How do I create a GUID / UUID?

Asked 16 years, 5 months ago Modified 3 months ago Viewed 3.2m times

5463

How do I create GUIDs (globally-unique identifiers) in JavaScript? The GUID / UUID should be at least 32 characters and should stay in the ASCII range to avoid trouble when passing them around.

I'm not sure what routines are available on all browsers, how "random" and seeded the built-in random number generator is, etc.

javascript **guid** **uuid**

Share Improve this question Follow

edited Jul 24, 2022 at 23:56

community wiki
13 revs, 8 users 25%
Jason Cohen

40 GUIDs when repesented as as strings are at least 36 and no more than 38 characters in length and match the pattern ^\{?[a-zA-Z0-9]{36}?\}$ and hence are always ascii. – AnthonyWJones Sep 19, 2008 at 20:35

5 David Bau provides a much better, seedable random number generator at davidbau.com/archives/2010/01/30/... I wrote up a slightly different approach to generating UUIDs at blogs.cozi.com/tech/2010/04/generating-uuids-in-javascript.html – George V. Reilly May 4, 2010 at 23:09

1 Weird that no one has mentioned this yet but for completeness, there's a plethora of guid generators on npm I'm willing to bet most of them work in browser too. – George Mauer Feb 3, 2014 at 15:54

1 If anyone wants more options like different versions of the uuid and non standard guid support, REST based uuid generation services like these [fungenerators.com/api/uuid] are an attractive option too. – dors Dec 15, 2020 at 16:20

2 Some 12 years later with `BigInt` and ES6 classes, other techniques that yield rates of 500,000 uuid/sec can be done. See reference – smallscript Dec 30, 2020 at 6:34

6 As others have mentioned, if you're only generating a small number of uuids in a browser, just use `URL.createObjectURL(new Blob()).substr(-36)` . (Excellent browser support, too). (To avoid memory leakage, call URL.revokeObjectURL(url)) – rinogo Oct 21, 2021 at 2:47

1 if you have mission-critical issue, better to write an endpoint which is written with pyhton and call it. Because its implemented such as described at datatracker.ietf.org/doc/html/rfc4122.html – Burcin Sep 15, 2022 at 18:46

Anno 2024, there is `crypto.randomUUID()` . Thanks to Pier-Luc Gendreau for mentioning it first. – Code4R7 Nov 5, 2024 at 8:12

This page has an uuid generator for all versions 1 through 7 in browser side javascript uuidgenerators.net Its code is not obfuscated, so if you open page source, you can see how it was implemented – Goran Jovic Nov 13, 2024 at 10:30

Sixteen years later, I'm looking into generating guids with javascript and here's some really useful dead links. Is it too much troube to cut and paste into an answer, then add a credit? – sanepete Feb 17 at 3:36

76 Answers

Sorted by: Highest score (default)

1 2 3 Next

5716

[Edited 2023-03-05 to reflect latest best-practices for producing RFC4122-compliant UUIDs]

`crypto.randomUUID()` is now standard on all modern browsers and JS runtimes. However, because new browser APIs are restricted to secure contexts, this method is only available to pages served locally (`localhost` or `127.0.0.1`) or over HTTPS.

For readers interested in other UUID versions, generating UUIDs on legacy platforms or in non-secure contexts, there is the `uuid` module. It is well-tested and supported.

Failing the above, there is this method (based on the original answer to this question):

```
function uuidv4() {
  return "10000000-1000-4000-8000-100000000000".replace(/[018]/g, c =>
    (+c ^ crypto.getRandomValues(new Uint8Array(1))[0] & 15 >> +c /
4).toString(16)
  );
}

console.log(uuidv4());
```

Run code snippet Expand snippet

Note: **The use of *any* UUID generator that relies on `Math.random()` is strongly discouraged** (including snippets featured in previous versions of this answer) for reasons best explained here. *TL;DR:* solutions based on `Math.random()` do not provide good uniqueness guarantees.

Share Improve this answer Follow

edited Mar 23, 2024 at 17:11

community wiki
34 revs, 21 users 51%
broofa

182 Surely the answer to @Muxa's question is 'no'? It's never truly safe to trust something that came from the client. I guess it depends on how likely your users are to bring up a javascript console and manually change the variable so to something they want. Or they could just POST you back the id that they want. It would also depend on whether the user picking their own ID is going to cause vulnerabilities. Either way, if it's a random number ID that's going into a table, I would probably be generating it server-side, so that I know I have control over the process. – Cam Jackson Nov 1, 2012 at 14:34

49 @DrewNoakes - UUIDs aren't just a string of completely random #'s. The "4" is the uuid version (4 = "random"). The "y" marks where the uuid variant (field layout, basically) needs to be embedded. See sections 4.1.1 and 4.1.3 of ietf.org/rfc/rfc4122.txt for more info. – broofa Nov 27, 2012 at 22:13

22 im a bit confused, in javascript `[1e7]+-1e3` does not really mean anything, an array is added to a number? what am I missing? note: in typescript it does not pass – Ayyash Oct 20, 2021 at 12:14

1 @Ayyash In my Chrome F12 console `[1e7]+-1e3` evaluates to a string: '10000000-1000' – manuell Oct 25, 2021 at 17:04

5 @Ayyash see destroyallsoftware.com/talks/wat – manuell Oct 25, 2021 at 17:11

1 @ayyash The original derivation of that code snippet can be found here: gist.github.com/jed/982883 . It is, admittedly, ludicrously cryptic. For something less cryptic: gist.github.com/jed/982883#gistcomment-2403369 – broofa Oct 26, 2021 at 18:37

4 Very new browsers required for crypto.randomUUID(). caniuse.com/mdn-api_crypto_randomuuid. – wazz Nov 12, 2021 at 18:02

@Ayyash this seems to produce the same result with valid JS/TS: `1e7 + '-' + 1e3` – Roope Hakulinen Feb 21, 2022 at 15:46

3 This doesn't work in Typescript it says can't add `number[] and number` – dota2pro Apr 6, 2022 at 12:09

nodejs/React gives the warning: `Use parentheses to clarify the intended order of operations: no-mixed-operators` ; great solution though. – PLG May 10, 2022 at 12:06

12 Typescript users: you can add `<any>` right before the first array, like this: `<any>[1e7]` - quick way to get it to pass. – NickyTheWrench May 25, 2022 at 3:39

6 I like the custom solution, except for the expression `[1e7]+-1e3+-4e3+-8e3+-1e11` in it. I personally prefer the expression `` `${1e7}-${1e3}-${4e3}-${8e3}-${1e11}` `` instead. It's somewhat longer, but also somewhat clearer to me. And it allows me to easily change the generated UUID's format as well. – Bart Hofland Jul 12, 2022 at 11:19

A big warning from me about `crypto.randomUUID()`: **it is only available in 'secure contexts'**. As far as I can tell, that means you can't use it if your site is HTTP, not HTTPS, or if you're (more likely) testing locally with HTTP. An absurd decision IMHO considering UUIDs are used for all sorts of purposes, many of them not remotely security-sensitive. I'll be forced to use the NPM module. – Jez Nov 15, 2022 at 22:49

For those who want to use `crypto.randomUUID()` by just running locally in a simple script file with Node.js, you'll need to import with `crypto = require("crypto");` – Samuel T. Aug 3, 2023 at 18:35

2 The left-hand side of an arithmetic operation must be of type 'any', 'number', 'bigint' or an enum type.ts(2362) The error it gives for this JS code. – Cemstrian Jan 4, 2024 at 14:30

1 @Cemstrian: Replace `c` with `+c` for the two occurrences within the body of the arrow function. That should make the type conversion explicit(ish) and keep TypeScript happy. – Mark Dickinson Mar 22, 2024 at 8:11

2642

UUIDs (Universally Unique IDentifier), also known as GUIDs (Globally Unique IDentifier), according to RFC 4122, are identifiers designed to provide certain uniqueness guarantees.

While it is possible to implement RFC-compliant UUIDs in a few lines of JavaScript code (e.g., see @broofa's answer, below) there are several common pitfalls:

- Invalid id format (UUIDs must be of the form " `xxxxxxxx-xxxx-Mxxx-Nxxx-xxxxxxxxxxxx` ", where x is one of [0-9, a-f] *M* is one of [1-5], and *N* is [8, 9, a, or b]
- Use of a low-quality source of randomness (such as `Math.random`)

Thus, developers writing code for production environments are encouraged to use a rigorous, well-maintained implementation such as the uuid module.

Share Improve this answer Follow

edited Dec 21, 2020 at 5:33

community wiki
25 revs, 18 users 18%
broofa

214 Actually, the RFC allows for UUIDs that are created from random numbers. You just have to twiddle a couple of bits to identify it as such. See section 4.4. Algorithms for Creating a UUID from Truly Random or Pseudo-Random Numbers: rfc-archive.org/getrfc.php?rfc=4122

– Jason DeFontes Sep 19, 2008 at 20:28

72 This should not be the accepted answer. It does not actually answer the question - instead encouraging the import of 25,000 lines of code for something you can do with one line of code in any modern browser. – Abhi Beckert Jul 8, 2020 at 0:36

1 @AbhiBeckert the answer is from 2008 and for node.js projects it might be valid to choose a dependency more over project size – Phil Sep 29, 2020 at 13:48

10 @Phil this is a "highly active question", which means it should have an excellent answer with a green tick. Unfortunately that's not the case. There is nothing wrong or incorrect with this answer (if there was, I'd edit the answer) - but another far better answer exists below and I think it should be at the top of the list. Also the question is specifically relating to javascript in a browser, not node.js. – Abhi Beckert Oct 8, 2020 at 3:47

4 I challenge the claim that Math.random is that low of a quality of randomness. v8.dev/blog/math-random. As you can see, it's passes a good test suite, and the same algorithm is used by v8, FF and Safari. And the RFC states, pseudo-random numbers are acceptable for UUIDs – Munawwar May 27, 2021 at 8:38

I think this answer should be edited by a qualified mod. Not only is it factually wrong, but it also doesn't answer the question at hand. Javascript development is different in the context of browser vs server. It's not trivial when a question targets the browser. Might well be saying use a node.js package when a question is specifically about Edge or Opera. Also, it may be better to request a UUID over REST, although that's only opinion. This way various means can be used to ensure it is truly unique. The server can even use the suggested package. But most servers don't use node.js either. – JSON May 28, 2022 at 4:11

Or even better use asm.js. if we're going to offer irrelevant answers we might as well go hard on it – JSON May 28, 2022 at 4:20

I tend to agree with Munawwar. What's in a UUID? Most of the time, I just want to get a value that is pretty unique. Using `Math.random` seems to suffice for me in practice. It would be interesting to learn about some practical reasons why `Math.random` should be avoided for daily non-critical use, and in which circumstances the use `Math.random` would yield problematic practical issues. I guess that's off-topic here. I will try to spend some time to dive deeper in this issue, but for now I am not (yet) convinced regarding the "`Math.random` ban". – Bart Hofland Jun 28, 2022 at 8:44

@BartHofland @munawwar: The issue with Math.random() isn't that implementations are low-quality. It's that they are not guaranteed to be high-quality. Even modern implementations that rely on "good" PRNG algos such as `xorshift128+` (as most modern browsers now do) suffer from being deterministic. Anyone with knowledge of the internal state can predict what values will be generated. This can be exploited for malicious purposes. Hence, the recommendation for "cryptographic quality" RNGs that introduce real entropy into the RNG state generation algorithm. – broofa Jul 11, 2022 at 19:04

@broofa: I understand the necessity of having a secure nondeterministic random number generator. However, I still fail to see why the use of a partially deterministic RNG can be a problem when generating UUIDs (especially if those UUIDs are only needed for internal purposes). I also fail to find information about the dangers of low-quality UUIDs and how such UUIDs can actually be exploited for malicious purposes. Can you please elaborate a little on this? Or provide an example? Or refer to an information source on the Internet that provides such explanations and/or examples? – Bart Hofland Jul 12, 2022 at 10:28

@broofa: It seems that it all comes down to the consideration if the next generated UUID should be hard to predict. For most of my own purposes which just involves generating unique values for identifying objects, entities, components etc., that predictability does not seem to be that important. I do not use any of my generated UUIDs in the context of cryptographic functionality or any other contexts where true randomness is important. So mostly I do not consider the benefits of high-quality UUIDs to outweigh the overhead of (and the additional dependency on) yet another external library. – Bart Hofland Jul 12, 2022 at 11:09

Now we have in node.js and in the browser `crypto.randomUUID()` no need for uuid packages anymore for most use cases. – Ciro Spaciari Oct 12, 2022 at 12:38

@BartHofland It is extremely important for security. Predictable UUIDs have been exploited to gain unauthorized access to resources. e.g. predict a password reset token, or gain access to resources identified by secret UUID links. – Eric Elliott Feb 23, 2023 at 1:38

@JasonDeFontes Thx! TIL: *4.4. Algorithms for Creating a UUID from Truly Random or Pseudo-Random Numbers - The version 4 UUID is meant for generating UUIDs f/ truly-random or pseudo-random numbers. The algorithm is as follows: o Set the two most significant bits (bits 6 & 7) of the clock_seq_hi_and_reserved to 0 and 1, respectively. o Set the four most significant bits (bits 12 through 15) of the time_hi_and_version field to the 4-bit version number from Section 4.1.3. o Set all the other bits to randomly (or pseudo-randomly) chosen values.* **The convention admits the algo's weakness.** – ruffin Sep 25, 2023 at 19:32

1027

I really like how clean Broofa's answer is, but it's unfortunate that poor implementations of `Math.random` leave the chance for collision.

Here's a similar RFC4122 version 4 compliant solution that solves that issue by offsetting the first 13 hex numbers by a hex portion of the timestamp, and once depleted offsets by a hex portion of the microseconds since pageload. That way, even if `Math.random` is on the same seed, both clients would have to generate the UUID the exact same number of microseconds since pageload (if high-perfomance time is supported) AND at the exact same millisecond (or 10,000+ years later) to get the same UUID:

```
function generateUUID() { // Public Domain/MIT
    var d = new Date().getTime();//Timestamp
    var d2 = ((typeof performance !== 'undefined') && performance.now &&
(performance.now()*1000)) || 0;//Time in microseconds since page-load or 0 if
unsupported
    return 'xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx'.replace(/[xy]/g, function(c) {
        var r = Math.random() * 16;//random number between 0 and 16
        if(d > 0){//Use timestamp until depleted
            r = (d + r)%16 | 0;
            d = Math.floor(d/16);
        } else {//Use microseconds since page-load if supported
            r = (d2 + r)%16 | 0;
```

```
            d2 = Math.floor(d2/16);
        }
        return (c === 'x' ? r : (r & 0x3 | 0x8)).toString(16);
    });
}

var onClick = function(){
    document.getElementById('uuid').textContent = generateUUID();
}
onClick();



#uuid { font-family: monospace; font-size: 1.5em; }



<p id="uuid"></p>
<button id="generateUUID" onclick="onClick();">Generate UUID</button>
```

Run code snippet Expand snippet

Here's a fiddle to test.

Modernized snippet for ES6

Show code snippet

Share Improve this answer Follow

edited Jan 21, 2023 at 19:06

community wiki
28 revs, 14 users 66%
Briguy37

40 Bear in mind, `new Date().getTime()` is not updated every millisecond. I'm not sure how this affects the expected randomness of your algorithm. – devios1 Mar 18, 2012 at 17:27

95 performance.now would be even better. Unlike Date.now, the timestamps returned by `performance.now()` are not limited to one-millisecond resolution. Instead, they represent times as floating-point numbers with up to **microsecond precision**. Also unlike Date.now, the values returned by performance.now() **always increase at a constant rate**, independent of the system clock which might be adjusted manually or skewed by software such as the Network Time Protocol. – SavoryBytes Mar 13, 2014 at 4:25

The actual time resolution may or may not be 17 ms (1/60 second), not 1 ms. – Peter Mortensen Dec 30, 2020 at 3:28

Would Crypto.getRandomValues fix the main problems with Math.random?? – John Apr 1, 2021 at 22:07

Note: Zak had made an update to add the following. `const { performance } = require('perf_hooks');` to define the `performance` variable. Thus for node.js strict implementations this may be needed. However, this has been removed from the answer because it broke the snippet and will not work with JavaScript used in most other environments, e.g. in browsers. – Briguy37 Apr 8, 2021 at 20:23

i am new to javascript. when I run the ES6 version with node 14, i get `ReferenceError: performance is not defined` . How do i fix it? – Naveen Reddy Marthala Sep 11, 2021 at 8:41

3 @NaveenReddyMarthala Node.js by default runs JavaScript in strict mode, which unfortunately doesn't allow boolean logic operators to shorthand check the truthiness of `undefined` variables. To fix this, try replacing `var d2 = (performance ..` with `var d2 = (typeof performance !== 'undefined' ..` as in the update version. The other option (which will actually utilize the enhanced precision of performance with Node.js rather than throwing it away) is to re-add `const { performance } = require('perf_hooks');` in your requirements. – Briguy37 Sep 13, 2021 at 14:47

I've added the `typeof performance !== 'undefined'` in the updated ES6 snippet because the code wasn't doing what was intended. The old code would have produced a crash if the `performance` variable wasn't declare instead of defaulting to `0` as expected. CC: @Briguy37 – José Cabo Jan 21, 2023 at 19:08

Why the first code snippet uses `(c === 'x' ? r : (r & 0x3 | 0x8))` while the "modernized" ES6 uses `(c == 'x' ? r : (r & 0x7 | 0x8))` . I.e. Why `0x3` vs `0x7` . That'd result in different values. Does anybody know why? Also, the only difference between the normal one and the "modernized" one are the fat-arrow functions, the use of let instead of var and the bit I've mentioned before. The ES6 one shouldn't be used IMO unless clarified by somebody. – José Cabo Jan 21, 2023 at 19:15

@JoséCabo: That is from 4.4 of ietf.org/rfc/rfc4122.txt: "Set the two most significant bits (bits 6 and 7) of the clock_seq_hi_and_reserved to zero and one, respectively". 0x3 is 0011 in binary, so binary AND with that sets the two most significant bits to zero, and then binary ORing with 0x8 (1000 in binary) sets the MSB to 1 as specified. – Briguy37 Jan 24, 2023 at 14:39

The modernized ES6 snippet generates non-V4 UUIDs, specifically the regex `^[0-9a-f]{8}-[0-9a-f]{4}-4[0-9a-f]{3}-[89ab][0-9a-f]{3}-[0-9a-f]{12}$` does not match, at the position where `[89ab]` is expected. – Alex Suzuki Apr 19, 2023 at 6:23

broofa's answer is pretty slick, indeed - impressively clever, really... RFC4122 compliant, somewhat readable, and compact. Awesome!

561

But if you're looking at that regular expression, those many `replace()` callbacks, `toString()`'s and `Math.random()` function calls (where he's only using four bits of the result and wasting the rest), you may start to wonder about performance. Indeed, joelpt even decided to toss out an RFC for generic GUID speed with `generateQuickGUID`.

But, can we get speed *and* RFC compliance? I say, YES! Can we maintain readability? Well... Not really, but it's easy if you follow along.

But first, my results, compared to broofa, `guid` (the accepted answer), and the non-rfc-compliant `generateQuickGuid`:

```
                  Desktop   Android
            broofa: 1617ms   12869ms
                e1:  636ms    5778ms
                e2:  606ms    4754ms
                e3:  364ms    3003ms
                e4:  329ms    2015ms
                e5:  147ms    1156ms
                e6:  146ms    1035ms
                e7:  105ms     726ms
              guid:  962ms   10762ms
generateQuickGuid:  292ms    2961ms
  - Note: 500k iterations, results will vary by browser/CPU.
```

So by my 6th iteration of optimizations, I beat the most popular answer by over **12 times**, the accepted answer by over **9 times**, and the fast-non-compliant answer by **2-3 times**. And I'm still RFC 4122 compliant.

Interested in how? I've put the full source on http://jsfiddle.net/jcward/7hyaC/3/ and on https://jsben.ch/xczxS

For an explanation, let's start with broofa's code:

```
function broofa() {
    return 'xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx'.replace(/[xy]/g, function(c) {
        var r = Math.random()*16|0, v = c == 'x' ? r : (r&0x3|0x8);
        return v.toString(16);
    });
}

console.log(broofa())
```

Run code snippet Expand snippet

So it replaces `x` with any random hexadecimal digit, `y` with random data (except forcing the top two bits to `10` per the RFC spec), and the regex doesn't match the `-` or `4` characters, so he doesn't have to deal with them. Very, very slick.

The first thing to know is that function calls are expensive, as are regular expressions (though he only uses 1, it has 32 callbacks, one for each match, and in each of the 32 callbacks it calls Math.random() and v.toString(16)).

The first step toward performance is to eliminate the RegEx and its callback functions and use a simple loop instead. This means we have to deal with the `-` and `4` characters whereas broofa did not. Also, note that we can use String Array indexing to keep his slick String template architecture:

```
function e1() {
    var u='',i=0;
    while(i++<36) {
        var c='xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx'[i-
1],r=Math.random()*16|0,v=c=='x'?r:(r&0x3|0x8);
        u+=(c=='-'||c=='4')?c:v.toString(16)
    }
    return u;
}

console.log(e1())
```

Run code snippet Expand snippet

Basically, the same inner logic, except we check for `-` or `4`, and using a while loop (instead of `replace()` callbacks) gets us an almost 3X improvement!

The next step is a small one on the desktop but makes a decent difference on mobile. Let's make fewer Math.random() calls and utilize all those random bits instead of throwing 87% of them away with a random buffer that gets shifted out each iteration. Let's also move that template definition out of the loop, just in case it helps:

```
function e2() {
    var u='',m='xxxxxxxx-xxxx-4xxx-yxxx-
xxxxxxxxxxxx',i=0,rb=Math.random()*0xffffffff|0;
    while(i++<36) {
        var c=m[i-1],r=rb&0xf,v=c=='x'?r:(r&0x3|0x8);
        u+=(c=='-'||c=='4')?c:v.toString(16);rb=i%8==0?
Math.random()*0xffffffff|0:rb>>4
    }
    return u
}

console.log(e2())
```

Run code snippet Expand snippet

This saves us 10-30% depending on platform. Not bad. But the next big step gets rid of the toString function calls altogether with an optimization classic - the look-up table. A simple 16-element lookup table will perform the job of toString(16) in much less time:

```
function e3() {
    var h='0123456789abcdef';
    var k='xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx';
    /* same as e4() below */
}
function e4() {
    var h=['0','1','2','3','4','5','6','7','8','9','a','b','c','d','e','f'];
    var k=['x','x','x','x','x','x','x','x','-','x','x','x','x','-
','4','x','x','x','-','y','x','x','x','-
','x','x','x','x','x','x','x','x','x','x','x','x'];
    var u='',i=0,rb=Math.random()*0xffffffff|0;
    while(i++<36) {
        var c=k[i-1],r=rb&0xf,v=c=='x'?r:(r&0x3|0x8);
        u+=(c=='-'||c=='4')?c:h[v];rb=i%8==0?Math.random()*0xffffffff|0:rb>>4
    }
    return u
}

console.log(e4())
```

Run code snippet Expand snippet

The next optimization is another classic. Since we're only handling four bits of output in each loop iteration, let's cut the number of loops in half and process eight bits in each iteration. This is tricky since we still have to handle the RFC compliant bit positions, but it's not too hard. We then have to make a larger lookup table (16x16, or 256) to store 0x00 - 0xFF, and we build it only once, outside the e5() function.

```
var lut = []; for (var i=0; i<256; i++) { lut[i] = (i<16?'0':'')+
(i).toString(16); }
function e5() {
    var k=['x','x','x','x','-','x','x','-','4','x','-','y','x','-
','x','x','x','x','x','x'];
    var u='',i=0,rb=Math.random()*0xffffffff|0;
    while(i++<20) {
        var c=k[i-1],r=rb&0xff,v=c=='x'?r:(c=='y'?(r&0x3f|0x80):(r&0xf|0x40));
        u+=(c=='-')?c:lut[v];rb=i%4==0?Math.random()*0xffffffff|0:rb>>8
    }
    return u
}

console.log(e5())
```

Run code snippet Expand snippet

I tried an e6() that processes 16-bits at a time, still using the 256-element LUT, and it showed the diminishing returns of optimization. Though it had fewer iterations, the inner logic was complicated by the increased processing, and it performed the same on desktop, and only ~10% faster on mobile.

The final optimization technique to apply - unroll the loop. Since we're looping a fixed number of times, we can technically write this all out by hand. I tried this once with a single random variable, `r`, that I kept reassigning, and performance tanked. But with four variables assigned random data up front, then using the lookup table, and applying the proper RFC bits, this version smokes them all:

```
var lut = []; for (var i=0; i<256; i++) { lut[i] = (i<16?'0':'')+
(i).toString(16); }
function e7()
{
    var d0 = Math.random()*0xffffffff|0;
    var d1 = Math.random()*0xffffffff|0;
    var d2 = Math.random()*0xffffffff|0;
    var d3 = Math.random()*0xffffffff|0;
    return lut[d0&0xff]+lut[d0>>8&0xff]+lut[d0>>16&0xff]+lut[d0>>24&0xff]+'-'+
    lut[d1&0xff]+lut[d1>>8&0xff]+'-'+lut[d1>>16&0x0f|0x40]+lut[d1>>24&0xff]+'-'+
    lut[d2&0x3f|0x80]+lut[d2>>8&0xff]+'-'+lut[d2>>16&0xff]+lut[d2>>24&0xff]+
    lut[d3&0xff]+lut[d3>>8&0xff]+lut[d3>>16&0xff]+lut[d3>>24&0xff];
}

console.log(e7())
```

Run code snippet Expand snippet

Modualized: http://jcward.com/UUID.js - `UUID.generate()`

The funny thing is, generating 16 bytes of random data is the easy part. The whole trick is expressing it in *string* format with RFC compliance, and it's most tightly accomplished with 16 bytes of random data, an unrolled loop and lookup table.

I hope my logic is correct -- it's very easy to make a mistake in this kind of tedious bit work. But the outputs look good to me. I hope you enjoyed this mad ride through code optimization!

Be advised: my primary goal was to show and teach potential optimization strategies. Other answers cover important topics such as collisions and truly random numbers, which are important for generating good UUIDs.

Share Improve this answer Follow

edited Apr 4, 2022 at 15:06

community wiki
14 revs, 5 users 61%
Jeff Ward

27 This code still contains a couple of errors: the `Math.random()*0xFFFFFFFF` lines should be `Math.random()*0x100000000` for full randomness, and `>>>0` should be used instead of `|0` to keep the values unsigned (though with the current code I think it gets away OK even though they are signed). Finally it would be a very good idea these days to use `window.crypto.getRandomValues` if available, and fall-back to Math.random only

if absolutely necessary. Math.random may well have less than 128 bits of entropy, in which case this would be more vulnerable to collisions than necessary. – Dave Jul 18, 2015 at 17:55

17 Can I just say -- I cannot count how many times I've pointed devs to this answer because it so beautifully points out the tradeoffs between performance, code-elegance, and readability. Thank you Jeff. – Nemesarial Nov 6, 2020 at 14:07

I don't know if @Broofa's answer has changed since these tests were run (or if the browser engines running the tests have changed - it has been five years), but I just ran them both on two different benchmarking services (jsben.ch and jsbench.github.io), and in each case Broofa's answer (using Math.random) was faster than this e7() version by 30 - 35%. – Andrew Nov 17, 2020 at 21:24

@Andy is right. Broofa's code is faster as of Aug 2021. I implemented Dave's suggestions and ran the test myself. But I don't imagine the difference should matter all that much in production: jsbench.github.io/#80610cde9bc93d0f3068e5793e60ff11 – Doomd Aug 12, 2021 at 22:38

I feel your comparisons may be unfair as broofa's anwer appears to be for an e4 UUID, and your testing against Ward's e7 implementation here. When you compare broofa's answer to the e4 version presented to here, this answer is faster. – bedalton Aug 20, 2021 at 14:11

1 @bedalton: Why would we compare broofa's answer to "the e4 version"? The "4" in e4 simply refers to the iteration of optimization and not to the version of UUID, right? – rinogo Oct 21, 2021 at 2:05

I did not catch that – bedalton Oct 26, 2021 at 15:30

@Andrew you have to notice that you generate LUT every time function is called, and in e7 example it happens only once, so after I created a LUT manually (it fetched through) e7 beats broofa – A4tur Sep 28, 2022 at 8:47

230

Use:

```
let uniqueId = Date.now().toString(36) + Math.random().toString(36).substring(2);
```

Show code snippet

If IDs are generated more than 1 millisecond apart, they are 100% unique.

If two IDs are generated at shorter intervals, and assuming that the random method is truly random, this would generate IDs that are 99.99999999999999% likely to be globally unique (collision in 1 of 10^15).

You can increase this number by adding more digits, but to generate 100% unique IDs you will need to use a global counter.

If you need RFC compatibility, this formatting will pass as a valid version 4 GUID:

```
let u = Date.now().toString(16) + Math.random().toString(16) + '0'.repeat(16);
let guid = [u.substr(0,8), u.substr(8,4), '4000-8' + u.substr(13,3),
u.substr(16,12)].join('-');
```

Show code snippet

The above code follow the intention, but not the letter of the RFC. Among other discrepancies it's a few random digits short. (Add more random digits if you need it) The upside is that this is really fast :) You can test validity of your GUID here

Share Improve this answer Follow

edited Aug 12, 2021 at 22:52

community wiki
19 revs, 3 users 91%
Simon Rigét

10 This is not UUID though? – Marco Kerwitz Dec 26, 2017 at 23:28

No. UUID/GUID's is a 122 bit (+ six reserved bits) number. it might guarantee uniqueness through a global counter service, but often it relays on time, MAC address and randomness. UUID's are not random! The UID I suggest here is not fully compressed. You could compress it, to a 122 bit integer, add the 6 predefined bits and extra random bits (remove a few timer bits) and you end up with a perfectly formed UUID/GUID, that you would then have to convert to hex. To me that doesn't really add anything other than compliance to the length of the ID. – Simon Rigét Feb 18, 2018 at 0:05

12 Relaying on MAC addresses for uniqueness on virtual machines is a bad idea! – Simon Rigét Feb 18, 2018 at 0:48

1 I do something like this, but with leading characters and some dashes (e.g `[slug, date, random].join("_")` to create `usr_1dcn27itd_hj6onj6phr` . It makes it so the id also doubles as a "created at" field – Seph Reed Jun 6, 2019 at 19:23

1 Building on @SephReed's comment, I think having the date part first is nice since it sorts chronologically, which may provide benefits later if storing or indexing the IDs. – totalhack Oct 14, 2020 at 20:15

3 For those wondering: `toString(36)` converts in a base-36 numeration (0..9a..z). Example: `(35).toString(36)` is `z` . – Basj Dec 15, 2020 at 8:57

Not compatible with IE since .repeat method is not available. – FatalError Jul 27, 2021 at 8:20

Changing *'0'.repeat(16)* to *'000000'* worked for me. I don't know why 0 needs to be repeated 16 times, only couple of times should be enough. – FatalError Jul 27, 2021 at 8:41

The the many zeros in front are needed, because a random number might be just one digit. Even if its very rare, it still has to work. – Simon Rigét Jul 31, 2021 at 22:13

@SimonRigét This ID isn't truly a UUID/GUID, even if it technically verifies as one, so it's not a good answer to the question (and if you're not going to use a proper UUID, you might as well just commit to using something entirely different like NanoID). Also, If you're referring to UUIDv1, you are correct, it isn't random, but UUIDv4 (which nowadays is by far the more common version and I suspect that was true in 2018 as well) *is* random, as well as the upcoming UUIDv7 which is actually fairly similar in concept to your "RFC-compatible" version. – Abion47 Jan 18, 2024 at 18:11

194

Here's some code based on RFC 4122, section 4.4 (Algorithms for Creating a UUID from Truly Random or Pseudo-Random Number).

```
function createUUID() {
    // http://www.ietf.org/rfc/rfc4122.txt
    var s = [];
    var hexDigits = "0123456789abcdef";
    for (var i = 0; i < 36; i++) {
        s[i] = hexDigits.substr(Math.floor(Math.random() * 0x10), 1);
    }
    s[14] = "4";  // bits 12-15 of the time_hi_and_version field to 0010
    s[19] = hexDigits.substr((s[19] & 0x3) | 0x8, 1);  // bits 6-7 of the
clock_seq_hi_and_reserved to 01
    s[8] = s[13] = s[18] = s[23] = "-";

    var uuid = s.join("");
    return uuid;
}
```

Share Improve this answer Follow

edited Oct 25, 2011 at 22:37

community wiki
Kevin Hakanson

6 You should declare the array size beforehand rather than sizing it dynamically as you build the GUID. `var s = new Array(36);` – MgSam Mar 25, 2013 at 20:03

2 I think there's a very minor bug in the line that sets bits bits 6-7 of the clock_seq_hi_and_reserved to 01. Since s[19] is a character '0'..'f' and not an int 0x0..0xf, (s[19] & 0x3) | 0x8 will not be randomly distributed -- it will tend to produce more '9's and fewer 'b's. This only makes a difference if you care about the random distribution for some reason. – John Velonis Apr 18, 2013 at 15:35

This is the fastest GUID-like string generator method in the format `xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx` . It does not generate a standard-compliant GUID.

106

Ten million executions of this implementation take just 32.5 seconds, which is the fastest I've ever seen in a browser (the only solution without loops/iterations).

The function is as simple as:

```
/**
 * Generates a GUID string.
 * @returns {string} The generated GUID.
 * @example af8a8416-6e18-a307-bd9c-f2c947bbb3aa
 * @author Slavik Meltser.
 * @link http://slavik.meltser.info/?p=142
 */
function guid() {
    function _p8(s) {
        var p = (Math.random().toString(16)+"000000000").substr(2,8);
        return s ? "-" + p.substr(0,4) + "-" + p.substr(4,4) : p ;
    }
    return _p8() + _p8(true) + _p8(true) + _p8();
}
```

To test the performance, you can run this code:

```
console.time('t');
for (var i = 0; i < 10000000; i++) {
    guid();
};
console.timeEnd('t');
```

I'm sure most of you will understand what I did there, but maybe there is at least one person that will need an explanation:

The algorithm:

- The `Math.random()` function returns a decimal number between 0 and 1 with 16 digits after the decimal fraction point (for example `0.4363923368509859`).
- Then we take this number and convert it to a string with base 16 (from the example above we'll get `0.6fb7687f`). `Math.random().toString(16)`.

- Then we cut off the `0.` prefix (`0.6fb7687f => 6fb7687f`) and get a string with eight hexadecimal characters long. `(Math.random().toString(16).substr(2,8)` .
- Sometimes the `Math.random()` function will return shorter number (for example `0.4363`), due to zeros at the end (from the example above, actually the number is `0.4363000000000000`). That's why I'm appending to this string `"000000000"` (a string with nine zeros) and then cutting it off with `substr()` function to make it nine characters exactly (filling zeros to the right).
- The reason for adding exactly nine zeros is because of the worse case scenario, which is when the `Math.random()` function will return exactly 0 or 1 (probability of 1/10^16 for each one of them). That's why we needed to add nine zeros to it (`"0"+"000000000"` or `"1"+"000000000"`), and then cutting it off from the second index (third character) with a length of eight characters. For the rest of the cases, the addition of zeros will not harm the result because it is cutting it off anyway. `Math.random().toString(16)+"000000000").substr(2,8)` .

The assembly:

- The GUID is in the following format `xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx` .
- I divided the GUID into four pieces, each piece divided into two types (or formats): `xxxxxxxx` and `-xxxx-xxxx` .
- Now I'm building the GUID using these two types to assemble the GUID with call four pieces, as follows: `xxxxxxxx -xxxx-xxxx -xxxx-xxxx xxxxxxxx` .
- To differ between these two types, I added a flag parameter to a pair creator function `_p8(s)` , the `s` parameter tells the function whether to add dashes or not.
- Eventually we build the GUID with the following chaining: `_p8() + _p8(true) + _p8(true) + _p8()` , and return it.

Link to this post on my blog

Enjoy! :-)

Share Improve this answer Follow

edited Apr 2, 2021 at 14:04

community wiki
14 revs, 4 users 84%
Slavik Meltser

19 This implementation is incorrect. Certain characters of the GUID require special treatment (e.g. the 13th digit needs to be the number 4). – JLRishe Nov 12, 2013 at 8:12

Slightly rewritten, with fat arrow functions and toStr(depricated) -> toString. Also removed the hyphens! guid = () => { _p8 = () => {return (Math.random() * 10000000000).toString(16).substr(0,8);} return `${_p8()}${_p8()}${_p8()}${_p8()}` ; }; – Peter Korinek TellusTalk Aug 15, 2022 at 12:12

@JLRishe 13th digit should not be always number 4, it represents the UUID version. It can be 3,4,5.. depending on the version – Ashish Singh Jan 8 at 7:20

@AshishSingh Version 4 is the only one that's based entirely on random number generation. The other versions involve hashing and seed values of various types. Since this answer ostensibly produces randomly generated UUIDs, the most logical choice for the 13th digit should be 4. At any rate, the 13th digit shouldn't be randomly generated, which is what this answer is doing, and that was my main point. – JLRishe Jan 8 at 16:01

@JLRishe thanks for that info though, I was not aware that random is based only on version 4 – Ashish Singh Jan 9 at 10:20

86

Here is a totally non-compliant but very performant implementation to generate an ASCII-safe GUID-like unique identifier.

```
function generateQuickGuid() {
    return Math.random().toString(36).substring(2, 15) +
        Math.random().toString(36).substring(2, 15);
}
```

Generates 26 [a-z0-9] characters, yielding a UID that is both shorter and more unique than RFC compliant GUIDs. Dashes can be trivially added if human-readability matters.

Here are usage examples and timings for this function and several of this question's other answers. The timing was performed under Chrome m25, 10 million iterations each.

```
>>> generateQuickGuid()
"nvcjf1hs7tf8yyk4lmlijqkuo9"
"yq6gipxqta4kui8z05tgh9qeel"
"36dh5sec7zdj90sk2rx7pjswi2"
runtime: 32.5s

>>> GUID() // John Millikin
"7a342ca2-e79f-528e-6302-8f901b0b6888"
runtime: 57.8s

>>> regexGuid() // broofa
"396e0c46-09e4-4b19-97db-bd423774a4b3"
```

```
runtime: 91.2s

>>> createUUID() // Kevin Hakanson
"403aa1ab-9f70-44ec-bc08-5d5ac56bd8a5"
runtime: 65.9s

>>> UUIDv4() // Jed Schmidt
"f4d7d31f-fa83-431a-b30c-3e6cc37cc6ee"
runtime: 282.4s

>>> Math.uuid() // broofa
"5BD52F55-E68F-40FC-93C2-90EE069CE545"
runtime: 225.8s

>>> Math.uuidFast() // broofa
"6CB97A68-23A2-473E-B75B-11263781BBE6"
runtime: 92.0s

>>> Math.uuidCompact() // broofa
"3d7b7a06-0a67-4b67-825c-e5c43ff8c1e8"
runtime: 229.0s

>>> bitwiseGUID() // jablko
"baeaa2f-7587-4ff1-af23-eeab3e92"
runtime: 79.6s

>>>> betterWayGUID() // Andrea Turri
"383585b0-9753-498d-99c3-416582e9662c"
runtime: 60.0s

>>>> UUID() // John Fowler
"855f997b-4369-4cdb-b7c9-7142ceaf39e8"
runtime: 62.2s
```

Here is the timing code.

```
var r;
console.time('t');
for (var i = 0; i < 10000000; i++) {
    r = FuncToTest();
};
console.timeEnd('t');
```

Share Improve this answer Follow edited Jan 21, 2013 at 21:52 community wiki joelpt

Not sure if implementations have changed in the last 10 years, but Math.random() doesn't produce as many digits for me as it seems to produce for you -- there are only 10 or 11 characters in Math.random().toString(36).substring(2, 15) for me. – Ben Wheeler Sep 4, 2023 at 3:49

80

From sagi shkedy's technical blog:

```
function generateGuid() {
  var result, i, j;
  result = '';
  for(j=0; j<32; j++) {
    if( j == 8 || j == 12 || j == 16 || j == 20)
      result = result + '-';
    i = Math.floor(Math.random()*16).toString(16).toUpperCase();
    result = result + i;
  }
  return result;
}
```

There are other methods that involve using an ActiveX control, but stay away from these!

I thought it was worth pointing out that no GUID generator can guarantee unique keys (check the Wikipedia article). There is always a chance of collisions. A GUID simply offers a large enough universe of keys to reduce the change of collisions to almost nil.

Share Improve this answer Follow edited Oct 3, 2020 at 15:20 community wiki 5 revs, 4 users 78% Prestaul

10 Note that this isn't a GUID in the technical sense, because it does nothing to guarantee uniqueness. That may or may not matter depending on your application. – Stephen Deken Sep 19, 2008 at 20:07

3 A quick note about performance. This solution creates 36 strings total to get a single result. If performance is critical, consider creating an array and joining as recommended by: tinyurl.com/y37xtx Further research indicates it may not matter, so YMMV: tinyurl.com/3l7945 – Brandon DuRette Sep 22, 2008 at 18:14

2 Regarding uniqueness, it's worth noting that version 1,3, and 5 UUIDs are deterministic in ways version 4 isn't. If the inputs to these uuid generators - node id in v1, namespace and name in v3 and v5 - are unique (as they're supposed to be), then the resulting UUIDs be unique. In theory, anyway. – broofa Jun 29, 2017 at 13:26

These GUIDs are invalid because they don't specify version and variant required by the ITU-T | ISO recommendation. – Daniel Marschall Dec 12, 2021 at 1:42

@DanielMarschall, this doesn't produce UUIDs, but does produce valid GUIDs which were common place in Microsoft code (e.g. .Net) in 2008 when this answer was written. Note, that this is also why the hex characters are forced to upper case. See: learn.microsoft.com/en-us/windows/win32/msi/guid – Prestaul Dec 13, 2021 at 16:51

@Prestaul I understand that GUID is an implementation of UUID. I am not aware that GUIDs don't follow the rules of the ITU-T, which say that random GUIDs ("version 4") need the "version" field set to 4 and the variant bits set to 0b01. Of course they are "valid" in regards syntax. But it is not correct to produce GUIDs which do not follow the specifications (i.e. produce a random GUID but label it as time based would cause invalid metadata about the GUID). – Daniel Marschall Dec 13, 2021 at 17:54

@DanielMarschall You are correct that this doesn't follow the UUID/GUID specs. I'm simply stating that, in 2008 when this was written, GUID and UUID did not mean the same thing and Microsoft had its own definition and implementation of GUIDs which was in quite common use. That definition (did you see the link I shared) does not have a version/variant in it. – Prestaul Dec 14, 2021 at 18:33

FWIW, this answer was edited early on to recommend using one of the better solutions but that recommendation was removed later by the community. – Prestaul Dec 14, 2021 at 18:36

71

Here is a combination of the top voted answer, with a workaround for Chrome's collisions:

```
generateGUID = (typeof(window.crypto) != 'undefined' &&
                typeof(window.crypto.getRandomValues) != 'undefined') ?
    function() {
        // If we have a cryptographically secure PRNG, use that
        // https://stackoverflow.com/questions/6906916/collisions-when-
generating-uuids-in-javascript
        var buf = new Uint16Array(8);
        window.crypto.getRandomValues(buf);
        var S4 = function(num) {
            var ret = num.toString(16);
            while(ret.length < 4){
                ret = "0"+ret;
            }
            return ret;
        };
        return (S4(buf[0])+S4(buf[1])+"-"+S4(buf[2])+"-"+S4(buf[3])+"-
```

```
"+S4(buf[4])+"-"+S4(buf[5])+S4(buf[6])+S4(buf[7]));
    }

    :

    function() {
        // Otherwise, just use Math.random
        // https://stackoverflow.com/questions/105034/how-to-create-a-guid-uuid-
in-javascript/2117523#2117523
        return 'xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx'.replace(/[xy]/g,
function(c) {
            var r = Math.random()*16|0, v = c == 'x' ? r : (r&0x3|0x8);
            return v.toString(16);
        });
    };
```

It is on jsbin if you want to test it.

Share Improve this answer Follow

edited Dec 30, 2020 at 3:35

community wiki
4 revs, 3 users 85%
ripper234

4 note that the first version, the one ` window.crypto.getRandomValues , does not keep the Version 4 UUIDs format defined by RFC 4122. That is instead of xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx` it yields xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx . – humanityANDpeace Sep 3, 2016 at 7:58

66

Here's a solution dated Oct. 9, 2011 from a comment by user *jed* at https://gist.github.com/982883:

```
UUIDv4 = function b(a){return a?(a^Math.random()*16>>a/4).toString(16):
([1e7]+-1e3+-4e3+-8e3+-1e11).replace(/[018]/g,b)}
```

This accomplishes the same goal as the current highest-rated answer, but in 50+ fewer bytes by exploiting coercion, recursion, and exponential notation. For those curious how it works, here's the annotated form of an older version of the function:

```
UUIDv4 =

function b(
```

```
  a // placeholder
){
  return a // if the placeholder was passed, return
    ? ( // a random number from 0 to 15
      a ^ // unless b is 8,
      Math.random() // in which case
      * 16 // a random number from
      >> a/4 // 8 to 11
      ).toString(16) // in hexadecimal
    : ( // or otherwise a concatenated string:
      [1e7] + // 10000000 +
      -1e3 + // -1000 +
      -4e3 + // -4000 +
      -8e3 + // -80000000 +
      -1e11 // -100000000000,
      ).replace( // replacing
        /[018]/g, // zeroes, ones, and eights with
        b // random hex digits
      )
}
```

Share Improve this answer Follow

edited May 23, 2017 at 10:31

community wiki
9 revs, 5 users 63%
Jed Schmidt

57

One line solution using Blobs.

```
window.URL.createObjectURL(new Blob([])).substring(31);
```

The value at the end (31) depends on the length of the URL.

EDIT:

A more compact and universal solution, as suggested by rinogo:

```
URL.createObjectURL(new Blob([])).slice(-36);
```

Share Improve this answer Follow

edited Apr 3, 2023 at 20:18

community wiki
4 revs, 2 users 95%
Tikolu

15 Alternatively `window.URL.createObjectURL(new Blob([])).split('/').pop()` will do the same without having to rely on external factors like URL length. – wafs Mar 8, 2021 at 3:57

2 What is "Blob"/"Blobs"? – Peter Mortensen Apr 2, 2021 at 18:33

@PeterMortensen A blob is an opaque, efficient representation of some amount of "raw" (binary) data, for convenience of scripting on the Web. – Armen Michaeli Aug 23, 2021 at 17:39

1 Umm this most definitely does not work. To work reliably on different domains, it needs to be changed to something like `window.URL.createObjectURL(new Blob([])).substr(-36)` – rinogo Oct 21, 2021 at 2:32

1 What's the drawback of this solution? – Charaf Mar 13, 2022 at 0:46

@Charaf Doesn't seem to have many drawbacks except for no support on Internet Explorer – Tikolu Mar 14, 2022 at 1:07

On Safari 15.3 you will get: "ikdev.com/98455e8b-7f1c-44b3-8424-91e06214cb50", so this is not good solution. – lissajous Apr 8, 2022 at 12:04

1 FYI there's some concern and advice given about memory management highlighted on MDN. developer.mozilla.org/en-US/docs/Web/API/URL/... – Lucas Apr 28, 2022 at 14:14

1 instead of `substr` which is deprecated now, the `slice` might be used – asduj Apr 29, 2022 at 12:09

since `substr` is depricated we can also use match `const [uuid] = URL.createObjectURL(new Blob([])).match(/[a-z0-9-]+$/)` – bert84 Sep 14, 2022 at 10:24

3 Better to replace substr with slice in the compact solution. `URL.createObjectURL(new Blob([])).slice(-36)` – lk404 Dec 21, 2022 at 5:09

56

You can use node-uuid. It provides simple, fast generation of RFC4122 UUIDS.

Features:

- Generate RFC4122 version 1 or version 4 UUIDs
- Runs in Node.js and browsers.
- Cryptographically strong random # generation on supporting platforms.

- Small footprint (Want something smaller? Check this out!)

Install Using NPM:

```
npm install uuid
```

Or using uuid via a browser:

Download Raw File (uuid v1): https://raw.githubusercontent.com/kelektiv/node-uuid/master/v1.js Download Raw File (uuid v4): https://raw.githubusercontent.com/kelektiv/node-uuid/master/v4.js

Want even smaller? Check this out: https://gist.github.com/jed/982883

Usage:

```
// Generate a v1 UUID (time-based)
const uuidV1 = require('uuid/v1');
uuidV1(); // -> '6c84fb90-12c4-11e1-840d-7b25c5ee775a'

// Generate a v4 UUID (random)
const uuidV4 = require('uuid/v4');
uuidV4(); // -> '110ec58a-a0f2-4ac4-8393-c866d813b8d1'

// Generate a v5 UUID (namespace)
const uuidV5 = require('uuid/v5');

// ... using predefined DNS namespace (for domain names)
uuidV5('hello.example.com', v5.DNS)); // -> 'fdda765f-fc57-5604-a269-
52a7df8164ec'

// ... using predefined URL namespace (for, well, URLs)
uuidV5('http://example.com/hello', v5.URL); // -> '3bbcee75-cecc-5b56-8031-
b6641c1ed1f1'

// ... using a custom namespace
```

```
const MY_NAMESPACE = '(previously generated unique uuid string)';
uuidV5('hello', MY_NAMESPACE); // -> '90123e1c-7512-523e-bb28-76fab9f2f73d'
```

ECMAScript 2015 (ES6):

```
import uuid from 'uuid/v4';
const id = uuid();
```

Share Improve this answer Follow

edited Apr 2, 2021 at 16:45

community wiki
7 revs, 3 users 85%
Kyros Koh

2 **Note:** These imports didn't work for me. Import statements have changed, as stated in the repo: `const { v4: uuidv4 } = require('uuid');` and ES6: `import { v4 as uuidv4 } from 'uuid';` – vladsiv Dec 2, 2020 at 0:09

39

This creates a version 4 UUID (created from pseudo random numbers):

```
function uuid()
{
   var chars = '0123456789abcdef'.split('');

   var uuid = [], rnd = Math.random, r;
   uuid[8] = uuid[13] = uuid[18] = uuid[23] = '-';
   uuid[14] = '4'; // version 4

   for (var i = 0; i < 36; i++)
   {
      if (!uuid[i])
      {
         r = 0 | rnd()*16;

         uuid[i] = chars[(i == 19) ? (r & 0x3) | 0x8 : r & 0xf];
      }
   }
```

```
    return uuid.join('');
}
```

Here is a sample of the UUIDs generated:

```
682db637-0f31-4847-9cdf-25ba9613a75c
97d19478-3ab2-4aa1-b8cc-a1c3540f54aa
2eed04c9-2692-456d-a0fd-51012f947136
```

Share Improve this answer Follow

edited Oct 3, 2020 at 15:22

community wiki
3 revs, 3 users 62%
Mathieu Pagé

39

```
var uuid = function() {
    var buf = new Uint32Array(4);
    window.crypto.getRandomValues(buf);
    var idx = -1;
    return 'xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx'.replace(/[xy]/g, function(c) {
        idx++;
        var r = (buf[idx>>3] >> ((idx%8)*4))&15;
        var v = c == 'x' ? r : (r&0x3|0x8);
        return v.toString(16);
    });
};
```

This version is based on Briguy37's answer and some bitwise operators to extract nibble sized windows from the buffer.

It should adhere to the RFC Type 4 (random) schema, since I had *problems* last time parsing non-compliant UUIDs with Java's UUID.

Share Improve this answer Follow

edited Dec 30, 2020 at 3:14

community wiki
7 revs, 4 users 73%
sleeplessnerd

Simple JavaScript module as a combination of best answers in this question.

36

```
var crypto = window.crypto || window.msCrypto || null; // IE11 fix

var Guid = Guid || (function() {

  var EMPTY = '00000000-0000-0000-0000-000000000000';

  var _padLeft = function(paddingString, width, replacementChar) {
    return paddingString.length >= width ? paddingString :
_padLeft(replacementChar + paddingString, width, replacementChar || ' ');
  };

  var _s4 = function(number) {
    var hexadecimalResult = number.toString(16);
    return _padLeft(hexadecimalResult, 4, '0');
  };

  var _cryptoGuid = function() {
    var buffer = new window.Uint16Array(8);
    crypto.getRandomValues(buffer);
    return [_s4(buffer[0]) + _s4(buffer[1]), _s4(buffer[2]), _s4(buffer[3]),
_s4(buffer[4]), _s4(buffer[5]) + _s4(buffer[6]) + _s4(buffer[7])].join('-');
  };

  var _guid = function() {
    var currentDateMilliseconds = new Date().getTime();
    return 'xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx'.replace(/[xy]/g,
function(currentChar) {
      var randomChar = (currentDateMilliseconds + Math.random() * 16) % 16 | 0;
      currentDateMilliseconds = Math.floor(currentDateMilliseconds / 16);
      return (currentChar === 'x' ? randomChar : (randomChar & 0x7 |
0x8)).toString(16);
    });
  };

  var create = function() {
    var hasCrypto = crypto != 'undefined' && crypto !== null,
      hasRandomValues = typeof(window.crypto.getRandomValues) != 'undefined';
    return (hasCrypto && hasRandomValues) ? _cryptoGuid() : _guid();
  };

  return {
    newGuid: create,
    empty: EMPTY
  };
})();
```

```
// DEMO: Create and show GUID
console.log('1. New Guid:   ' + Guid.newGuid());

// DEMO: Show empty GUID
console.log('2. Empty Guid: ' + Guid.empty);
```

Run code snippet Expand snippet

Usage:

Guid.newGuid()

"c6c2d12f-d76b-5739-e551-07e6de5b0807"

Guid.empty

"00000000-0000-0000-0000-000000000000"

Share Improve this answer Follow

edited Jan 5, 2022 at 11:48

community wiki
6 revs, 4 users 67%
kayz1

1 What is bothering about *all* answers is that it seems *ok* for JavaScript to store the `GUID` as a `string`. Your answer at least tackles the *much* more efficient storage using a `Uint16Array`. The `toString` function should be using the binary representation in an JavaScript `object` – Sebastian May 4, 2014 at 11:03

This UUIDs produced by this code are either weak-but-RFC-compliant (_guid), or strong-but-not-RFC-compliant (_cryptoGuid). The former uses Math.random(), which is now known to be a poor RNG. The latter is failing to set the version and variant fields. – broofa Jun 29, 2017 at 13:37

@broofa - What would you suggest to make it strong **and** RFC-compliant? And why is _cryptoGuid not RFC-compliant? – Matt Mar 15, 2018 at 9:57

@Matt _cryptoGuid() sets all 128 bits randomly, meaning it doesn't set the version and variant fields as described in the RFC. See my alternate implementation of uuidv4() that uses crypto.getRandomValues() in my top-voted answer, above, for a strong+compliant implementation. – broofa Mar 16, 2018 at 17:13

36

Added in: v15.6.0, v14.17.0 there is a built-in crypto.randomUUID() function.

```
import { randomUUID } from "node:crypto";

const uuid = crypto.randomUUID();
```

In the browser, `crypto.randomUUID()` is currently supported in Chromium 92+ and Firefox 95+.

Share Improve this answer Follow

edited Oct 15, 2023 at 2:03

community wiki
4 revs, 3 users 89%
Pier-Luc Gendreau

And Safari is coming as well! – Brian Cannard Feb 16, 2022 at 19:13

29

The version below is an adaptation of broofa's answer, but updated to include a "true" random function that uses crypto libraries where available, and the Alea() function as a fallback.

```
Math.log2 = Math.log2 || function(n){ return Math.log(n) / Math.log(2); }
Math.trueRandom = (function() {
var crypt = window.crypto || window.msCrypto;

if (crypt && crypt.getRandomValues) {
    // If we have a crypto library, use it
    var random = function(min, max) {
        var rval = 0;
```

```
            var range = max - min;
            if (range < 2) {
                return min;
            }

            var bits_needed = Math.ceil(Math.log2(range));
            if (bits_needed > 53) {
              throw new Exception("We cannot generate numbers larger than 53
bits.");
            }
            var bytes_needed = Math.ceil(bits_needed / 8);
            var mask = Math.pow(2, bits_needed) - 1;
            // 7776 -> (2^13 = 8192) -1 == 8191 or 0x00001111 11111111

            // Create byte array and fill with N random numbers
            var byteArray = new Uint8Array(bytes_needed);
            crypt.getRandomValues(byteArray);

            var p = (bytes_needed - 1) * 8;
            for(var i = 0; i < bytes_needed; i++ ) {
                rval += byteArray[i] * Math.pow(2, p);
                p -= 8;
            }

            // Use & to apply the mask and reduce the number of recursive lookups
            rval = rval & mask;

            if (rval >= range) {
                // Integer out of acceptable range
                return random(min, max);
            }
            // Return an integer that falls within the range
            return min + rval;
        }
        return function() {
            var r = random(0, 1000000000) / 1000000000;
            return r;
        };
      } else {
        // From
https://web.archive.org/web/20120502223108/http://baagoe.com/en/RandomMusings/javasc
        // Johannes Baagøe <baagoe@baagoe.com>, 2010
        function Mash() {
            var n = 0xefc8249d;

            var mash = function(data) {
                data = data.toString();
```

```
            for (var i = 0; i < data.length; i++) {
                n += data.charCodeAt(i);
                var h = 0.02519603282416938 * n;
                n = h >>> 0;
                h -= n;
                h *= n;
                n = h >>> 0;
                h -= n;
                n += h * 0x100000000; // 2^32
            }
            return (n >>> 0) * 2.3283064365386963e-10; // 2^-32
        };

        mash.version = 'Mash 0.9';
        return mash;
    }

    // From http://baagoe.com/en/RandomMusings/javascript/
    function Alea() {
        return (function(args) {
            // Johannes BaagÃ¸e <baagoe@baagoe.com>, 2010
            var s0 = 0;
            var s1 = 0;
            var s2 = 0;
            var c = 1;

            if (args.length == 0) {
                args = [+new Date()];
            }
            var mash = Mash();
            s0 = mash(' ');
            s1 = mash(' ');
            s2 = mash(' ');

            for (var i = 0; i < args.length; i++) {
                s0 -= mash(args[i]);
                if (s0 < 0) {
                    s0 += 1;
                }
                s1 -= mash(args[i]);
                if (s1 < 0) {
                    s1 += 1;
                }
                s2 -= mash(args[i]);
                if (s2 < 0) {
                    s2 += 1;
                }
```

```
                }
                mash = null;

                var random = function() {
                    var t = 2091639 * s0 + c * 2.3283064365386963e-10; // 2^-32
                    s0 = s1;
                    s1 = s2;
                    return s2 = t - (c = t | 0);
                };
                random.uint32 = function() {
                    return random() * 0x100000000; // 2^32
                };
                random.fract53 = function() {
                    return random() +
                        (random() * 0x200000 | 0) * 1.1102230246251565e-16; //
2^-53
                };
                random.version = 'Alea 0.9';
                random.args = args;
                return random;

            }(Array.prototype.slice.call(arguments));
        };
        return Alea();
    }
}());

Math.guid = function() {
    return 'xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx'.replace(/[xy]/g, function(c)
 {
        var r = Math.trueRandom() * 16 | 0,
            v = c == 'x' ? r : (r & 0x3 | 0x8);
        return v.toString(16);
    });
};
```

Share Improve this answer Follow

edited Apr 2, 2021 at 16:14

community wiki
7 revs, 4 users 79%
jvenema

JavaScript project on GitHub - https://github.com/LiosK/UUID.js

27

UUID.js The RFC-compliant UUID generator for JavaScript.

See RFC 4122 http://www.ietf.org/rfc/rfc4122.txt.

Features Generates RFC 4122 compliant UUIDs.

Version 4 UUIDs (UUIDs from random numbers) and version 1 UUIDs (time-based UUIDs) are available.

UUID object allows a variety of access to the UUID including access to the UUID fields.

Low timestamp resolution of JavaScript is compensated by random numbers.

Share Improve this answer Follow

answered Jul 2, 2012 at 21:00

community wiki
Wojciech Bednarski

24

```
// RFC 4122
//
// A UUID is 128 bits long
//
// String representation is five fields of 4, 2, 2, 2, and 6 bytes.
// Fields represented as lowercase, zero-filled, hexadecimal strings, and
// are separated by dash characters
//
// A version 4 UUID is generated by setting all but six bits to randomly
// chosen values
var uuid = [
  Math.random().toString(16).slice(2, 10),
  Math.random().toString(16).slice(2, 6),

  // Set the four most significant bits (bits 12 through 15) of the
  // time_hi_and_version field to the 4-bit version number from Section
  // 4.1.3
  (Math.random() * .0625 /* 0x.1 */ + .25 /* 0x.4 */).toString(16).slice(2, 6),

  // Set the two most significant bits (bits 6 and 7) of the
  // clock_seq_hi_and_reserved to zero and one, respectively
  (Math.random() * .25 /* 0x.4 */ + .5 /* 0x.8 */).toString(16).slice(2, 6),
```

```
    Math.random().toString(16).slice(2, 14)].join('-');
```

Share Improve this answer Follow

answered Jul 14, 2010 at 23:30

community wiki
jablko

18

For those wanting an RFC 4122 version 4 compliant solution with speed considerations (few calls to Math.random()):

```
var rand = Math.random;

function UUID() {
    var nbr, randStr = "";
    do {
        randStr += (nbr = rand()).toString(16).substr(3, 6);
    } while (randStr.length < 30);
    return (
        randStr.substr(0, 8) + "-" +
        randStr.substr(8, 4) + "-4" +
        randStr.substr(12, 3) + "-" +
        ((nbr*4|0)+8).toString(16) + // [89ab]
        randStr.substr(15, 3) + "-" +
        randStr.substr(18, 12)
    );
}

console.log( UUID() );
```

Run code snippet Expand snippet

The above function should have a decent balance between speed and randomness.

Share Improve this answer Follow

edited Dec 30, 2020 at 3:30

community wiki
3 revs, 3 users 63%
John Fowler

17

I couldn't find any answer that uses a single 16-octet `TypedArray` and a `DataView`, so I think the following solution for generating a version 4 UUID per the RFC will stand on its own here:

```
const uuid4 = () => {
    const ho = (n, p) => n.toString(16).padStart(p, 0); /// Return the
hexadecimal text representation of number `n`, padded with zeroes to be of
length `p`; e.g. `ho(13, 2)` returns `"0d"`
    const data = crypto.getRandomValues(new Uint8Array(16)); /// Fill a buffer
with random bits
    data[6] = (data[6] & 0xf) | 0x40; /// Patch the 6th byte to reflect a
version 4 UUID
    data[8] = (data[8] & 0x3f) | 0x80; /// Patch the 8th byte to reflect a
variant 1 UUID (version 4 UUIDs are)
    const view = new DataView(data.buffer); /// Create a view backed by the 16-
byte buffer
    return `${ho(view.getUint32(0), 8)}-${ho(view.getUint16(4),
4)}-${ho(view.getUint16(6), 4)}-${ho(view.getUint16(8),
4)}-${ho(view.getUint32(10), 8)}${ho(view.getUint16(14), 4)}`; /// Compile the
canonical textual form from the array data
};
```

I prefer it because:

- it only relies on functions available to the standard ECMAScript platform, where possible -- which is all but one procedure
- it only uses a single buffer, minimizing copying of data, which should in theory yield performance advantage

At the time of writing this, `getRandomValues` is not something implemented for the `crypto` object in Node.js. However, it has the equivalent `randomBytes` function which may be used instead.

Share Improve this answer Follow

edited Feb 10, 2024 at 13:08

community wiki
13 revs
Armen Michaeli

1 This is readable code, thank you. IMO the accepted answer has too complicated code, but this instead is understandable. – Ciantic Mar 17, 2023 at 13:19

16

I wanted to understand broofa's answer, so I expanded it and added comments:

```
var uuid = function () {
    return 'xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx'.replace(
        /[xy]/g,
        function (match) {
            /*
            * Create a random nibble. The two clever bits of this code:
            *
            * - Bitwise operations will truncate floating point numbers
            * - For a bitwise OR of any x, x | 0 = x
            *
            * So:
            *
            * Math.random * 16
            *
            * creates a random floating point number
            * between 0 (inclusive) and 16 (exclusive) and
            *
            * | 0
            *
            * truncates the floating point number into an integer.
            */
            var randomNibble = Math.random() * 16 | 0;

            /*
            * Resolves the variant field. If the variant field (delineated
            * as y in the initial string) is matched, the nibble must
            * match the mask (where x is a do-not-care bit):
            *
            * 10xx
            *
            * This is achieved by performing the following operations in
            * sequence (where x is an intermediate result):
            *
            * - x & 0x3, which is equivalent to x % 3
            * - x | 0x8, which is equivalent to x + 8
            *
            * This results in a nibble between 8 inclusive and 11 exclusive,
            * (or 1000 and 1011 in binary), all of which satisfy the variant
            * field mask above.
            */
            var nibble = (match == 'y') ?
                (randomNibble & 0x3 | 0x8) :
```

```
                randomNibble;

            /*
            * Ensure the nibble integer is encoded as base 16 (hexadecimal).
            */
            return nibble.toString(16);
        }
    );
};
```

Share Improve this answer Follow

edited Apr 2, 2021 at 16:18

community wiki
5 revs, 2 users 92%
Andrew

Thank you for detailed description! Specifically nibble caged between 8 and 11 with equivalents explanation is super helpful. – Egor Litvinchuk Apr 11, 2020 at 11:26

16

The native `URL.createObjectURL` is generating an UUID. You can take advantage of this.

```
function uuid() {
  const url = URL.createObjectURL(new Blob())
  const [id] = url.toString().split('/').reverse()
  URL.revokeObjectURL(url)
  return id
}
```

Share Improve this answer Follow

edited Apr 2, 2021 at 18:09

community wiki
2 revs, 2 users 82%
Aral Roca

works like a charm. Better than trying to generate manually. Very clever! – Paulo Henrique Queiroz Dec 11, 2020 at 18:15

The performance is quite worst, but depending on the case it can be enough – Aral Roca Dec 15, 2020 at 16:37

For the fastest combined generator that is compliant w/node-clock-seq, monotonic in time, etc. This forms a good basis to seed a `uuid4` generator w/60-bits of epoch70 µ-seconds of monotonic time, 4-bit uuid-version, and 48-bit node-id and 13-bit clock-seq with 3-bit uuid-variant. --

Combining using `BigInt` to write `ntohl` and related conversion this works very fast with the `lut` approach here. --
 I can provide code if desired. – smallscript Dec 29, 2020 at 2:08

Is the inclusion of a UUID here guaranteed, or is it just something that the current browser implementations all happen to do? – M. Justin Jul 15, 2021 at 20:10

1 @M.Justin It's guaranteed -- you can follow w3c.github.io/FileAPI/#dfn-createObjectURL to where it is specified, quoting, "Generate a UUID [RFC4122] as a string and append it to result". – Armen Michaeli Dec 28, 2022 at 16:06

15

ES6 sample

```
const guid=()=> {
  const s4=()=> Math.floor((1 + Math.random()) *
0x10000).toString(16).substring(1);
  return `${s4() + s4()}-${s4()}-${s4()}-${s4()}-${s4() + s4() + s4()}`;
}
```

Share Improve this answer Follow

answered Jul 9, 2017 at 13:01

community wiki
Behnam

An explanation would be in order. E.g., what ES6 features does it use that previous answers don't? Please respond by editing your answer, not here in comments (***without*** "Edit:", "Update:", or similar - the answer should appear as if it was written today). – Peter Mortensen Apr 2, 2021 at 17:28

15

I adjusted my own UUID/GUID generator with some extras here.

I'm using the following Kybos random number generator to be a bit more cryptographically sound.

Below is my script with the Mash and Kybos methods from baagoe.com excluded.

```
//UUID/Guid Generator
// use: UUID.create() or UUID.createSequential()
// convenience:  UUID.empty, UUID.tryParse(string)
(function(w){
  // From http://baagoe.com/en/RandomMusings/javascript/
```

```
  // Johannes BaagÃ¸e <baagoe@baagoe.com>, 2010
  //function Mash() {...};

  // From http://baagoe.com/en/RandomMusings/javascript/
  //function Kybos() {...};

  var rnd = Kybos();

  //UUID/GUID Implementation from
http://frugalcoder.us/post/2012/01/13/javascript-guid-uuid-generator.aspx
  var UUID = {
    "empty": "00000000-0000-0000-0000-000000000000"
    ,"parse": function(input) {
      var ret = input.toString().trim().toLowerCase().replace(/^[\s\r\n]+|[\{\}]|
[\s\r\n]+$/g, "");
      if ((/[a-f0-9]{8}\-[a-f0-9]{4}\-[a-f0-9]{4}\-[a-f0-9]{4}\-[a-f0-9]
{12}/).test(ret))
        return ret;
      else
        throw new Error("Unable to parse UUID");
    }
    ,"createSequential": function() {
      var ret = new Date().valueOf().toString(16).replace("-","")
      for (;ret.length < 12; ret = "0" + ret);
      ret = ret.substr(ret.length-12,12); //only least significant part
      for (;ret.length < 32;ret += Math.floor(rnd() * 0xffffffff).toString(16));
      return [ret.substr(0,8), ret.substr(8,4), "4" + ret.substr(12,3), "89AB"
[Math.floor(Math.random()*4)] + ret.substr(16,3),  ret.substr(20,12)].join("-");
    }
    ,"create": function() {
      var ret = "";
      for (;ret.length < 32;ret += Math.floor(rnd() * 0xffffffff).toString(16));
      return [ret.substr(0,8), ret.substr(8,4), "4" + ret.substr(12,3), "89AB"
[Math.floor(Math.random()*4)] + ret.substr(16,3),  ret.substr(20,12)].join("-");
    }
    ,"random": function() {
      return rnd();
    }
    ,"tryParse": function(input) {
      try {
        return UUID.parse(input);
      } catch(ex) {
        return UUID.empty;
      }
    }
  };
  UUID["new"] = UUID.create;
```

```
  w.UUID = w.Guid = UUID;
}(window || this));
```

Share Improve this answer Follow

edited Dec 30, 2020 at 3:37

community wiki
2 revs, 2 users 94%
Tracker1

13

The better way:

```
function(
  a, b              // Placeholders
){
  for(              // Loop :)
      b = a = '';   // b - result , a - numeric variable
      a++ < 36;     //
      b += a*51&52  // If "a" is not 9 or 14 or 19 or 24
                 ?  //  return a random number or 4
           (
               a^15              // If "a" is not 15,
                  ?              // generate a random number from 0 to 15
               8^Math.random() *
               (a^20 ? 16 : 4)   // unless "a" is 20, in which case a random
number from 8 to 11,
                  :
               4                 //  otherwise 4
          ).toString(16)
                  :
         '-'                     //  In other cases, (if "a" is 9,14,19,24)
insert "-"
      );
  return b
 }
```

Minimized:

```
function(a,b){for(b=a='';a++<36;b+=a*51&52?(a^15?8^Math.random()*(a^20?
16:4):4).toString(16):'-');return b}
```

Share Improve this answer Follow

edited Dec 30, 2020 at 3:41

community wiki
2 revs, 2 users 78%
Andrea Turri

4 Why is it better? – Peter Mortensen Dec 30, 2020 at 3:37

13

The following is simple code that uses `crypto.getRandomValues(a)` on supported browsers (Internet Explorer 11+, iOS 7+, Firefox 21+, Chrome, and Android Chrome).

It avoids using `Math.random()`, because that can cause collisions (for example 20 collisions for 4000 generated UUIDs in a real situation by Muxa).

```
function uuid() {
    function randomDigit() {
        if (crypto && crypto.getRandomValues) {
            var rands = new Uint8Array(1);
            crypto.getRandomValues(rands);
            return (rands[0] % 16).toString(16);
        } else {
            return ((Math.random() * 16) | 0).toString(16);
        }
    }

    var crypto = window.crypto || window.msCrypto;
    return 'xxxxxxxx-xxxx-4xxx-8xxx-xxxxxxxxxxxx'.replace(/x/g, randomDigit);
}
```

Notes:

- Optimised for code readability, not speed, so it is suitable for, say, a few hundred UUIDs per second. It generates about 10000 uuid() per second in Chromium on my laptop using http://jsbin.com/fuwigo/1 to measure performance.
- It only uses *8* for "y" because that simplifies code readability (y is allowed to be *8*, *9*, *A*, or *B*).

Share Improve this answer Follow

edited Apr 2, 2021 at 16:27

community wiki
3 revs, 3 users 69%
robocat

If you just need a random 128 bit string in no particular format, you can use:

```
function uuid() {
    return crypto.getRandomValues(new Uint32Array(4)).join('-');
}
```

Which will return something like `2350143528-4164020887-938913176-2513998651` .

Share Improve this answer Follow

edited Apr 2, 2021 at 17:23

community wiki
2 revs, 2 users 71%
Jonathan Potter

BTW, why does it generate only numbers and not characters as well? much less secure – vsync Sep 30, 2018 at 6:27

2 you can also add characters (letters) like this: `Array.from((window.crypto || window.msCrypto).getRandomValues(new Uint32Array(4))).map(n => n.toString(16)).join('-')` – magikMaker Mar 29, 2019 at 19:55

Just another more readable variant with just two mutations.

```
function uuid4()
{
  function hex (s, b)
  {
    return s +
      (b >>> 4   ).toString (16) +  // high nibble
      (b & 0b1111).toString (16);   // low nibble
  }

  let r = crypto.getRandomValues (new Uint8Array (16));

  r[6] = r[6] >>> 4 | 0b01000000; // Set type 4: 0100
  r[8] = r[8] >>> 3 | 0b10000000; // Set variant: 100

  return r.slice ( 0,  4).reduce (hex, '' ) +
         r.slice ( 4,  6).reduce (hex, '-') +
         r.slice ( 6,  8).reduce (hex, '-') +
```

```
            r.slice ( 8, 10).reduce (hex, '-') +
            r.slice (10, 16).reduce (hex, '-');
    }
```

Share Improve this answer Follow

edited Sep 23, 2019 at 23:10

community wiki
3 revs, 3 users 88%
ceving

1 Well most of the js devs are web developers, and we won't understand what bitwise operators do, because we don't use them most of the times we develop. Actually I never needed any of them, and I am a js dev since '97. So your example code is still totally unreadable to the average web developer who will read it. Not to mention that you still use single letter variable names, which makes it even more cryptic. Probably read Clean Code, maybe that helps: amazon.com/Clean-Code-Handbook-Software-Craftsmanship/dp/... – inf3rno Sep 22, 2018 at 22:16

1 @inf3rno don't bash him, all the proposed solutions in this thread are cryptic but they are correct answers considering the question was to have a one-liner of sorts. that's what one-liners are cryptic. they can't afford to be readable to the average developer but they save screen real estate where a simple preceding comment will do. And as a result, ends up being much more readable that way then if it had been in "readable code" instead. – tatsu Dec 6, 2019 at 15:19

@user1529413 Yes. Uniqueness requires an index. – ceving Feb 14, 2020 at 9:37

This is my favourite answer, because it's building a UUID as a 16-byte (128 bit) value, and not its serialized, nice to read form. It'd be trivially easy to drop the string stuff and just set the correct bits of a random 128bit, which is all a uuidv4 needs to be. You could base64 it for shorter URLs, pass it back to some webassembly, store it in less memory space than as a string, make it a 4096-size buffer and put 256 uuids in it, store in a browser db, etc. Much better than having everything as a long, lowercase hex-encoded string from the start. – Josh from Qaribou Jun 3, 2020 at 1:13

1 2 3 Next

Highly active question. Earn 10 reputation (not counting the association bonus) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.

Start asking to get answers

Find the answer to your question by asking.

Explore related questions

javascript **guid** **uuid**

See similar questions with these tags.

Ask question