# [qs [![Version Badge][npm-version-svg]][package-url]](#3rdcrjn)

[github actions](https://github.com/ljharb/qs/actions) [coverage](https://app.codecov.io/gh/ljharb/qs/) [dependency status](https://david-dm.org/ljharb/qs) [dev dependency status](https://david-dm.org/ljharb/qs#info=devDependencies) [License](http://license) [Downloads](https://npm-stat.com/charts.html?package=qs)

[](https://npmjs.org/package/qs)

A querystring parsing and stringifying library with some added security.

Lead Maintainer: [Jordan Harband](https://github.com/ljharb)

The **qs** module was originally created and maintained by [TJ Holowaychuk](https://github.com/visionmedia/node-querystring).

## Usage

var qs = require('qs');

var assert = require('assert');

var obj = qs.parse('a=c');

assert.deepEqual(obj, { a: 'c' });

var str = qs.stringify(obj);

assert.equal(str, 'a=c');

### Parsing Objects

qs.parse(string, [options]);

**qs** allows you to create nested objects within your query strings, by surrounding the name of sub-keys with square brackets []. For example, the string 'foo[bar]=baz' converts to:

assert.deepEqual(qs.parse('foo[bar]=baz'), {

foo: {

bar: 'baz'

}

});

When using the plainObjects option the parsed value is returned as a null object, created via Object.create(null) and as such you should be aware that prototype methods will not exist on it and a user may set those names to whatever value they like:

var nullObject = qs.parse('a[hasOwnProperty]=b', { plainObjects: true });

assert.deepEqual(nullObject, { a: { hasOwnProperty: 'b' } });

By default parameters that would overwrite properties on the object prototype are ignored, if you wish to keep the data from those fields either use plainObjects as mentioned above, or set allowPrototypes to true which will allow user input to overwrite those properties. *WARNING* It is generally a bad idea to enable this option as it can cause problems when attempting to use the properties that have been overwritten. Always be careful with this option.

var protoObject = qs.parse('a[hasOwnProperty]=b', { allowPrototypes: true });

assert.deepEqual(protoObject, { a: { hasOwnProperty: 'b' } });

URI encoded strings work too:

assert.deepEqual(qs.parse('a%5Bb%5D=c'), {

a: { b: 'c' }

});

You can also nest your objects, like 'foo[bar][baz]=foobarbaz':

assert.deepEqual(qs.parse('foo[bar][baz]=foobarbaz'), {

foo: {

bar: {

baz: 'foobarbaz'

}

}

});

By default, when nesting objects **qs** will only parse up to 5 children deep. This means if you attempt to parse a string like 'a[b][c][d][e][f][g][h][i]=j' your resulting object will be:

var expected = {

a: {

b: {

c: {

d: {

e: {

f: {

'[g][h][i]': 'j'

}

}

}

}

}

}

};

var string = 'a[b][c][d][e][f][g][h][i]=j';

assert.deepEqual(qs.parse(string), expected);

This depth can be overridden by passing a depth option to qs.parse(string, [options]):

var deep = qs.parse('a[b][c][d][e][f][g][h][i]=j', { depth: 1 });

assert.deepEqual(deep, { a: { b: { '[c][d][e][f][g][h][i]': 'j' } } });

The depth limit helps mitigate abuse when **qs** is used to parse user input, and it is recommended to keep it a reasonably small number.

For similar reasons, by default **qs** will only parse up to 1000 parameters. This can be overridden by passing a parameterLimit option:

var limited = qs.parse('a=b&c=d', { parameterLimit: 1 });

assert.deepEqual(limited, { a: 'b' });

To bypass the leading question mark, use ignoreQueryPrefix:

var prefixed = qs.parse('?a=b&c=d', { ignoreQueryPrefix: true });

assert.deepEqual(prefixed, { a: 'b', c: 'd' });

An optional delimiter can also be passed:

var delimited = qs.parse('a=b;c=d', { delimiter: ';' });

assert.deepEqual(delimited, { a: 'b', c: 'd' });

Delimiters can be a regular expression too:

var regexed = qs.parse('a=b;c=d,e=f', { delimiter: /[;,]/ });

assert.deepEqual(regexed, { a: 'b', c: 'd', e: 'f' });

Option allowDots can be used to enable dot notation:

var withDots = qs.parse('a.b=c', { allowDots: true });

assert.deepEqual(withDots, { a: { b: 'c' } });

If you have to deal with legacy browsers or services, there's also support for decoding percent-encoded octets as iso-8859-1:

var oldCharset = qs.parse('a=%A7', { charset: 'iso-8859-1' });

assert.deepEqual(oldCharset, { a: '§' });

Some services add an initial utf8=✓ value to forms so that old Internet Explorer versions are more likely to submit the form as utf-8. Additionally, the server can check the value against wrong encodings of the checkmark character and detect that a query string or application/x-www-form-urlencoded body was *not* sent as utf-8, eg. if the form had an accept-charset parameter or the containing page had a different character set.

**qs** supports this mechanism via the charsetSentinel option. If specified, the utf8 parameter will be omitted from the returned object. It will be used to switch to iso-8859-1/utf-8 mode depending on how the checkmark is encoded.

**Important**: When you specify both the charset option and the charsetSentinel option, the charset will be overridden when the request contains a utf8 parameter from which the actual charset can be deduced. In that sense the charset will behave as the default charset rather than the authoritative charset.

var detectedAsUtf8 = qs.parse('utf8=%E2%9C%93&a=%C3%B8', {

charset: 'iso-8859-1',

charsetSentinel: true

});

assert.deepEqual(detectedAsUtf8, { a: 'ø' });

// Browsers encode the checkmark as &#10003; when submitting as iso-8859-1:

var detectedAsIso8859\_1 = qs.parse('utf8=%26%2310003%3B&a=%F8', {

charset: 'utf-8',

charsetSentinel: true

});

assert.deepEqual(detectedAsIso8859\_1, { a: 'ø' });

If you want to decode the &#...; syntax to the actual character, you can specify the interpretNumericEntities option as well:

var detectedAsIso8859\_1 = qs.parse('a=%26%239786%3B', {

charset: 'iso-8859-1',

interpretNumericEntities: true

});

assert.deepEqual(detectedAsIso8859\_1, { a: '☺' });

It also works when the charset has been detected in charsetSentinel mode.

### Parsing Arrays

**qs** can also parse arrays using a similar [] notation:

var withArray = qs.parse('a[]=b&a[]=c');

assert.deepEqual(withArray, { a: ['b', 'c'] });

You may specify an index as well:

var withIndexes = qs.parse('a[1]=c&a[0]=b');

assert.deepEqual(withIndexes, { a: ['b', 'c'] });

Note that the only difference between an index in an array and a key in an object is that the value between the brackets must be a number to create an array. When creating arrays with specific indices, **qs** will compact a sparse array to only the existing values preserving their order:

var noSparse = qs.parse('a[1]=b&a[15]=c');

assert.deepEqual(noSparse, { a: ['b', 'c'] });

You may also use allowSparse option to parse sparse arrays:

var sparseArray = qs.parse('a[1]=2&a[3]=5', { allowSparse: true });

assert.deepEqual(sparseArray, { a: [, '2', , '5'] });

Note that an empty string is also a value, and will be preserved:

var withEmptyString = qs.parse('a[]=&a[]=b');

assert.deepEqual(withEmptyString, { a: ['', 'b'] });

var withIndexedEmptyString = qs.parse('a[0]=b&a[1]=&a[2]=c');

assert.deepEqual(withIndexedEmptyString, { a: ['b', '', 'c'] });

**qs** will also limit specifying indices in an array to a maximum index of 20. Any array members with an index of greater than 20 will instead be converted to an object with the index as the key. This is needed to handle cases when someone sent, for example, a[999999999] and it will take significant time to iterate over this huge array.

var withMaxIndex = qs.parse('a[100]=b');

assert.deepEqual(withMaxIndex, { a: { '100': 'b' } });

This limit can be overridden by passing an arrayLimit option:

var withArrayLimit = qs.parse('a[1]=b', { arrayLimit: 0 });

assert.deepEqual(withArrayLimit, { a: { '1': 'b' } });

To disable array parsing entirely, set parseArrays to false.

var noParsingArrays = qs.parse('a[]=b', { parseArrays: false });

assert.deepEqual(noParsingArrays, { a: { '0': 'b' } });

If you mix notations, **qs** will merge the two items into an object:

var mixedNotation = qs.parse('a[0]=b&a[b]=c');

assert.deepEqual(mixedNotation, { a: { '0': 'b', b: 'c' } });

You can also create arrays of objects:

var arraysOfObjects = qs.parse('a[][b]=c');

assert.deepEqual(arraysOfObjects, { a: [{ b: 'c' }] });

Some people use comma to join array, **qs** can parse it:

var arraysOfObjects = qs.parse('a=b,c', { comma: true })

assert.deepEqual(arraysOfObjects, { a: ['b', 'c'] })

(*this cannot convert nested objects, such as a={b:1},{c:d}*)

### Parsing primitive/scalar values (numbers, booleans, null, etc)

By default, all values are parsed as strings. This behavior will not change and is explained in [issue #91](https://github.com/ljharb/qs/issues/91).

var primitiveValues = qs.parse('a=15&b=true&c=null');

assert.deepEqual(primitiveValues, { a: '15', b: 'true', c: 'null' });

If you wish to auto-convert values which look like numbers, booleans, and other values into their primitive counterparts, you can use the [query-types Express JS middleware](https://github.com/xpepermint/query-types) which will auto-convert all request query parameters.

### Stringifying

qs.stringify(object, [options]);

When stringifying, **qs** by default URI encodes output. Objects are stringified as you would expect:

assert.equal(qs.stringify({ a: 'b' }), 'a=b');

assert.equal(qs.stringify({ a: { b: 'c' } }), 'a%5Bb%5D=c');

This encoding can be disabled by setting the encode option to false:

var unencoded = qs.stringify({ a: { b: 'c' } }, { encode: false });

assert.equal(unencoded, 'a[b]=c');

Encoding can be disabled for keys by setting the encodeValuesOnly option to true:

var encodedValues = qs.stringify(

{ a: 'b', c: ['d', 'e=f'], f: [['g'], ['h']] },

{ encodeValuesOnly: true }

);

assert.equal(encodedValues,'a=b&c[0]=d&c[1]=e%3Df&f[0][0]=g&f[1][0]=h');

This encoding can also be replaced by a custom encoding method set as encoder option:

var encoded = qs.stringify({ a: { b: 'c' } }, { encoder: function (str) {

// Passed in values `a`, `b`, `c`

return // Return encoded string

}})

*(Note: the encoder option does not apply if encode is false)*

Analogue to the encoder there is a decoder option for parse to override decoding of properties and values:

var decoded = qs.parse('x=z', { decoder: function (str) {

// Passed in values `x`, `z`

return // Return decoded string

}})

You can encode keys and values using different logic by using the type argument provided to the encoder:

var encoded = qs.stringify({ a: { b: 'c' } }, { encoder: function (str, defaultEncoder, charset, type) {

if (type === 'key') {

return // Encoded key

} else if (type === 'value') {

return // Encoded value

}

}})

The type argument is also provided to the decoder:

var decoded = qs.parse('x=z', { decoder: function (str, defaultDecoder, charset, type) {

if (type === 'key') {

return // Decoded key

} else if (type === 'value') {

return // Decoded value

}

}})

Examples beyond this point will be shown as though the output is not URI encoded for clarity. Please note that the return values in these cases *will* be URI encoded during real usage.

When arrays are stringified, by default they are given explicit indices:

qs.stringify({ a: ['b', 'c', 'd'] });

// 'a[0]=b&a[1]=c&a[2]=d'

You may override this by setting the indices option to false:

qs.stringify({ a: ['b', 'c', 'd'] }, { indices: false });

// 'a=b&a=c&a=d'

You may use the arrayFormat option to specify the format of the output array:

qs.stringify({ a: ['b', 'c'] }, { arrayFormat: 'indices' })

// 'a[0]=b&a[1]=c'

qs.stringify({ a: ['b', 'c'] }, { arrayFormat: 'brackets' })

// 'a[]=b&a[]=c'

qs.stringify({ a: ['b', 'c'] }, { arrayFormat: 'repeat' })

// 'a=b&a=c'

qs.stringify({ a: ['b', 'c'] }, { arrayFormat: 'comma' })

// 'a=b,c'

Note: when using arrayFormat set to 'comma', you can also pass the commaRoundTrip option set to true or false, to append [] on single-item arrays, so that they can round trip through a parse.

When objects are stringified, by default they use bracket notation:

qs.stringify({ a: { b: { c: 'd', e: 'f' } } });

// 'a[b][c]=d&a[b][e]=f'

You may override this to use dot notation by setting the allowDots option to true:

qs.stringify({ a: { b: { c: 'd', e: 'f' } } }, { allowDots: true });

// 'a.b.c=d&a.b.e=f'

Empty strings and null values will omit the value, but the equals sign (=) remains in place:

assert.equal(qs.stringify({ a: '' }), 'a=');

Key with no values (such as an empty object or array) will return nothing:

assert.equal(qs.stringify({ a: [] }), '');

assert.equal(qs.stringify({ a: {} }), '');

assert.equal(qs.stringify({ a: [{}] }), '');

assert.equal(qs.stringify({ a: { b: []} }), '');

assert.equal(qs.stringify({ a: { b: {}} }), '');

Properties that are set to undefined will be omitted entirely:

assert.equal(qs.stringify({ a: null, b: undefined }), 'a=');

The query string may optionally be prepended with a question mark:

assert.equal(qs.stringify({ a: 'b', c: 'd' }, { addQueryPrefix: true }), '?a=b&c=d');

The delimiter may be overridden with stringify as well:

assert.equal(qs.stringify({ a: 'b', c: 'd' }, { delimiter: ';' }), 'a=b;c=d');

If you only want to override the serialization of Date objects, you can provide a serializeDate option:

var date = new Date(7);

assert.equal(qs.stringify({ a: date }), 'a=1970-01-01T00:00:00.007Z'.replace(/:/g, '%3A'));

assert.equal(

qs.stringify({ a: date }, { serializeDate: function (d) { return d.getTime(); } }),

'a=7'

);

You may use the sort option to affect the order of parameter keys:

function alphabeticalSort(a, b) {

return a.localeCompare(b);

}

assert.equal(qs.stringify({ a: 'c', z: 'y', b : 'f' }, { sort: alphabeticalSort }), 'a=c&b=f&z=y');

Finally, you can use the filter option to restrict which keys will be included in the stringified output. If you pass a function, it will be called for each key to obtain the replacement value. Otherwise, if you pass an array, it will be used to select properties and array indices for stringification:

function filterFunc(prefix, value) {

if (prefix == 'b') {

// Return an `undefined` value to omit a property.

return;

}

if (prefix == 'e[f]') {

return value.getTime();

}

if (prefix == 'e[g][0]') {

return value \* 2;

}

return value;

}

qs.stringify({ a: 'b', c: 'd', e: { f: new Date(123), g: [2] } }, { filter: filterFunc });

// 'a=b&c=d&e[f]=123&e[g][0]=4'

qs.stringify({ a: 'b', c: 'd', e: 'f' }, { filter: ['a', 'e'] });

// 'a=b&e=f'

qs.stringify({ a: ['b', 'c', 'd'], e: 'f' }, { filter: ['a', 0, 2] });

// 'a[0]=b&a[2]=d'

### Handling of null values

By default, null values are treated like empty strings:

var withNull = qs.stringify({ a: null, b: '' });

assert.equal(withNull, 'a=&b=');

Parsing does not distinguish between parameters with and without equal signs. Both are converted to empty strings.

var equalsInsensitive = qs.parse('a&b=');

assert.deepEqual(equalsInsensitive, { a: '', b: '' });

To distinguish between null values and empty strings use the strictNullHandling flag. In the result string the null values have no = sign:

var strictNull = qs.stringify({ a: null, b: '' }, { strictNullHandling: true });

assert.equal(strictNull, 'a&b=');

To parse values without = back to null use the strictNullHandling flag:

var parsedStrictNull = qs.parse('a&b=', { strictNullHandling: true });

assert.deepEqual(parsedStrictNull, { a: null, b: '' });

To completely skip rendering keys with null values, use the skipNulls flag:

var nullsSkipped = qs.stringify({ a: 'b', c: null}, { skipNulls: true });

assert.equal(nullsSkipped, 'a=b');

If you're communicating with legacy systems, you can switch to iso-8859-1 using the charset option:

var iso = qs.stringify({ æ: 'æ' }, { charset: 'iso-8859-1' });

assert.equal(iso, '%E6=%E6');

Characters that don't exist in iso-8859-1 will be converted to numeric entities, similar to what browsers do:

var numeric = qs.stringify({ a: '☺' }, { charset: 'iso-8859-1' });

assert.equal(numeric, 'a=%26%239786%3B');

You can use the charsetSentinel option to announce the character by including an utf8=✓ parameter with the proper encoding if the checkmark, similar to what Ruby on Rails and others do when submitting forms.

var sentinel = qs.stringify({ a: '☺' }, { charsetSentinel: true });

assert.equal(sentinel, 'utf8=%E2%9C%93&a=%E2%98%BA');

var isoSentinel = qs.stringify({ a: 'æ' }, { charsetSentinel: true, charset: 'iso-8859-1' });

assert.equal(isoSentinel, 'utf8=%26%2310003%3B&a=%E6');

### Dealing with special character sets

By default the encoding and decoding of characters is done in utf-8, and iso-8859-1 support is also built in via the charset parameter.

If you wish to encode querystrings to a different character set (i.e. [Shift JIS](https://en.wikipedia.org/wiki/Shift_JIS)) you can use the [qs-iconv](https://github.com/martinheidegger/qs-iconv) library:

var encoder = require('qs-iconv/encoder')('shift\_jis');

var shiftJISEncoded = qs.stringify({ a: 'こんにちは！' }, { encoder: encoder });

assert.equal(shiftJISEncoded, 'a=%82%B1%82%F1%82%C9%82%BF%82%CD%81I');

This also works for decoding of query strings:

var decoder = require('qs-iconv/decoder')('shift\_jis');

var obj = qs.parse('a=%82%B1%82%F1%82%C9%82%BF%82%CD%81I', { decoder: decoder });

assert.deepEqual(obj, { a: 'こんにちは！' });

### RFC 3986 and RFC 1738 space encoding

RFC3986 used as default option and encodes ' ' to *%20* which is backward compatible. In the same time, output can be stringified as per RFC1738 with ' ' equal to '+'.

assert.equal(qs.stringify({ a: 'b c' }), 'a=b%20c');

assert.equal(qs.stringify({ a: 'b c' }, { format : 'RFC3986' }), 'a=b%20c');

assert.equal(qs.stringify({ a: 'b c' }, { format : 'RFC1738' }), 'a=b+c');

## Security

Please email [@ljharb](https://github.com/ljharb) or see <https://tidelift.com/security> if you have a potential security vulnerability to report.

## qs for enterprise

Available as part of the Tidelift Subscription

The maintainers of qs and thousands of other packages are working with Tidelift to deliver commercial support and maintenance for the open source dependencies you use to build your applications. Save time, reduce risk, and improve code health, while paying the maintainers of the exact dependencies you use. [Learn more.](https://tidelift.com/subscription/pkg/npm-qs?utm_source=npm-qs&utm_medium=referral&utm_campaign=enterprise&utm_term=repo)