

This book is available for download for free here: https://github.com/roberto-butti/50-drops-of-javascript/

Thanks to...

Thanks to all the Open Source community.

License

System

The chapter will cover functions about "system" functionalities (for example how to retrieve the operating system version).

Info from CPU: `os.cpus()`

The `os` module allows you to retrieve some relevant information about the environment used for running your *Node.js* script or application.

```
import { cpus } from 'os'
const c = cpus()
```

The `os` module provides you with some methods. One of them is the `cpus()` that returns the list of CPUs available in your environment. For each element of the list (each element is one CPU), you have the `model` attribute that has the CPU model name (for example "Apple M1 Pro") and the `speed` attribute, an integer for showing the speed of the CPU. Then you have also `times` attribute for the times (in milliseconds) that the CPU has spent in these modes: `user`, `nice` (valid only for Posix systems, in Windows this value is always 0), `sys`, `idle`, `irq`. So, you can walk through the list of CPUs:

```
import { cpus } from 'os'
cpus().forEach(cpu => {
  console.info(cpu.model + ' speed: ' + cpu.speed)
  console.table(cpu.times)
})
```

Info from Operating System: os.version()

The `os` module has the `version`()` method to return the version of the Operating System in use.

```
import { version } from 'os'
const version = version()
console.log(v)
```

The `version()` method returns a string with the full name of the Operating System version, like this:

```
Darwin Kernel Version 21.6.0: Wed Aug 10 14:28:23 PDT 2022;
root:xnu-8020.141.5~2/RELEASE_ARM64_T6000
```

Info for the current user: os.userInfo()

The `os` module has the `userInfo()` method to return information on the current system user.

```
import { userInfo } from 'os'
const u = userInfo()
```

The `userInfo()` method returns an object with attributes:

- `uid`: the user identifier (integer)
- `gid`: the group identifier (integer)
- `username`: the username (string)
- `homedir`: the user's home directory (string)
- `shell`: the user's shell (string)

Get the amount of free memory in bytes (as integer): os.freemem()

The `os` module has the `freemem()` method to return information on the free memory available in the system. The `freemem()` function returns an integer and it represents the bytes. If you need megabytes or kilobytes you have to convert it.

```
import { freemem } from 'os'
const mem = freemem()
console.log('The amount of free memory is %d bytes', mem)
```

Access to environment variables: process.env

The `process` global module has the `env` attribute to return information on the environment variable. The `process.env` attribute contains an object and it represents the list of environment variables. Each attribute is an environment variable. The `process` module is a global module, which means that you don't have to import the process module manually, but it is automatically available in your code.

For retrieving the environment variables object:

```
console.log(process.env)
```

If you want to access a specific environment variable via the name (for example to the `PATH` environment variable):

```
console.log(process.env.PATH)
```

If you have to access dynamically to an environment variable, you can use the square brackets:

```
const envVarName = 'PATH'
if (envVarName in process.env) {
  console.log(process.env[envVarName])
} else {
  console.log('no %s defined', envVarName)
}
```

If you want to walk through all the environment variables you can iterate on the object attributes via `Object.keys()` method:

```
Object.keys(process.env).forEach(function (key, index) {
  console.log(key, index, process.env[key])
})
```

Get the amount of total memory available in bytes (as integer): os.totalmem()

The `os` module has the `totalmem()` method to return information on the total memory available in the system. The `totalmem()` function returns an integer and it represents the bytes. If you need megabytes or kilobytes you have to convert it.

```
import { totalmem } from 'os'
const mem = totalmem()
console.log(
  'Hi, the total memory is %d gigabytes',
  mem / 1024 / 1024 / 1024
)
```

The load average of CPU: `os.loadavg()`

The `os` module has the `loadavg()` method to return the "load" measurement information about the CPU usage of the system. The "load" measurement is calculated by the number of processes that are being executed by the CPU or in a 'waiting' state.

The `loadavg()` function returns an array with 3 float numbers. These three numbers represents the average system load calculated over a given period of 1, 5 and 15 minutes

```
import { loadavg } from 'os'
const la = loadavg()
console.log(
  la[0], // last minute
  la[1], // last 5 minutes
  la[2]  // last 15 minutes
)
// it returns: 1.59619140625 2.42822265625 2.60400390625
```

> *On Windows machine this functionality is not available, the method returns an array with 0 values, like `[0, 0, 0]`*

Network interfaces information: `os.networkInterfaces()`

The `os` module has the `networkInterfaces()` method to return the object that shows the information about the network interfaces available on the system. The object has multiple interfaces. Each interface has an identifier used as an object key attribute like `lo0`, `en0`, `utun0`, `utun1` ... etc. Each interface has an array of addresses. Each address is an object like this one:

```
{
    address: '127.0.0.1',
    netmask: '255.0.0.0',
    family: 4,
    mac: '00:00:00:00:00:00',
    internal: true,
    cidr: '127.0.0.1/8'
}
```

Where `address`, `netmask`, `mac`, `cidr` are the address (ip address, mac addres) and netmask. The `family` attribute identifies the type of the ip protocol. The `internal` attribute is a boolean value, `true` if the address is remotely accessible.

```
import { networkInterfaces } from 'os'
const ni = networkInterfaces()

Object.keys(ni).forEach(function (key, index) {
  // each interfaces has an array
  ni[key].forEach(function (element, index) {
    // selecting only family === 4 (ipv4 interfaces)
    if (element.family === 4) {
      console.log('IP ADDRESS: ' + element.address) // IPv4 address
    }
  })
})
```

Stop the script execution: process.exit()

The `process` global module has the `exit()` method to stop the current execution and return to the shell, or in general to the environment where the script was launched. All the processes can return a status code to the shell. By convention, if the execution ends correctly a status code value equal to 0 is returned. In case of an error, the process can return a not 0 status code value. So if you want to stop the execution of the script and return a status code equal to 99 you can use `process.exit(99)` like the example:

```
console.log('Executing ...')
process.exit(99)
console.log('Never executed')
```

Executing this snippet, a `99` status code error will be returned to the shell. For example, once the script execution is terminated, you can test the shell environment variable `$?` that contains the status code of the last process executed. If you execute in a shell (`bash` or `zsh` or some other shell) the node script and then try to show the last status code via `echo $?`:

```
node examples/01-09_process-exit.mjs
echo $?
```

You will see the `99` value will be shown in the terminal (because the JS script returns with `process.exit(99)`).

Get temporary directory: `os.tmpdir()`

If you need to store temporary a file in your script, you can save the temporary file in the default operating system temporary directory. You can retrieve the current operating system temporary directory via `tmpdir()` function:

```
import { tmpdir } from 'os'
console.log('Temporary directory: %s', tmpdir())
```

Get monotonic time: `performance.now()`

If you have to track execution time or the time spent in a script or a long-running application, you have to keep in mind that while you are executing your script and your logic the date time of the operating system could change because the user adjusts the hour, or the date time is adjusted automatically by the NTP system. So if you want to track the millisecond spent executing a logic you need a monotonic time. In JavaScript (Node.js) you have a kind of stopwatch that is started at the beginning of the execution of the script. This timer is used for measuring performance because is a relative metric and it isn't affected by system clock changes.

```
// saving the timer in startTime
let startTime = performance.now()
console.log('Starting')
// calculating the time by difference performance.now() - startTime
console.log(performance.now() - startTime, startTime)
```

Array

The chapter will cover functions about "array" functionalities (for example how to filter, map and reduce arrays).

Create a string from an array: `join()`

If you want to generate a string, joining all the array elements, you can use the `join()` method. By default, the character is used as separator. If you want to use a specific separator you can call the join('-') method with the separator needed as the first argument.

```
const elements = ['kiwi', 'strawberry', 'lemon']

console.log(elements.join())
// expected output: "kiwi,strawberry,lemon"
console.log(elements.join('-'))
// expected output: "kiwi-strawberry-lemon"
```

Check if the array includes a certain value: `includes()`

If you want to check if the array includes a specific value, you can use `include()` method. The first mandatory input argument of `include()` method is the value of the element you want to check.

```
const elements = ['kiwi', 'strawberry', 'lemon']

console.log(elements.includes('strawberry'))
// Does the array include the 'strawberry'?
// expected output: true
```

If you want to check if the value is included starting from a specific index you can use the second input parameter for setting the index (remember to start from 0 for counting the position).

```
const elements = ['kiwi', 'strawberry', 'lemon']

console.log(elements.includes('strawberry'), 2)
// Does the array starting from index 2, include the 'strawberry'?
// expected output: false
```

Check if a property exists in the object or array: `in`

If you have an object and you want to check if a property is present you can use the `in` operator.

```
const fruits = {
  strawberry: 'Strawberry',
  kiwi: 'Kiwi',
  lemon: 'Lemon'
}
// for checking the presence of an property you can use in operator
if ('kiwi' in fruits) {
  console.log('Kiwi is present')
}
// you can use in the short form
console.log('kiwi' in fruits ? 'Yes! Kiwi' : 'ther is no kiwi here')
```

I think the `in` operator is one of the best ways to check the presence of a property because, for example, if you want to check if the value is `undefined`, you can't distinguish the cases if the property doesn't exist or the property exists and has an undefined value:

```
const fruits = {
  strawberry: 'Strawberry',
  kiwi: 'Kiwi',
  lemon: 'Lemon'
}

fruits.something = undefined
if (fruits.somethingelse === undefined) {
    console.log('Somethingelse property does not exist')
}
if (fruits.something === undefined) {
  console.log('Something property exists but the value is undefined')
}
```

The `in` operator works also with the arrays. With the arrays, you can check the presence of the numeric index:

```
const fruitsArray = []
fruitsArray.push('Strawberry')
fruitsArray.push('Kiwi')
fruitsArray.push('Lemon')
console.dir(fruitsArray)
console.log(2 in fruitsArray) // true
console.log(3 in fruitsArray) // false
```

Concatenate two (or more) arrays: `concat()`

You can concatenate (append) two or more arrays with the `concat()` method.

```
const array1 = ['a', 'b', 'c']
const array2 = ['d', 'e', 'f']
console.dir(array1.concat(array2)) // ['a', 'b', 'c', 'd', 'e', 'f']
```

you can concatenate more arrays:

```
const array1 = ['a', 'b', 'c']
const array2 = ['d', 'e', 'f']
const array3 = ['g', 'h']
console.dir(array1.concat(array2, array3))
console.dir([].concat(array1, array2, array3))
```

Concat arrays via destructuring

For concatenating or merging two or more arrays, you can use the destructuring technique.

With this technique essentially you are going to unpack the arrays into a list of distinct variables, and use this list to recreate a new array with square brackets.

```
const array1 = ['a', 'b', 'c']
const array2 = ['d', 'e', 'f']
console.dir([...array1, ...array2])
// [ 'a', 'b', 'c', 'd', 'e', 'f' ]
```

You can use this technique with more than 2 arrays:

```
const array1 = ['a', 'b', 'c']
const array2 = ['d', 'e', 'f']
const array3 = ['g', 'h']
console.dir([...array1, ...array2, ...array3])
// [ 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h' ]
```

The order you are using the `[...array1, ...array2, ...array3]` is important because determines the order of the items for the new array

```
const array1 = ['a', 'b', 'c']
const array2 = ['d', 'e', 'f']
const array3 = ['g', 'h']
console.dir([...array3, ...array2, ...array1])
// [ 'g', 'h', 'd', 'e', 'f', 'a', 'b', 'c' ]
```

Remove duplicate values in an array via `Set()`

If you want to remove duplicates from an array there is no specific array method for removing duplicate elements but you can achieve this using destructuring, the `Set()` function and recreating the array. It means that the original array will not change.

```
const array = ['a', 'b', 'c', 'b', 'd', 'e', 'a']
const unique = [...new Set(array)]
// [ 'a', 'b', 'c', 'd', 'e' ]
```

Generate and fill a new array `fill()`

The `fill()` method sets a specific value to the elements of the array.

If you have an array with some values:

```
const array = ['a', 'b', 'c', 'b', 'd', 'e', 'a']
```

and you want to replace/fill the elements with the value `0`, you can call `fill()` method on the array, using `0` as a parameter:

```
array.fill(0)
// [ 0, 0, 0, 0, 0, 0, 0 ]
```

As you can see the `fill()` method changes the content of the original array. The size of the array will be the same as the original one.

The `fill()` method has two more optional parameters about the starting index to apply the changes and the ending index. For example, if you want to replace values starting with the second element (remember 0 index array):

```
const array2 = ['a', 'b', 'c', 'b', 'd', 'e', 'a']
array2.fill(
    0, // the value to fill
    1 // the index where to start
)
console.dir(array2)
// [ 'a', 0, 0, 0, 0, 0, 0 ]
```

you can also set the last index to apply the replacement. For example, if you want to replace elements from index 1 (the second element) until index 3 (the third element included):

```
const array3 = ['a', 'b', 'c', 'b', 'd', 'e', 'a']
array3.fill(
    0, // the value to fill
    1 // the index where to start
    3 // the index where to end
)
console.dir(array3)
// [ 'a', 0, 0, 'b', 'd', 'e', 'a' ]
```

If you want to fill/replace the last element you can use the negative index:

```
const array4 = ['a', 'b', 'c', 'b', 'd', 'e', 'a']
array4.fill(0, -1) // last element
console.dir(array4)
// [ 'a', 0, 0, 'b', 'd', 'e', 'a' ]
array4.fill(0, -2) // last 2 elements
console.dir(array4)
// [ 'a', 'b', 'c', 'b', 'd', 0, 0 ]
```

Filtering elements: `filter()`

With the arrays, you have the `filter()` method to filter the elements of the array with a specific condition. For example with an array of numbers, you want to retrieve only the element greater than 50. The `filter()` doesn't change the original array. It returns a new array with the elements that satisfy the condition.

```
const numbers = [3, 75, 42, 13, 69]
const resultNumbers = numbers.filter(
    number => number >= 50
)
console.dir(resultNumbers)
// output: Array [ 75, 69 ]
```

If the element is an object you can filter for the attribute of the object. For example, if you have objects like this one:

```
{ product: 'Desk', price: 200, active: true }
```

and you have an array of elements, and you want to filter and retrieve the object with a price greater (or equal) than 150:

```
const elements = [
  { product: 'Desk', price: 200, active: true },
  { product: 'Chair', price: 100, active: true },
  { product: 'Door', price: 300, active: false },
  { product: 'Bookcase', price: 150, active: true },
  { product: 'Door', price: 100, active: true }
]

const resultElements = elements.filter(
    element => element.price >= 150
)
console.dir(resultElements)
```

the result is:

```
Array [
  { product: 'Desk', price: 200, active: true },
  { product: 'Door', price: 300, active: false },
  { product: 'Bookcase', price: 150, active: true }
]
```

you can create a more complex conditions like:

```
const resultActiveElements = elements.filter(
    element =>
    (element.active && element.price >= 150)
)
```

where the result is:

```
Array [
  { product: 'Desk', price: 200, active: true },
  { product: 'Bookcase', price: 150, active: true }
]
```