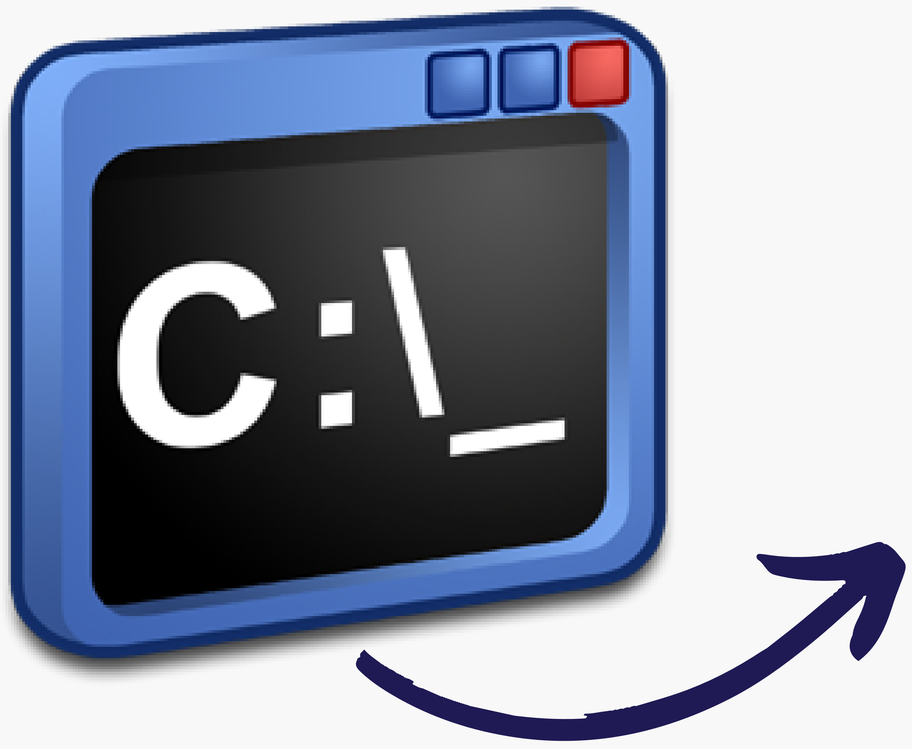
techts

FUN WITH CONSOLE.LOG()



Hassan Mujtaba | Full Stack Developer

@javascriptwithhassan

techis

TABLES

The **console.table()** method prints objects/ arrays as a neatly formatted tables.

```
console.table({
  'Time Stamp': new Date().getTime(),
  OS: navigator['platform'],
  Browser: navigator['appCodeName'],
  Language: navigator['language'],
});
```

(index)	Values
Time Stamp	1662901875516
OS	Linux x86_64
Browser	Mozilla
Language	en-GB

Hassan Mujtaba | Full Stack Developer
@javascriptwithhassan

techis

GROUPS

Group related console statements together with collapsible sections, using **console.group().**

```
console.group('URL Info');
console.log('Protocol', window.location.protocol);
console.log('Host', window.origin);
console.log('Path', window.location.pathname);
console.groupCollapsed('Meta Info');
console.log('Date Fetched', new Date().getTime());
console.log('OS', navigator['platform']);
console.log('Browser', navigator['appCodeName']);
console.log('Language', navigator['language']);
console.groupEnd();
console.groupEnd();
```

```
▼ URL Info
    Protocol https:
    Host https://developer.mozilla.org
    Path /en-US/docs/Web/API/Console/group
  ▼ Meta Info
      Date Fetched 1662901104077
      OS Linux x86_64
      Browser Mozilla
      Language en-GB
```

Hassan Mujtaba | Full Stack Developer

@javascriptwithhassan

techis

STYLED LOGS

It's possible to style your log outputs with some basic CSS, such as colors, fonts, text styles, and sizes. Note that browser support for this is quite variable.

For example, try running the following:

```
console.log('%cHello World!', 'color: #f709bb; font-family: sans-serif; text-decoration: underline;');
```

You should get the following output:

```
» console.log('%cHello World!', 'color: #f709bb; font-family:
  cursive; text-decoration: underline; font-size: 1.5rem; ');
  Hello World!                                  debugger eval code:1:9
```

Hassan Mujtaba | Full Stack Developer

@javascriptwithhassan

techts

Pretty cool, huh? Well, there's a lot more you can do too! Maybe change the font, style, and background color, and add some shadows and some curves...

```
> console.log('%cDinosaurs are Awesome 🦖', 'color:#1cce69;
  background: #3d09bf; font-size: 1.5rem; padding: 0.15rem
  0.25rem; margin: 1rem; font-family: Helvetica; border: 2px solid
  #1cce69; border-radius: 4px;font-weight: bold; text-shadow: 1px
  1px 1px #0a0121; font-style: italic; ');
                                                        VM1028:1
```

Dinosaurs are Awesome 🦖

Here's something similar I'm using in a developer dashboard, the code is here

Dashy 4.2.0 🚀

Auth Handler VM76:27
Incorrect Credentials

Auth Handler VM76:19
Logged in Succesfully

VM76:7

⚠ Warning ⚠

Selected config not found

This is likely not an issue with Dashy, but rather your configuration. If you think it is a bug, please open a ticket on GitHub: https://git.io/JukXk

Hassan Mujtaba | Full Stack Developer

@javascriptwithhassan

techis

TIME

The **console.table()** method prints objects/ arrays as a neatly formatted tables.

```
console.table({
  'Time Stamp': new Date().getTime(),
  OS: navigator['platform'],
  Browser: navigator['appCodeName'],
  Language: navigator['language'],
});
```

(index)	Values
Time Stamp	1662901875516
OS	Linux x86_64
Browser	Mozilla
Language	en-GB

Hassan Mujtaba | Full Stack Developer
@javascriptwithhassan

techis

GROUPS

Group related console statements together with collapsible sections, using **console.group()**.

The following example will output an open section, containing some info

```
console.group('URL Info');
console.log('Protocol', window.location.protocol);
console.log('Host', window.origin);
console.log('Path', window.location.pathname);
console.groupCollapsed('Meta Info');
console.log('Date Fetched', new Date().getTime());
console.log('OS', navigator['platform']);
console.log('Browser', navigator['appCodeName']);
console.log('Language', navigator['language']);
console.groupEnd();
console.groupEnd();
```

```
▼ URL Info
    Protocol https:
    Host https://developer.mozilla.org
    Path /en-US/docs/Web/API/Console/group
  ▼ Meta Info
      Date Fetched 1662901104077
      OS Linux x86_64
      Browser Mozilla
      Language en-GB
```

Hassan Mujtaba | Full Stack Developer

@javascriptwithhassan

techis

ASSERT

You may only want to log into the console if an error occurs, or if a specific condition is true or false. This can be done using **console.assert()**, which won't log anything to the console unless the first parameter is false.

```
const errorMsg = 'the # is not even';
for (let num = 2; num ⩽ 5; num++) {
  console.log(`the # is ${num}`);
  console.assert(num % 2 ═══ 0, { num }, errorMsg);
}
```

```
» const errorMsg = 'the # is not even';
  for (let num = 2; num <= 5; num++) {
    console.log(`the # is ${num}`);
    console.assert(num % 2 === 0, { num }, errorMsg);
  }
  the # is 2                                    debugger eval code:3:11
  the # is 3                                    debugger eval code:3:11
  Assertion failed: ▸ Object { num: 3 }         debugger eval code:4:11
  the # is not even
  the # is 4                                    debugger eval code:3:11
  the # is 5                                    debugger eval code:3:11
  Assertion failed: ▸ Object { num: 5 }         debugger eval code:4:11
  the # is not even
```

Hassan Mujtaba | Full Stack Developer

@javascriptwithhassan

techis

COUNT

Ever find yourself manually incrementing a number for logging? **console.count()** is helpful for keeping track of how many times something was executed, or how often a block of code was entered.

```
const numbers = [1, 2, 3, 30, 69, 120, 240, 420];
numbers.forEach((name) ⇒ {
  console.count();
});
```

```
const names = ['Alicia', 'Oliver', 'G', 'Steven', 'Bl'];

const handleShortName = () => {
  console.count('name-too-short');
  // Do something else...
}

names.forEach((name) => {
  if (name.length <= 3) {
    handleShortName();
  }
  console.count('name-checked');
});
```

```
name-checked: 1                    debugger eval code:12:11
name-checked: 2                    debugger eval code:12:11
name-too-short: 1                   debugger eval code:4:11
name-checked: 3                    debugger eval code:12:11
name-checked: 4                    debugger eval code:12:11
name-too-short: 2                   debugger eval code:4:11
name-checked: 5                    debugger eval code:12:11
```

Hassan Mujtaba | Full Stack Developer

@javascriptwithhassan

techts

Instead of passing in a label, if you use a value, then you'll have a separate counter for each condition's value. For example:

```
console.count(NaN); // NaN: 1
console.count(NaN + 3); // NaN: 2
console.count(1 / 0); // Infinity: 1
console.count(String(1 / 0)); // Infinity: 2
```

Hassan Mujtaba | Full Stack Developer
@javascriptwithhassan

techis

TRACE

In JavaScript, we're often working with deeply nested methods and objects. You can use **console.trace()** to traverse through the stack trace, and output which methods were called to get to a certain point.

```
» checkCredentials();
  console.trace()                          debugger eval code:2:10
    redirectToProfile debugger eval code:2
    logInUser         debugger eval code:6
    credentialsValid  debugger eval code:10
    checkCredentials  debugger eval code:14
    <anonymous>       debugger eval code:1
← undefined
```

You can optionally pass data to also be outputted along with the stacktrace.

Hassan Mujtaba | Full Stack Developer

@javascriptwithhassan

techis

DIR

If your logging a large object to the console, it may become hard to read. The **console.dir()** method will format it in an expandable tree structure.

The following is an example of a directory-style console output:

```
» console.dir(things)
▼ Object { humans: (3) […], dinosaurs: debugger eval code:1:9
(2) […], unicorns: (1) […] }
  ▶ dinosaurs: Array [ {…}, {…} ]
  ▼ humans: Array(3) [ {…}, {…}, {…}
  ]
    ▶ 0: Object { name: "Bob",
    brainCellCount: 100, faveDrinks:
    (2) […] }
    ▶ 1: Object { name: "Mary",
    brainCellCount: 120, faveDrinks:
    (2) […] }
    ▼ 2: Object { name: "Alicia",
    brainCellCount: 2, faveDrinks:
    (4) […] }
      brainCellCount: 2
      ▼ faveDrinks: Array(4) [
      "Coke", "beer", "coffee", … ]
          0: "Coke"
          1: "beer"
          2: "coffee"
          3: "RedBull"
          length: 4
        ▶ <prototype>: Array []
        name: "Alicia"
      ▶ <prototype>: Object { … }
      length: 3
    ▶ <prototype>: Array []
  ▶ unicorns: Array [ {…} ]
  ▶ <prototype>: Object { … }
```

You can also print XML or HTML based trees in a similar way, by using **console.dirxml().**

Hassan Mujtaba | Full Stack Developer

@javascriptwithhassan

techts

DEBUG

You may have some logging set up within your app, that you rely on during development, but don't wish the user to see. Replacing log statements with **console.debug()** will do just this, it functions in exactly the same way as **console.log** but will be stripped out by most build systems, or disabled when running in production mode.

Hassan Mujtaba | Full Stack Developer
@javascriptwithhassan

techis

LOG LEVELS

You may have noticed that there are several filters in the browser console (info, warnings, and error), that allows you to change the verbosity of logged data. To make use of these filters, just switch out log statements with one of the following:

- console.info() - Informational messages for logging purposes, commonly includes a small "i" and/or a blue background
- console.warn() - Warnings / non-critical errors, commonly includes a triangular exclamation mark and/or yellow background
- console.error() - Errors that may affect the functionality, commonly includes a circular exclamation mark and/or red background

Hassan Mujtaba | Full Stack Developer

@javascriptwithhassan

techis

MULTI-VALUE LOGS

Most functions on the **console** object will accept multiple parameters, so you can add labels to your output, or print multiple data points at a time, for example: **console.log('User: ', user.name);**

But an easier approach for printing multiple, labeled values, is to make use of **object deconstructing.** For example, if you had three variables (e.g. **x**, **y** and **z**), you could log them as an object by surrounding them in curly braces, so that each variables name and value is outputted - **like console.log({ x, y, z });**

```
» const protocol = window.location.protocol;
  const date = new Date().getTime();
  const os = navigator['platform'];
  const browser = navigator['appCodeName'];
  const locale = navigator['language'];
← undefined
» console.log({ protocol, date, os, browser, locale });
  ▼ Object { protocol: "https:", date:    debugger eval code:1:9
  1662921066827, os: "Linux x86_64",
  browser: "Mozilla", locale: "en-GB" }
      browser: "Mozilla"
      date: 1662921066827
      locale: "en-GB"
      os: "Linux x86_64"
      protocol: "https:"
```

Hassan Mujtaba | Full Stack Developer

@javascriptwithhassan

techts

LOG STRING FORMATS

If you need to build formatted strings to output, you can do this with C-style printf using format specifiers.

The following specifiers are supported:

- **%s** - String or any other type converted to a string
- **%d** / %i - Integer
- **%f** - Float
- **%o** - Use optimal formatting
- **%O** - Use default formatting
- **%c** - Use custom formatting

```
1  console.log('Hello %s, welcome to the year %d!', 'Alicia', new Date().getFullYear());
2  // Hello Alicia, welcome to the year 2022!
```

Hassan Mujtaba | Full Stack Developer

@javascriptwithhassan

techis

CLEAR

Finally, when you're looking for output from an event, you might want to get rid of everything logged to the console when the page first loaded. This can be done with **console.clear(),** which will clear all content, but not reset any data.

It's usually also possible to clear the console by clicking the Bin icon, as well as to search through it using the Filter text input.

Hassan Mujtaba | Full Stack Developer
@javascriptwithhassan

techts

SPECIAL BROWSER METHODS

When running code directly in the browser's console, you'll have access to a couple of short-hand methods, which are super useful for debugging, automation and testing.

The most useful of which are:

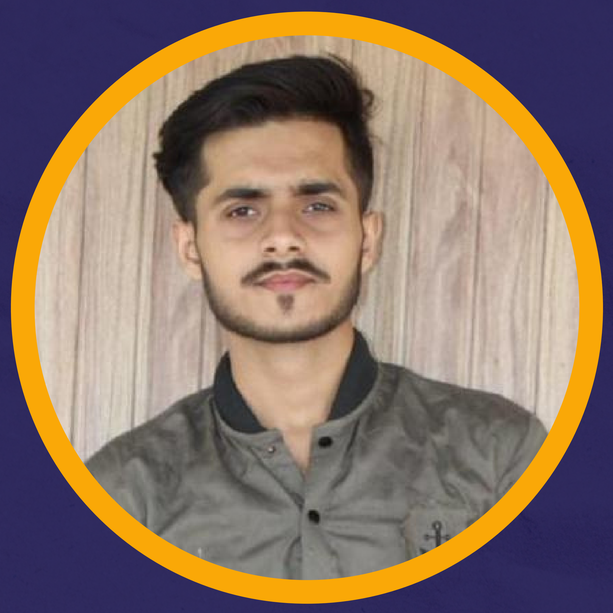
- $() - Short-hand for Document.querySelector() (to select DOM elements, jQuery-style!)
- $$() - Same as above, but for selectAll for when returning multiple elements in an array
- $_ - Returns the value of the last evaluated expression
- $0 - Returns the most recently selected DOM element (in the inspector)
- $1...$4 can also be used to grab previously selected UI elements

Hassan Mujtaba | Full Stack Developer

@javascriptwithhassan

Do you find it helpful?

Let me know in the comment section



Hassan Mujtaba | Full Stack Developer

Follow for more tips and tricks related to web development