

NAME

Cpanel::JSON::XS – cPanel fork of JSON::XS, fast and correct serializing

SYNOPSIS

```
use Cpanel::JSON::XS;

# exported functions, they croak on error
# and expect/generate UTF-8

$utf8_encoded_json_text = encode_json $perl_hash_or_arrayref;
$perl_hash_or_arrayref = decode_json $utf8_encoded_json_text;

# OO-interface

$coder = Cpanel::JSON::XS->new->ascii->pretty->allow_nonref;
$pretty_printed_unencoded = $coder->encode ($perl_scalar);
$perl_scalar = $coder->decode ($unicode_json_text);

# Note that 5.6 misses most smart utf8 and encoding functionalities
# of newer releases.

# Note that L<JSON::MaybeXS> will automatically use Cpanel::JSON::XS
# if available, at virtually no speed overhead either, so you should
# be able to just:

use JSON::MaybeXS;

# and do the same things, except that you have a pure-perl fallback now.

Note that this module will be replaced by a new JSON::Safe module soon,
with the same API just guaranteed safe defaults.
```

DESCRIPTION

This module converts Perl data structures to JSON and vice versa. Its primary goal is to be *correct* and its secondary goal is to be *fast*. To reach the latter goal it was written in C.

As this is the n-th-something JSON module on CPAN, what was the reason to write yet another JSON module? While it seems there are many JSON modules, none of them correctly handle all corner cases, and in most cases their maintainers are unresponsive, gone missing, or not listening to bug reports for other reasons.

See below for the cPanel fork.

See MAPPING, below, on how Cpanel::JSON::XS maps perl values to JSON values and vice versa.

FEATURES

- correct Unicode handling

This module knows how to handle Unicode with Perl version higher than 5.8.5, documents how and when it does so, and even documents what “correct” means.

- round-trip integrity

When you serialize a perl data structure using only data types supported by JSON and Perl, the deserialized data structure is identical on the Perl level. (e.g. the string “2.0” doesn’t suddenly become “2” just because it looks like a number). There *are* minor exceptions to this, read the MAPPING section below to learn about those.

- strict checking of JSON correctness

There is no guessing, no generating of illegal JSON texts by default, and only JSON is accepted as

input by default. the latter is a security feature.

- fast

Compared to other JSON modules and other serializers such as Storable, this module usually compares favourably in terms of speed, too.

- simple to use

This module has both a simple functional interface as well as an object oriented interface.

- reasonably versatile output formats

You can choose between the most compact guaranteed-single-line format possible (nice for simple line-based protocols), a pure-ASCII format (for when your transport is not 8-bit clean, still supports the whole Unicode range), or a pretty-printed format (for when you want to read that stuff). Or you can combine those features in whatever way you like.

cPanel fork

Since the original author MLEHMANN has no public bugtracker, this cPanel fork sits now on github.

src repo: <<https://github.com/rurban/Cpanel-JSON-XS>> original: <<http://cvs.schmorp.de/JSON-XS/>>

RT: <https://github.com/rurban/Cpanel-JSON-XS/issues> or
<<https://rt.cpan.org/Public/Dist/Display.html?Queue=Cpanel-JSON-XS>>

Changes to JSON::XS

- stricter **decode_json()** as documented. non-refs are disallowed.
added a 2nd optional argument. **decode()** honors now `allow_nonref`.
- fixed encode of numbers for dual-vars. Different string representations are preserved, but numbers with temporary strings which represent the same number are here treated as numbers, not strings. Cpanel::JSON::XS is a bit slower, but preserves numeric types better.
- numbers ending with .0 stray numbers, are not converted to integers. [#63] dual-vars which are represented as number not integer (42+“bar” != 5.8.9) are now encoded as number (=> 42.0) because internally it’s now a NOK type. However !!1 which is wrongly encoded in 5.8 as “1”/1.0 is still represented as integer.
- different handling of inf/nan. Default now to null, optionally with **stringify_infnan()** to “inf”/“nan”. [#28, #32]
- added `binary` extension, non-JSON and non JSON parsable, allows `\xNN` and `\NNN` sequences.
- 5.6.2 support; sacrificing some utf8 features (assuming bytes all-over), no multi-byte unicode characters with 5.6.
- interop for true/false overloading. JSON::XS, JSON::PP and Mojo::JSON representations for booleans are accepted and JSON::XS accepts Cpanel::JSON::XS booleans [#13, #37]
Fixed overloading of booleans. Cpanel::JSON::XS::true stringifies again to “1”, not “true”, analog to all other JSON modules.
- native boolean mapping of yes and no to true and false, as in YAML::XS.
In perl !0 is yes, !1 is no.
The JSON value true maps to 1, false maps to 0. [#39]
- support arbitrary stringification with `encode`, with `convert_blessed` and `allow_blessed`.
- ithread support. Cpanel::JSON::XS is thread-safe, JSON::XS not

- is_bool can be called as method, JSON::XS::is_bool not.
- performance optimizations for threaded Perls
- relaxed mode, allowing many popular extensions
- additional fixes for:
 - [cpan #88061] AIX atof without USE_LONG_DOUBLE
 - #10 unshare_hek crash
 - #7, #29 avoid re-blessing where possible. It fails in JSON::XS for READONLY values, i.e. restricted hashes.
 - #41 overloading of booleans, use the object not the reference.
 - #62 -Dusequadmath conversion and no SEGV.
 - #72 parsing of values followed \0, like 1\0 does fail.
 - #72 parsing of illegal unicode or non-unicode characters.
 - #96 locale-insensitive numeric conversion
 - #154 numeric conversion fixed since 5.22, using the same strtold as perl5.
- public maintenance and bugtracker
- use ppport.h, sanify XS.xs comment styles, harness C coding style
- common::sense is optional. When available it is not used in the published production module, just during development and testing.
- extended testsuite, passes all http://seriot.ch/parsing_json.html tests. In fact it is the only know JSON decoder which does so, while also being the fastest.
- support many more options and methods from JSON::PP:
 - stringify_infnan, allow_unknown, allow_stringify, allow_barekey, encode_stringify, allow_bignum, allow_singlequote, sort_by (partially), escape_slash, convert_blessed, ... optional decode_json(, allow_nonref) arg.
 - relaxed implements allow_dupkeys.
- support all 5 unicode BOM's: UTF-8, UTF-16LE, UTF-16BE, UTF-32LE, UTF-32BE, encoding internally to UTF-8.

FUNCTIONAL INTERFACE

The following convenience methods are provided by this module. They are exported by default:

```
$json_text = encode_json $perl_scalar, [json_type]
```

Converts the given Perl data structure to a UTF-8 encoded, binary string (that is, the string contains octets only). Croaks on error.

This function call is functionally identical to:

```
$json_text = Cpanel::JSON::XS->new->utf8->encode ($perl_scalar, $json_type)
```

Except being faster.

For the type argument see Cpanel::JSON::XS::Type.

```
$perl_scalar = decode_json $json_text [, $allow_nonref [, my $json_type ]]
```

The opposite of `encode_json`: expects an UTF-8 (binary) string of an json reference and tries to parse that as an UTF-8 encoded JSON text, returning the resulting reference. Croaks on error.

This function call is functionally identical to:

```
$perl_scalar = Cpanel::JSON::XS->new->utf8->decode ($json_text, $json_type)
```

except being faster.

Note that older `decode_json` versions in `Cpanel::JSON::XS` older than 3.0116 and `JSON::XS` did not set `allow_nonref` but allowed them due to a bug in the decoder.

If the new optional `$allow_nonref` argument is set and not false, the `allow_nonref` option will be set and the function will act is described as in the relaxed RFC 7159 allowing all values such as objects, arrays, strings, numbers, “null”, “true”, and “false”.

For the type argument see `Cpanel::JSON::XS::Type`.

```
$is_boolean = Cpanel::JSON::XS::is_bool $scalar
```

Returns true if the passed scalar represents either `JSON::XS::true` or `JSON::XS::false`, two constants that act like 1 and 0, respectively and are used to represent JSON `true` and `false` values in Perl.

See MAPPING, below, for more information on how JSON values are mapped to Perl.

DEPRECATED FUNCTIONS

`from_json`

`from_json` has been renamed to `decode_json`

`to_json`

`to_json` has been renamed to `encode_json`

A FEW NOTES ON UNICODE AND PERL

Since this often leads to confusion, here are a few very clear words on how Unicode works in Perl, modulo bugs.

1. Perl strings can store characters with ordinal values > 255.

This enables you to store Unicode characters as single characters in a Perl string – very natural.

2. Perl does *not* associate an encoding with your strings.

... until you force it to, e.g. when matching it against a regex, or printing the scalar to a file, in which case Perl either interprets your string as locale-encoded text, octets/binary, or as Unicode, depending on various settings. In no case is an encoding stored together with your data, it is *use* that decides encoding, not any magical meta data.

3. The internal utf-8 flag has no meaning with regards to the encoding of your string.

4. A “Unicode String” is simply a string where each character can be validly interpreted as a Unicode code point.

If you have UTF-8 encoded data, it is no longer a Unicode string, but a Unicode string encoded in UTF-8, giving you a binary string.

5. A string containing “high” (> 255) character values is *not* a UTF-8 string.

6. Unicode noncharacters only warn, as in core.

The 66 Unicode noncharacters `U+FDD0..U+FDEF`, and `U+*FFFE`, `U+*FFFF` just warn, see <http://www.unicode.org/versions/corrigendum9.html>>. But illegal surrogate pairs fail to parse.

7. Raw non-Unicode characters above `U+10FFFF` are disallowed.

Raw non-Unicode characters outside the valid unicode range fail to parse, because “A string is a sequence of zero or more Unicode characters” RFC 7159 section 1 and “JSON text SHALL be encoded in Unicode RFC 7159 section 8.1. We use now the `UTF8_DISALLOW_SUPER` flag when parsing unicode.

I hope this helps :)

OBJECT-ORIENTED INTERFACE

The object oriented interface lets you configure your own encoding or decoding style, within the limits of supported formats.

```
$json = new Cpanel::JSON::XS
```

Creates a new JSON object that can be used to de/encode JSON strings. All boolean flags described below are by default *disabled*.

The mutators for flags all return the JSON object again and thus calls can be chained:

```
my $json = Cpanel::JSON::XS->new->utf8->space_after->encode ({a => [1,2]})
=> {"a": [1, 2]}
```

```
$json = $json->ascii([$enable])
```

```
$enabled = $json->get_ascii
```

If `$enable` is true (or missing), then the `encode` method will not generate characters outside the code range 0..127 (which is ASCII). Any Unicode characters outside that range will be escaped using either a single `\uXXXX` (BMP characters) or a double `\uHHHH\uLLLLL` escape sequence, as per RFC4627. The resulting encoded JSON text can be treated as a native Unicode string, an ascii-encoded, latin1-encoded or UTF-8 encoded string, or any other superset of ASCII.

If `$enable` is false, then the `encode` method will not escape Unicode characters unless required by the JSON syntax or other flags. This results in a faster and more compact format.

See also the section *ENCODING/CODESET FLAG NOTES* later in this document.

The main use for this flag is to produce JSON texts that can be transmitted over a 7-bit channel, as the encoded JSON texts will not contain any 8 bit characters.

```
Cpanel::JSON::XS->new->ascii(1)->encode ([chr 0x10401])
=> ["\ud801\udc01"]
```

```
$json = $json->latin1([$enable])
```

```
$enabled = $json->get_latin1
```

If `$enable` is true (or missing), then the `encode` method will encode the resulting JSON text as latin1 (or ISO-8859-1), escaping any characters outside the code range 0..255. The resulting string can be treated as a latin1-encoded JSON text or a native Unicode string. The `decode` method will not be affected in any way by this flag, as `decode` by default expects Unicode, which is a strict superset of latin1.

If `$enable` is false, then the `encode` method will not escape Unicode characters unless required by the JSON syntax or other flags.

See also the section *ENCODING/CODESET FLAG NOTES* later in this document.

The main use for this flag is efficiently encoding binary data as JSON text, as most octets will not be escaped, resulting in a smaller encoded size. The disadvantage is that the resulting JSON text is encoded in latin1 (and must correctly be treated as such when storing and transferring), a rare encoding for JSON. It is therefore most useful when you want to store data structures known to contain binary data efficiently in files or databases, not when talking to other JSON encoders/decoders.

```
Cpanel::JSON::XS->new->latin1->encode (["\x{89}\x{abc}"])
=> ["\x{89}\\u0abc"] # (perl syntax, U+abc escaped, U+89 not)
```

```
$json = $json->binary([$enable])
```

```
$enabled = $json = $json->get_binary
```

If the `$enable` argument is true (or missing), then the `encode` method will not try to detect an UTF-8 encoding in any JSON string, it will strictly interpret it as byte sequence. The result might contain new `\xNN` sequences, which is **unparsable JSON**. The `decode` method forbids `\uNNNN` sequences and accepts `\xNN` and octal `\NNN` sequences.

There is also a special logic for perl 5.6 and utf8. 5.6 encodes any string to utf-8 automatically when

seeing a codepoint $\geq 0x80$ and $< 0x100$. With the binary flag enabled decode the perl utf8 encoded string to the original byte encoding and encode this with `\xNN` escapes. This will result to the same encodings as with newer perls. But note that binary multi-byte codepoints with 5.6 will result in illegal unicode character in binary string errors, unlike with newer perls.

If `$enable` is false, then the `encode` method will smartly try to detect Unicode characters unless required by the JSON syntax or other flags and hex and octal sequences are forbidden.

See also the section *ENCODING/CODESET FLAG NOTES* later in this document.

The main use for this flag is to avoid the smart unicode detection and possible double encoding. The disadvantage is that the resulting JSON text is encoded in new `\xNN` and in latin1 characters and must correctly be treated as such when storing and transferring, a rare encoding for JSON. It will produce non-readable JSON strings in the browser. It is therefore most useful when you want to store data structures known to contain binary data efficiently in files or databases, not when talking to other JSON encoders/decoders. The binary decoding method can also be used when an encoder produced a non-JSON conformant hex or octal encoding `\xNN` or `\NNN`.

```
Cpanel::JSON::XS->new->binary->encode (["\x{89}\x{abc}"])
5.6:   Error: malformed or illegal unicode character in binary string
>=5.8: ['\x89\xe0\xaa\xbc']
```

```
Cpanel::JSON::XS->new->binary->encode (["\x{89}\x{bc}"])
=> ["\x89\xbc"]
```

```
Cpanel::JSON::XS->new->binary->decode (["\x89\ua001"])
Error: malformed or illegal unicode character in binary string
```

```
Cpanel::JSON::XS->new->decode (["\x89"])
Error: illegal hex character in non-binary string
```

```
$json = $json->utf8 ($enable)
$enabled = $json->get_utf8
```

If `$enable` is true (or missing), then the `encode` method will encode the JSON result into UTF-8, as required by many protocols, while the `decode` method expects to be handled an UTF-8-encoded string. Please note that UTF-8-encoded strings do not contain any characters outside the range 0..255, they are thus useful for bitwise/binary I/O. In future versions, enabling this option might enable autodetection of the UTF-16 and UTF-32 encoding families, as described in RFC4627.

If `$enable` is false, then the `encode` method will return the JSON string as a (non-encoded) Unicode string, while `decode` expects thus a Unicode string. Any decoding or encoding (e.g. to UTF-8 or UTF-16) needs to be done yourself, e.g. using the `Encode` module.

See also the section *ENCODING/CODESET FLAG NOTES* later in this document.

Example, output UTF-16BE-encoded JSON:

```
use Encode;
$json_text = encode "UTF-16BE", Cpanel::JSON::XS->new->encode ($object);
```

Example, decode UTF-32LE-encoded JSON:

```
use Encode;
$object = Cpanel::JSON::XS->new->decode (decode "UTF-32LE", $json_text);
```

```
$json = $json->pretty ($enable)
```

This enables (or disables) all of the `indent`, `space_before` and `space_after` (and in the future possibly more) flags in one call to generate the most readable (or most compact) form possible.

Example, pretty-print some simple structure:

```

my $json = Cpanel::JSON::XS->new->pretty(1)->encode ({a => [1,2]})
=>
{
    "a" : [
        1,
        2
    ]
}

```

```
$json = $json->indent([$enable])
```

```
$enabled = $json->get_indent
```

If `$enable` is true (or missing), then the `encode` method will use a multiline format as output, putting every array member or object/hash key-value pair into its own line, indenting them properly.

If `$enable` is false, no newlines or indenting will be produced, and the resulting JSON text is guaranteed not to contain any newlines.

This setting has no effect when decoding JSON texts.

```
$json = $json->indent_length([$number_of_spaces])
```

```
$length = $json->get_indent_length()
```

Set the indent length (default 3). This option is only useful when you also enable `indent` or `pretty`. The acceptable range is from 0 (no indentation) to 15

```
$json = $json->space_before([$enable])
```

```
$enabled = $json->get_space_before
```

If `$enable` is true (or missing), then the `encode` method will add an extra optional space before the `:` separating keys from values in JSON objects.

If `$enable` is false, then the `encode` method will not add any extra space at those places.

This setting has no effect when decoding JSON texts. You will also most likely combine this setting with `space_after`.

Example, `space_before` enabled, `space_after` and `indent` disabled:

```
{ "key" : "value" }
```

```
$json = $json->space_after([$enable])
```

```
$enabled = $json->get_space_after
```

If `$enable` is true (or missing), then the `encode` method will add an extra optional space after the `:` separating keys from values in JSON objects and extra whitespace after the `,` separating key-value pairs and array members.

If `$enable` is false, then the `encode` method will not add any extra space at those places.

This setting has no effect when decoding JSON texts.

Example, `space_before` and `indent` disabled, `space_after` enabled:

```
{ "key": "value" }
```

```
$json = $json->relaxed([$enable])
```

```
$enabled = $json->get_relaxed
```

If `$enable` is true (or missing), then `decode` will accept some extensions to normal JSON syntax (see below). `encode` will not be affected in anyway. *Be aware that this option makes you accept invalid JSON texts as if they were valid!* I suggest only to use this option to parse application-specific files written by humans (configuration files, resource files etc.)

If `$enable` is false (the default), then `decode` will only accept valid JSON texts.

Currently accepted extensions are:

- list items can have an end-comma

JSON *separates* array elements and key-value pairs with commas. This can be annoying if you write JSON texts manually and want to be able to quickly append elements, so this extension accepts comma at the end of such items not just between them:

```
[
  1,
  2, <- this comma not normally allowed
]
{
  "k1": "v1",
  "k2": "v2", <- this comma not normally allowed
}
```

- shell-style '#'-comments

Whenever JSON allows whitespace, shell-style comments are additionally allowed. They are terminated by the first carriage-return or line-feed character, after which more white-space and comments are allowed.

```
[
  1, # this comment not allowed in JSON
    # neither this one...
]
```

- literal ASCII TAB characters in strings

Literal ASCII TAB characters are now allowed in strings (and treated as `\t`) in relaxed mode. Despite JSON mandates, that TAB character is substituted for “`\t`” sequence.

```
[
  "Hello\tWorld",
  "Hello<TAB>World", # literal <TAB> would not normally be allowed
]
```

- `allow_singlequote`

Single quotes are accepted instead of double quotes. See the “`allow_singlequote`” option.

```
{ "foo": 'bar' }
{ 'foo': "bar" }
{ 'foo': 'bar' }
```

- `allow_barekey`

Accept unquoted object keys instead of with mandatory double quotes. See the “`allow_barekey`” option.

```
{ foo:"bar" }
```

- `allow_dupkeys`

Allow decoding of duplicate keys in hashes. By default duplicate keys are forbidden. See http://seriot.ch/parsing_json.php#24: RFC 7159 section 4: “The names within an object should be unique.” See the “`allow_dupkeys`” option.

```
$json = $json->canonical([$enable])
```

```
$enabled = $json->get_canonical
```

If `$enable` is true (or missing), then the `encode` method will output JSON objects by sorting their keys. This is adding a comparatively high overhead.

If `$enable` is false, then the `encode` method will output key-value pairs in the order Perl stores them (which will likely change between runs of the same script, and can change even within the same

run from 5.18 onwards).

This option is useful if you want the same data structure to be encoded as the same JSON text (given the same overall settings). If it is disabled, the same hash might be encoded differently even if contains the same data, as key-value pairs have no inherent ordering in Perl.

This setting has no effect when decoding JSON texts.

This setting has currently no effect on tied hashes.

```
$json = $json->sort_by (undef, 0, 1 or a block)
```

This currently only (un)sets the `canonical` option, and ignores custom sort blocks.

This setting has no effect when decoding JSON texts.

This setting has currently no effect on tied hashes.

```
$json = $json->escape_slash ([$enable])
```

```
$enabled = $json->get_escape_slash
```

According to the JSON Grammar, the *forward slash* character (U+002F) `"/"` need to be escaped. But by default strings are encoded without escaping slashes in all perl JSON encoders.

If `$enable` is true (or missing), then `encode` will escape slashes, `"\"/`.

This setting has no effect when decoding JSON texts.

```
$json = $json->unblessed_bool ([$enable])
```

```
$enabled = $json->get_unblessed_bool
```

```
    $json = $json->unblessed_bool ([$enable])
```

If `$enable` is true (or missing), then `decode` will return Perl non-object boolean variables (1 and 0) for JSON booleans (`true` and `false`). If `$enable` is false, then `decode` will return `Cpanel::JSON::XS::Boolean` objects for JSON booleans.

```
$json = $json->allow_singlequote ([$enable])
```

```
$enabled = $json->get_allow_singlequote
```

```
    $json = $json->allow_singlequote ([$enable])
```

If `$enable` is true (or missing), then `decode` will accept JSON strings quoted by single quotations that are invalid JSON format.

```
    $json->allow_singlequote->decode({"foo":'bar'});
```

```
    $json->allow_singlequote->decode({'foo':"bar"});
```

```
    $json->allow_singlequote->decode({'foo':'bar'});
```

This is also enabled with `relaxed`. As same as the `relaxed` option, this option may be used to parse application-specific files written by humans.

```
$json = $json->allow_barekey ([$enable])
```

```
$enabled = $json->get_allow_barekey
```

```
    $json = $json->allow_barekey ([$enable])
```

If `$enable` is true (or missing), then `decode` will accept bare keys of JSON object that are invalid JSON format.

Same as with the `relaxed` option, this option may be used to parse application-specific files written by humans.

```
    $json->allow_barekey->decode(' {foo:"bar"} ');
```

```
$json = $json->allow_bignum ([$enable])
```

```
$enabled = $json->get_allow_bignum
```

```
    $json = $json->allow_bignum ([$enable])
```

If `$enable` is true (or missing), then `decode` will convert the big integer Perl cannot handle as integer into a `Math::BigInt` object and convert a floating number (any) into a `Math::BigFloat`.

On the contrary, `encode` converts `Math::BigInt` objects and `Math::BigFloat` objects into JSON numbers with `allow_blessed` enable.

```
$json->allow_nonref->allow_blessed->allow_bignum;
$bigfloat = $json->decode('2.00000000000000000000000000000001');
print $json->encode($bigfloat);
# => 2.00000000000000000000000000000001
```

See “MAPPING” about the normal conversion of JSON number.

```
$json = $json->allow_bigint([$enable])
```

This option is obsolete and replaced by `allow_bignum`.

```
$json = $json->allow_nonref([$enable])
```

```
$enabled = $json->get_allow_nonref
```

If `$enable` is true (or missing), then the `encode` method can convert a non-reference into its corresponding string, number or null JSON value, which is an extension to RFC4627. Likewise, `decode` will accept those JSON values instead of croaking.

If `$enable` is false, then the `encode` method will croak if it isn't passed an arrayref or hashref, as JSON texts must either be an object or array. Likewise, `decode` will croak if given something that is not a JSON object or array.

Example, encode a Perl scalar as JSON value with enabled `allow_nonref`, resulting in an invalid JSON text:

```
Cpanel::JSON::XS->new->allow_nonref->encode ("Hello, World!")
=> "Hello, World!"
```

```
$json = $json->allow_unknown([$enable])
```

```
$enabled = $json->get_allow_unknown
```

If `$enable` is true (or missing), then `encode` will *not* throw an exception when it encounters values it cannot represent in JSON (for example, filehandles) but instead will encode a JSON null value. Note that blessed objects are not included here and are handled separately by `c<allow_nonref>`.

If `$enable` is false (the default), then `encode` will throw an exception when it encounters anything it cannot encode as JSON.

This option does not affect `decode` in any way, and it is recommended to leave it off unless you know your communications partner.

```
$json = $json->allow_stringify([$enable])
```

```
$enabled = $json->get_allow_stringify
```

If `$enable` is true (or missing), then `encode` will stringify the non-object perl value or reference. Note that blessed objects are not included here and are handled separately by `allow_blessed` and `convert_blessed`. String references are stringified to the string value, other references as in perl.

This option does not affect `decode` in any way.

This option is special to this module, it is not supported by other encoders. So it is not recommended to use it.

```
$json = $json->require_types([$enable])
```

```
$enable = $json->get_require_types
```

```
    $json = $json->require_types([$enable])
```

If `$enable` is true (or missing), then `encode` will require either enabled `type_all_string` or second argument with supplied JSON types. See `Cpanel::JSON::XS::Type`. When `type_all_string` is not enabled or second argument is not provided (or is undef), then `encode` croaks. It also croaks when the type for provided structure in `encode` is incomplete.

```
$json = $json->type_all_string([$enable])
$enable = $json->get_type_all_string
    $json = $json->type_all_string([$enable])
```

If `$enable` is true (or missing), then `encode` will always produce stable deterministic JSON string types in resulted output.

When `$enable` is false, then result of encoded JSON output may be different for different Perl versions and may depends on loaded modules.

This is useful if you need deterministic JSON types, independently of used Perl version and other modules, but do not want to write complicated type definitions for `Cpanel::JSON::XS::Type`.

```
$json = $json->allow_dupkeys([$enable])
$enabled = $json->get_allow_dupkeys
```

If `$enable` is true (or missing), then the `decode` method will not die when it encounters duplicate keys in a hash. `allow_dupkeys` is also enabled in the `relaxed` mode.

The JSON spec allows duplicate name in objects but recommends to disable it, however with Perl hashes they are impossible, parsing JSON in Perl silently ignores duplicate names, using the last value found.

See <http://seriot.ch/parsing_json.php#24>: RFC 7159 section 4: “The names within an object should be unique.”

```
$json = $json->allow_blessed([$enable])
$enabled = $json->get_allow_blessed
```

If `$enable` is true (or missing), then the `encode` method will not barf when it encounters a blessed reference. Instead, the value of the **convert_blessed** option will decide whether `null` (`convert_blessed` disabled or no `TO_JSON` method found) or a representation of the object (`convert_blessed` enabled and `TO_JSON` method found) is being encoded. Has no effect on `decode`.

If `$enable` is false (the default), then `encode` will throw an exception when it encounters a blessed object.

This setting has no effect on `decode`.

```
$json = $json->convert_blessed([$enable])
$enabled = $json->get_convert_blessed
```

If `$enable` is true (or missing), then `encode`, upon encountering a blessed object, will check for the availability of the `TO_JSON` method on the object’s class. If found, it will be called in scalar context and the resulting scalar will be encoded instead of the object. If no `TO_JSON` method is found, a stringification overload method is tried next. If both are not found, the value of `allow_blessed` will decide what to do.

The `TO_JSON` method may safely call `die` if it wants. If `TO_JSON` returns other blessed objects, those will be handled in the same way. `TO_JSON` must take care of not causing an endless recursion cycle (== crash) in this case. The same care must be taken with calling `encode` in `stringify` overloads (even if this works by luck in older perls) or other callbacks. The name of `TO_JSON` was chosen because other methods called by the Perl core (== not by the user of the object) are usually in upper case letters and to avoid collisions with any `to_json` function or method.

If `$enable` is false (the default), then `encode` will not consider this type of conversion.

This setting has no effect on `decode`.

```
$json = $json->allow_tags([$enable])
$enabled = $json->get_allow_tags
```

See “OBJECT SERIALIZATION” for details.

If `$enable` is true (or missing), then `encode`, upon encountering a blessed object, will check for the

availability of the `FREEZE` method on the object's class. If found, it will be used to serialize the object into a nonstandard tagged JSON value (that JSON decoders cannot decode).

It also causes `decode` to parse such tagged JSON values and deserialize them via a call to the `THAW` method.

If `$enable` is false (the default), then `encode` will not consider this type of conversion, and tagged JSON values will cause a parse error in `decode`, as if tags were not part of the grammar.

```
$json = $json->filter_json_object ([ $coderef->($hashref)])
```

When `$coderef` is specified, it will be called from `decode` each time it decodes a JSON object. The only argument is a reference to the newly-created hash. If the code references returns a single scalar (which need not be a reference), this value (i.e. a copy of that scalar to avoid aliasing) is inserted into the deserialized data structure. If it returns an empty list (NOTE: *not* `undef`, which is a valid scalar), the original deserialized hash will be inserted. This setting can slow down decoding considerably.

When `$coderef` is omitted or undefined, any existing callback will be removed and `decode` will not change the deserialized hash in any way.

Example, convert all JSON objects into the integer 5:

```
my $js = Cpanel::JSON::XS->new->filter_json_object (sub { 5 });
# returns [5]
$js->decode ('[{}]')
# throw an exception because allow_nonref is not enabled
# so a lone 5 is not allowed.
$js->decode ('{"a":1, "b":2}');
```

```
$json = $json->filter_json_single_key_object ($key [=> $coderef->($value)])
```

Works remotely similar to `filter_json_object`, but is only called for JSON objects having a single key named `$key`.

This `$coderef` is called before the one specified via `filter_json_object`, if any. It gets passed the single value in the JSON object. If it returns a single value, it will be inserted into the data structure. If it returns nothing (not even `undef` but the empty list), the callback from `filter_json_object` will be called next, as if no single-key callback were specified.

If `$coderef` is omitted or undefined, the corresponding callback will be disabled. There can only ever be one callback for a given key.

As this callback gets called less often than the `filter_json_object` one, decoding speed will not usually suffer as much. Therefore, single-key objects make excellent targets to serialize Perl objects into, especially as single-key JSON objects are as close to the type-tagged value concept as JSON gets (it's basically an ID/VALUE tuple). Of course, JSON does not support this in any way, so you need to make sure your data never looks like a serialized Perl hash.

Typical names for the single object key are `__class_whatever__`, or `$__dollars_are_rarely_used__$` or `}ugly_brace_placement`, or even things like `__class_md5sum(classname)__`, to reduce the risk of clashing with real hashes.

Example, decode JSON objects of the form `{ "__widget__" => <id> }` into the corresponding `$WIDGET{<id>}` object:

```
# return whatever is in $WIDGET{5}:
Cpanel::JSON::XS
->new
->filter_json_single_key_object (__widget__ => sub {
    $WIDGET{ $_[0] }
})
->decode ('{"__widget__": 5}')
```

```

# this can be used with a TO_JSON method in some "widget" class
# for serialization to json:
sub WidgetBase::TO_JSON {
    my ($self) = @_;

    unless ($self->{id}) {
        $self->{id} = ..get..some..id..;
        $WIDGET{$self->{id}} = $self;
    }

    { __widget__ => $self->{id} }
}

```

```
$json = $json->shrink ([ $enable ])
```

```
$enable = $json->get_shrink
```

Perl usually over-allocates memory a bit when allocating space for strings. This flag optionally resizes strings generated by either `encode` or `decode` to their minimum size possible. This can save memory when your JSON texts are either very very long or you have many short strings. It will also try to downgrade any strings to octet-form if possible: perl stores strings internally either in an encoding called UTF-X or in octet-form. The latter cannot store everything but uses less space in general (and some buggy Perl or C code might even rely on that internal representation being used).

The actual definition of what `shrink` does might change in future versions, but it will always try to save space at the expense of time.

If `$enable` is true (or missing), the string returned by `encode` will be shrunk-to-fit, while all strings generated by `decode` will also be shrunk-to-fit.

If `$enable` is false, then the normal perl allocation algorithms are used. If you work with your data, then this is likely to be faster.

In the future, this setting might control other things, such as converting strings that look like integers or floats into integers or floats internally (there is no difference on the Perl level), saving space.

```
$json = $json->max_depth ([ $maximum_nesting_depth ])
```

```
$max_depth = $json->get_max_depth
```

Sets the maximum nesting level (default 512) accepted while encoding or decoding. If a higher nesting level is detected in JSON text or a Perl data structure, then the encoder and decoder will stop and croak at that point.

Nesting level is defined by number of hash- or arrayrefs that the encoder needs to traverse to reach a given point or the number of `{` or `[` characters without their matching closing parenthesis crossed to reach a given character in a string.

Setting the maximum depth to one disallows any nesting, so that ensures that the object is only a single hash/object or array.

If no argument is given, the highest possible setting will be used, which is rarely useful.

Note that nesting is implemented by recursion in C. The default value has been chosen to be as large as typical operating systems allow without crashing.

See SECURITY CONSIDERATIONS, below, for more info on why this is useful.

```
$json = $json->max_size ([ $maximum_string_size ])
```

```
$max_size = $json->get_max_size
```

Set the maximum length a JSON text may have (in bytes) where decoding is being attempted. The default is 0, meaning no limit. When `decode` is called on a string that is longer than this many bytes, it will not attempt to decode the string but throw an exception. This setting has no effect on `encode` (yet).

If no argument is given, the limit check will be deactivated (same as when 0 is specified).

See “SECURITY CONSIDERATIONS”, below, for more info on why this is useful.

```
$json->stringify_infnan ([$infnan_mode = 1])
```

```
$infnan_mode = $json->get_stringify_infnan
```

Get or set how Cpanel::JSON::XS encodes `inf`, `-inf` or `nan` for numeric values. Also `qnan`, `snan` or negative `nan` on some platforms.

`null`: `infnan_mode = 0`. Similar to most JSON modules in other languages. Always `null`.

`stringified`: `infnan_mode = 1`. As in `Mojo::JSON`. Platform specific strings. Stringified via `sprintf(%g)`, with double quotes.

`inf/nan`: `infnan_mode = 2`. As in `JSON::XS`, and older releases. Passes through platform dependent values, invalid JSON. Stringified via `sprintf(%g)`, but without double quotes.

`“inf/-inf/nan”`: `infnan_mode = 3`. Platform independent `inf/nan/-inf` strings. No `QNaN/SNaN/negative NAN` support, unified to `“nan”`. Much easier to detect, but may conflict with valid strings.

```
$json_text = $json->encode ($perl_scalar, $json_type)
```

Converts the given Perl data structure (a simple scalar or a reference to a hash or array) to its JSON representation. Simple scalars will be converted into JSON string or number sequences, while references to arrays become JSON arrays and references to hashes become JSON objects. Undefined Perl values (e.g. `undef`) become JSON `null` values. Neither `true` nor `false` values will be generated.

For the type argument see `Cpanel::JSON::XS::Type`.

```
$perl_scalar = $json->decode ($json_text, my $json_type)
```

The opposite of `encode`: expects a JSON text and tries to parse it, returning the resulting simple scalar or reference. Croaks on error.

JSON numbers and strings become simple Perl scalars. JSON arrays become Perl arrayrefs and JSON objects become Perl hashrefs. `true` becomes `1`, `false` becomes `0` and `null` becomes `undef`.

For the type argument see `Cpanel::JSON::XS::Type`.

```
($perl_scalar, $characters) = $json->decode_prefix ($json_text)
```

This works like the `decode` method, but instead of raising an exception when there is trailing garbage after the first JSON object, it will silently stop parsing there and return the number of characters consumed so far.

This is useful if your JSON texts are not delimited by an outer protocol and you need to know where the JSON text ends.

```
Cpanel::JSON::XS->new->decode_prefix (" [1] the tail")
=> ([1], 3)
```

```
$json->to_json ($perl_hash_or_arrayref)
```

Deprecated method for perl 5.8 and newer. Use `encode_json` instead.

```
$json->from_json ($utf8_encoded_json_text)
```

Deprecated method for perl 5.8 and newer. Use `decode_json` instead.

INCREMENTAL PARSING

In some cases, there is the need for incremental parsing of JSON texts. While this module always has to keep both JSON text and resulting Perl data structure in memory at one time, it does allow you to parse a JSON stream incrementally. It does so by accumulating text until it has a full JSON object, which it then can decode. This process is similar to using `decode_prefix` to see if a full JSON object is available, but is much more efficient (and can be implemented with a minimum of method calls).

`Cpanel::JSON::XS` will only attempt to parse the JSON text once it is sure it has enough text to get a

decisive result, using a very simple but truly incremental parser. This means that it sometimes won't stop as early as the full parser, for example, it doesn't detect mismatched parentheses. The only thing it guarantees is that it starts decoding as soon as a syntactically valid JSON text has been seen. This means you need to set resource limits (e.g. `max_size`) to ensure the parser will stop parsing in the presence of syntax errors.

The following methods implement this incremental parser.

`[void, scalar or list context] = $json->incr_parse ([$string])`

This is the central parsing function. It can both append new text and extract objects from the stream accumulated so far (both of these functions are optional).

If `$string` is given, then this string is appended to the already existing JSON fragment stored in the `$json` object.

After that, if the function is called in void context, it will simply return without doing anything further. This can be used to add more text in as many chunks as you want.

If the method is called in scalar context, then it will try to extract exactly *one* JSON object. If that is successful, it will return this object, otherwise it will return `undef`. If there is a parse error, this method will croak just as `decode` would do (one can then use `incr_skip` to skip the erroneous part). This is the most common way of using the method.

And finally, in list context, it will try to extract as many objects from the stream as it can find and return them, or the empty list otherwise. For this to work, there must be no separators between the JSON objects or arrays, instead they must be concatenated back-to-back. If an error occurs, an exception will be raised as in the scalar context case. Note that in this case, any previously-parsed JSON texts will be lost.

Example: Parse some JSON arrays/objects in a given string and return them.

```
my @objs = Cpanel::JSON::XS->new->incr_parse (" [5] [7] [1,2] ");
```

`$lvalue_string = $json->incr_text (>5.8 only)`

This method returns the currently stored JSON fragment as an lvalue, that is, you can manipulate it. This *only* works when a preceding call to `incr_parse` in *scalar context* successfully returned an object, and 2. only with Perl `>= 5.8`

Under all other circumstances you must not call this function (I mean it. although in simple tests it might actually work, it *will* fail under real world conditions). As a special exception, you can also call this method before having parsed anything.

This function is useful in two cases: a) finding the trailing text after a JSON object or b) parsing multiple JSON objects separated by non-JSON text (such as commas).

`$json->incr_skip`

This will reset the state of the incremental parser and will remove the parsed text from the input buffer so far. This is useful after `incr_parse` died, in which case the input buffer and incremental parser state is left unchanged, to skip the text parsed so far and to reset the parse state.

The difference to `incr_reset` is that only text until the parse error occurred is removed.

`$json->incr_reset`

This completely resets the incremental parser, that is, after this call, it will be as if the parser had never parsed anything.

This is useful if you want to repeatedly parse JSON objects and want to ignore any trailing data, which means you have to reset the parser after each successful decode.

LIMITATIONS

All options that affect decoding are supported, except `allow_nonref`. The reason for this is that it cannot be made to work sensibly: JSON objects and arrays are self-delimited, i.e. you can concatenate them back to back and still decode them perfectly. This does not hold true for JSON numbers, however.

For example, is the string `1` a single JSON number, or is it simply the start of `12`? Or is `12` a single JSON

number, or the concatenation of 1 and 2? In neither case you can tell, and this is why `Cpanel::JSON::XS` takes the conservative route and disallows this case.

EXAMPLES

Some examples will make all this clearer. First, a simple example that works similarly to `decode_prefix`: We want to decode the JSON object at the start of a string and identify the portion after the JSON object:

```
my $text = "[1,2,3] hello";

my $json = new Cpanel::JSON::XS;

my $obj = $json->incr_parse ($text)
    or die "expected JSON object or array at beginning of string";

my $tail = $json->incr_text;
# $tail now contains " hello"
```

Easy, isn't it?

Now for a more complicated example: Imagine a hypothetical protocol where you read some requests from a TCP stream, and each request is a JSON array, without any separation between them (in fact, it is often useful to use newlines as "separators", as these get interpreted as whitespace at the start of the JSON text, which makes it possible to test said protocol with `telnet...`).

Here is how you'd do it (it is trivial to write this in an event-based manner):

```
my $json = new Cpanel::JSON::XS;

# read some data from the socket
while (sysread $socket, my $buf, 4096) {

    # split and decode as many requests as possible
    for my $request ($json->incr_parse ($buf)) {
        # act on the $request
    }
}
```

Another complicated example: Assume you have a string with JSON objects or arrays, all separated by (optional) comma characters (e.g. `[1], [2], [3]`). To parse them, we have to skip the commas between the JSON texts, and here is where the lvalue-ness of `incr_text` comes in useful:

```
my $text = "[1],[2],[3]";
my $json = new Cpanel::JSON::XS;

# void context, so no parsing done
$json->incr_parse ($text);

# now extract as many objects as possible. note the
# use of scalar context so incr_text can be called.
while (my $obj = $json->incr_parse) {
    # do something with $obj

    # now skip the optional comma
    $json->incr_text =~ s/^\s* , //x;
}
```

Now lets go for a very complex example: Assume that you have a gigantic JSON array-of-objects, many gigabytes in size, and you want to parse it, but you cannot load it into memory fully (this has actually happened in the real world :).

Well, you lost, you have to implement your own JSON parser. But `Cpanel::JSON::XS` can still help you: You implement a (very simple) array parser and let JSON decode the array elements, which are all full JSON objects on their own (this wouldn't work if the array elements could be JSON numbers, for example):

```

my $jjson = new Cpanel::JSON::XS;

# open the monster
open my $fh, "<bigfile.json"
  or die "bigfile: $!";

# first parse the initial "["
for (;;) {
  sysread $fh, my $buf, 65536
    or die "read error: $!";
  $jjson->incr_parse ($buf); # void context, so no parsing

  # Exit the loop once we found and removed(!) the initial "[".
  # In essence, we are (ab-)using the $jjson object as a simple scalar
  # we append data to.
  last if $jjson->incr_text =~ s/^ \s* \[ //x;
}

# now we have the skipped the initial "[", so continue
# parsing all the elements.
for (;;) {
  # in this loop we read data until we got a single JSON object
  for (;;) {
    if (my $obj = $jjson->incr_parse) {
      # do something with $obj
      last;
    }

    # add more data
    sysread $fh, my $buf, 65536
      or die "read error: $!";
    $jjson->incr_parse ($buf); # void context, so no parsing
  }

  # in this loop we read data until we either found and parsed the
  # separating ",", between elements, or the final "]"
  for (;;) {
    # first skip whitespace
    $jjson->incr_text =~ s/^ \s* //;

    # if we find "]", we are done
    if ($jjson->incr_text =~ s/^ \] //) {
      print "finished.\n";
      exit;
    }

    # if we find ",", we can continue with the next element
    if ($jjson->incr_text =~ s/^ , //) {
      last;
    }
  }
}

```

```

# if we find anything else, we have a parse error!
if (length $json->incr_text) {
    die "parse error near ", $json->incr_text;
}

# else add more data
sysread $fh, my $buf, 65536
    or die "read error: $!";
$json->incr_parse ($buf); # void context, so no parsing
}

```

This is a complex example, but most of the complexity comes from the fact that we are trying to be correct (bear with me if I am wrong, I never ran the above example :).

BOM

Detect all unicode **Byte Order Marks** on decode. Which are UTF-8, UTF-16LE, UTF-16BE, UTF-32LE and UTF-32BE.

The BOM encoding is set only for one specific decode call, it does not change the state of the JSON object.

Warning: With perls older than 5.20 you need load the Encode module before loading a multibyte BOM, i.e. `>= UTF-16`. Otherwise an error is thrown. This is an implementation limitation and might get fixed later.

See <https://tools.ietf.org/html/rfc7159#section-8.1> “JSON text SHALL be encoded in UTF-8, UTF-16, or UTF-32.”

“Implementations MUST NOT add a byte order mark to the beginning of a JSON text”, “implementations (...) MAY ignore the presence of a byte order mark rather than treating it as an error”.

See also http://www.unicode.org/faq/utf_bom.html#BOM.

Beware that `Cpanel::JSON::XS` is currently the only JSON module which does accept and decode a BOM.

The latest JSON spec <https://www.greenbytes.de/tech/webdav/rfc8259.html#character.encoding> forbid the usage of UTF-16 or UTF-32, the character encoding is UTF-8. Thus in subsequent updates BOM's of UTF-16 or UTF-32 will throw an error.

MAPPING

This section describes how `Cpanel::JSON::XS` maps Perl values to JSON values and vice versa. These mappings are designed to “do the right thing” in most circumstances automatically, preserving round-tripping characteristics (what you put in comes out as something equivalent).

For the more enlightened: note that in the following descriptions, lowercase *perl* refers to the Perl interpreter, while uppercase *Perl* refers to the abstract Perl language itself.

JSON → PERL

object

A JSON object becomes a reference to a hash in Perl. No ordering of object keys is preserved (JSON does not preserve object key ordering itself).

array

A JSON array becomes a reference to an array in Perl.

string

A JSON string becomes a string scalar in Perl – Unicode codepoints in JSON are represented by the same codepoints in the Perl string, so no manual decoding is necessary.

number

A JSON number becomes either an integer, numeric (floating point) or string scalar in perl, depending on its range and any fractional parts. On the Perl level, there is no difference between those as Perl handles all the conversion details, but an integer may take slightly less memory and might represent more values exactly than floating point numbers.

If the number consists of digits only, `Cpanel::JSON::XS` will try to represent it as an integer value. If

that fails, it will try to represent it as a numeric (floating point) value if that is possible without loss of precision. Otherwise it will preserve the number as a string value (in which case you lose roundtripping ability, as the JSON number will be re-encoded to a JSON string).

Numbers containing a fractional or exponential part will always be represented as numeric (floating point) values, possibly at a loss of precision (in which case you might lose perfect roundtripping ability, but the JSON number will still be re-encoded as a JSON number).

Note that precision is not accuracy – binary floating point values cannot represent most decimal fractions exactly, and when converting from and to floating point, `Cpanel::JSON::XS` only guarantees precision up to but not including the least significant bit.

true, false

When `unblessed_bool` is set to true, then JSON `true` becomes 1 and JSON `false` becomes 0.

Otherwise these JSON atoms become `Cpanel::JSON::XS::true` and `Cpanel::JSON::XS::false`, respectively. They are `JSON::PP::Boolean` objects and are overloaded to act almost exactly like the numbers 1 and 0. You can check whether a scalar is a JSON boolean by using the `Cpanel::JSON::XS::is_bool` function.

The other round, from perl to JSON, `!0` which is represented as `yes` becomes `true`, and `!1` which is represented as `no` becomes `false`.

Via `Cpanel::JSON::XS::Type` you can now even force negation in `encode`, without overloading of `!`:

```
my $false = Cpanel::JSON::XS::false;
print ($json->encode([!$false], [JSON_TYPE_BOOL]));
=> [true]
```

null

A JSON null atom becomes `undef` in Perl.

shell-style comments (`# text`)

As a nonstandard extension to the JSON syntax that is enabled by the `relaxed` setting, shell-style comments are allowed. They can start anywhere outside strings and go till the end of the line.

tagged values (`(tag) value`).

Another nonstandard extension to the JSON syntax, enabled with the `allow_tags` setting, are tagged values. In this implementation, the `tag` must be a perl package/class name encoded as a JSON string, and the `value` must be a JSON array encoding optional constructor arguments.

See “OBJECT SERIALIZATION”, below, for details.

PERL -> JSON

The mapping from Perl to JSON is slightly more difficult, as Perl is a truly typeless language, so we can only guess which JSON type is meant by a Perl value.

hash references

Perl hash references become JSON objects. As there is no inherent ordering in hash keys (or JSON objects), they will usually be encoded in a pseudo-random order that can change between runs of the same program but stays generally the same within a single run of a program. `Cpanel::JSON::XS` can optionally sort the hash keys (determined by the `canonical` flag), so the same datastructure will serialize to the same JSON text (given same settings and version of `Cpanel::JSON::XS`), but this incurs a runtime overhead and is only rarely useful, e.g. when you want to compare some JSON text against another for equality.

array references

Perl array references become JSON arrays.

other references

Other unblessed references are generally not allowed and will cause an exception to be thrown, except for references to the integers 0 and 1, which get turned into `false` and `true` atoms in JSON.

With the option `allow_stringify`, you can ignore the exception and return the stringification of the perl value.

With the option `allow_unknown`, you can ignore the exception and return null instead.

```

encode_json ["x"]          # => cannot encode reference to scalar 'SCALAR(0x...)
                           # unless the scalar is 0 or 1
encode_json [\0, \1]      # yields [false,true]

allow_stringify->encode_json ["x"] # yields "x" unlike JSON::PP
allow_unknown->encode_json ["x"]   # yields null as in JSON::PP

```

`Cpanel::JSON::XS::true`, `Cpanel::JSON::XS::false`

These special values become JSON true and JSON false values, respectively. You can also use `\1` and `\0` or `!0` and `!1` directly if you want.

```

encode_json [Cpanel::JSON::XS::false, Cpanel::JSON::XS::true] # yields [false,
encode_json [!1, !0], [JSON_TYPE_BOOL, JSON_TYPE_BOOL] # yields [false,true]

```

eq/ne comparisons with true, false:

false is eq to the empty string or the string 'false' or the special empty string `!0` or `!1`, i.e. `SV_NO`, or the numbers 0 or 0.0.

true is eq to the string 'true' or to the special string `!0` (i.e. `SV_YES`) or to the numbers 1 or 1.0.

blessed objects

Blessed objects are not directly representable in JSON, but `Cpanel::JSON::XS` allows various optional ways of handling objects. See "OBJECT SERIALIZATION", below, for details.

See the `allow_blessed` and `convert_blessed` methods on various options on how to deal with this: basically, you can choose between throwing an exception, encoding the reference as if it weren't blessed, use the objects overloaded stringification method or provide your own serializer method.

simple scalars

Simple Perl scalars (any scalar that is not a reference) are the most difficult objects to encode: `Cpanel::JSON::XS` will encode undefined scalars or `inf/nan` as JSON null values and other scalars to either number or string in non-deterministic way which may be affected or changed by Perl version or any other loaded Perl module.

If you want to have stable and deterministic types in JSON encoder then use `Cpanel::JSON::XS::Type`.

Alternative way for deterministic types is to use `type_all_string` method when all perl scalars are encoded to JSON strings.

Non-deterministic behavior is following: scalars that have last been used in a string context before encoding as JSON strings, and anything else as number value:

```

# dump as number
encode_json [2]          # yields [2]
encode_json [-3.0e17]   # yields [-3e+17]
my $value = 5; encode_json [$value] # yields [5]

# used as string, but the two representations are for the same number
print $value;
encode_json [$value]    # yields [5]

# used as different string (non-matching dual-var)
my $str = '0 but true';
my $num = 1 + $str;
encode_json [$num, $str] # yields [1,"0 but true"]

```

```
# undef becomes null
encode_json [undef]           # yields [null]

# inf or nan becomes null, unless you answered
# "Do you want to handle inf/nan as strings" with yes
encode_json [9**9**9]         # yields [null]
```

You can force the type to be a JSON string by stringifying it:

```
my $x = 3.1; # some variable containing a number
"$x";       # stringified
$x .= "";   # another, more awkward way to stringify
print $x;   # perl does it for you, too, quite often
```

You can force the type to be a JSON number by numifying it:

```
my $x = "3"; # some variable containing a string
$x += 0;     # numify it, ensuring it will be dumped as a number
$x *= 1;     # same thing, the choice is yours.
```

Note that numerical precision has the same meaning as under Perl (so binary to decimal conversion follows the same rules as in Perl, which can differ to other languages). Also, your perl interpreter might expose extensions to the floating point numbers of your platform, such as infinities or NaN's – these cannot be represented in JSON, and thus null is returned instead. Optionally you can configure it to stringify inf and nan values.

OBJECT SERIALIZATION

As JSON cannot directly represent Perl objects, you have to choose between a pure JSON representation (without the ability to deserialize the object automatically again), and a nonstandard extension to the JSON syntax, tagged values.

SERIALIZATION

What happens when `Cpanel::JSON::XS` encounters a Perl object depends on the `allow_blessed`, `convert_blessed` and `allow_tags` settings, which are used in this order:

1. `allow_tags` is enabled and the object has a `FREEZE` method.

In this case, `Cpanel::JSON::XS` uses the `Types::Serialiser` object serialization protocol to create a tagged JSON value, using a nonstandard extension to the JSON syntax.

This works by invoking the `FREEZE` method on the object, with the first argument being the object to serialize, and the second argument being the constant string `JSON` to distinguish it from other serializers.

The `FREEZE` method can return any number of values (i.e. zero or more). These values and the package/classname of the object will then be encoded as a tagged JSON value in the following format:

```
("classname") [FREEZE return values...]
```

e.g.:

```
("URI") ["http://www.google.com/"]
("MyDate") [2013,10,29]
("ImageData::JPEG") ["Z3...VlCg=="]
```

For example, the hypothetical `My::Object` `FREEZE` method might use the objects `type` and `id` members to encode the object:

```
sub My::Object::FREEZE {
    my ($self, $serializer) = @_;

    ($self->{type}, $self->{id})
```

```
}
```

2. `convert_blessed` is enabled and the object has a `TO_JSON` method.

In this case, the `TO_JSON` method of the object is invoked in scalar context. It must return a single scalar that can be directly encoded into JSON. This scalar replaces the object in the JSON text.

For example, the following `TO_JSON` method will convert all URI objects to JSON strings when serialized. The fact that these values originally were URI objects is lost.

```
sub URI::TO_JSON {
    my ($uri) = @_;
    $uri->as_string
}
```

3. `convert_blessed` is enabled and the object has a stringification overload.

In this case, the overloaded `"` method of the object is invoked in scalar context. It must return a single scalar that can be directly encoded into JSON. This scalar replaces the object in the JSON text.

For example, the following `"` method will convert all URI objects to JSON strings when serialized. The fact that these values originally were URI objects is lost.

```
package URI;
use overload "" => sub { shift->as_string };
```

4. `allow_blessed` is enabled.

The object will be serialized as a JSON null value.

5. none of the above

If none of the settings are enabled or the respective methods are missing, `Cpanel::JSON::XS` throws an exception.

DESERIALIZATION

For deserialization there are only two cases to consider: either nonstandard tagging was used, in which case `allow_tags` decides, or objects cannot be automatically be deserialized, in which case you can use postprocessing or the `filter_json_object` or `filter_json_single_key_object` callbacks to get some real objects out of your JSON.

This section only considers the tagged value case: If a tagged JSON object is encountered during decoding and `allow_tags` is disabled, a parse error will result (as if tagged values were not part of the grammar).

If `allow_tags` is enabled, `Cpanel::JSON::XS` will look up the `THAW` method of the package/classname used during serialization (it will not attempt to load the package as a Perl module). If there is no such method, the decoding will fail with an error.

Otherwise, the `THAW` method is invoked with the classname as first argument, the constant string `JSON` as second argument, and all the values from the JSON array (the values originally returned by the `FREEZE` method) as remaining arguments.

The method must then return the object. While technically you can return any Perl scalar, you might have to enable the `enable_nonref` setting to make that work in all cases, so better return an actual blessed reference.

As an example, let's implement a `THAW` function that regenerates the `My::Object` from the `FREEZE` example earlier:

```
sub My::Object::THAW {
    my ($class, $serializer, $type, $id) = @_;

    $class->new (type => $type, id => $id)
}
```

See the "SECURITY CONSIDERATIONS" section below. Allowing external json objects being deserialized to perl objects is usually a very bad idea.

ENCODING/CODESET FLAG NOTES

The interested reader might have seen a number of flags that signify encodings or codesets – `utf8`, `latin1`, `binary` and `ascii`. There seems to be some confusion on what these do, so here is a short comparison:

`utf8` controls whether the JSON text created by `encode` (and expected by `decode`) is UTF-8 encoded or not, while `latin1` and `ascii` only control whether `encode` escapes character values outside their respective codeset range. Neither of these flags conflict with each other, although some combinations make less sense than others.

Care has been taken to make all flags symmetrical with respect to `encode` and `decode`, that is, texts encoded with any combination of these flag values will be correctly decoded when the same flags are used – in general, if you use different flag settings while encoding vs. when decoding you likely have a bug somewhere.

Below comes a verbose discussion of these flags. Note that a “codeset” is simply an abstract set of character-codepoint pairs, while an encoding takes those codepoint numbers and *encodes* them, in our case into octets. Unicode is (among other things) a codeset, UTF-8 is an encoding, and ISO-8859-1 (= latin 1) and ASCII are both codesets *and* encodings at the same time, which can be confusing.

`utf8` flag disabled

When `utf8` is disabled (the default), then `encode/decode` generate and expect Unicode strings, that is, characters with high ordinal Unicode values (> 255) will be encoded as such characters, and likewise such characters are decoded as-is, no changes to them will be done, except “(re-)interpreting” them as Unicode codepoints or Unicode characters, respectively (to Perl, these are the same thing in strings unless you do funny/weird/dumb stuff).

This is useful when you want to do the encoding yourself (e.g. when you want to have UTF-16 encoded JSON texts) or when some other layer does the encoding for you (for example, when printing to a terminal using a filehandle that transparently encodes to UTF-8 you certainly do NOT want to UTF-8 encode your data first and have Perl encode it another time).

`utf8` flag enabled

If the `utf8`-flag is enabled, `encode/decode` will encode all characters using the corresponding UTF-8 multi-byte sequence, and will expect your input strings to be encoded as UTF-8, that is, no “character” of the input string must have any value > 255, as UTF-8 does not allow that.

The `utf8` flag therefore switches between two modes: disabled means you will get a Unicode string in Perl, enabled means you get an UTF-8 encoded octet/binary string in Perl.

`latin1`, `binary` or `ascii` flags enabled

With `latin1` (or `ascii`) enabled, `encode` will escape characters with ordinal values > 255 (> 127 with `ascii`) and encode the remaining characters as specified by the `utf8` flag. With `binary` enabled, ordinal values > 255 are illegal.

If `utf8` is disabled, then the result is also correctly encoded in those character sets (as both are proper subsets of Unicode, meaning that a Unicode string with all character values < 256 is the same thing as a ISO-8859-1 string, and a Unicode string with all character values < 128 is the same thing as an ASCII string in Perl).

If `utf8` is enabled, you still get a correct UTF-8-encoded string, regardless of these flags, just some more characters will be escaped using `\uXXXX` then before.

Note that ISO-8859-1-*encoded* strings are not compatible with UTF-8 encoding, while ASCII-encoded strings are. That is because the ISO-8859-1 encoding is NOT a subset of UTF-8 (despite the ISO-8859-1 *codeset* being a subset of Unicode), while ASCII is.

Surprisingly, `decode` will ignore these flags and so treat all input values as governed by the `utf8` flag. If it is disabled, this allows you to decode ISO-8859-1- and ASCII-encoded strings, as both strict subsets of Unicode. If it is enabled, you can correctly decode UTF-8 encoded strings.

So neither `latin1`, `binary` nor `ascii` are incompatible with the `utf8` flag – they only govern

when the JSON output engine escapes a character or not.

The main use for `latin1` or `binary` is to relatively efficiently store binary data as JSON, at the expense of breaking compatibility with most JSON decoders.

The main use for `ascii` is to force the output to not contain characters with values > 127 , which means you can interpret the resulting string as UTF-8, ISO-8859-1, ASCII, KOI8-R or most about any character set and 8-bit-encoding, and still get the same data structure back. This is useful when your channel for JSON transfer is not 8-bit clean or the encoding might be mangled in between (e.g. in mail), and works because ASCII is a proper subset of most 8-bit and multibyte encodings in use in the world.

JSON and ECMAScript

JSON syntax is based on how literals are represented in javascript (the not-standardized predecessor of ECMAScript) which is presumably why it is called “JavaScript Object Notation”.

However, JSON is not a subset (and also not a superset of course) of ECMAScript (the standard) or javascript (whatever browsers actually implement).

If you want to use javascript’s `eval` function to “parse” JSON, you might run into parse errors for valid JSON texts, or the resulting data structure might not be queryable:

One of the problems is that U+2028 and U+2029 are valid characters inside JSON strings, but are not allowed in ECMAScript string literals, so the following Perl fragment will not output something that can be guaranteed to be parsable by javascript’s `eval`:

```
use Cpanel::JSON::XS;

print encode_json [chr 0x2028];
```

The right fix for this is to use a proper JSON parser in your javascript programs, and not rely on `eval` (see for example Douglas Crockford’s *json2.js* parser).

If this is not an option, you can, as a stop-gap measure, simply encode to ASCII-only JSON:

```
use Cpanel::JSON::XS;

print Cpanel::JSON::XS->new->ascii->encode ([chr 0x2028]);
```

Note that this will enlarge the resulting JSON text quite a bit if you have many non-ASCII characters. You might be tempted to run some regexes to only escape U+2028 and U+2029, e.g.:

```
# DO NOT USE THIS!
my $json = Cpanel::JSON::XS->new->utf8->encode ([chr 0x2028]);
$json =~ s/\xe2\x80\xa8/\u2028/g; # escape U+2028
$json =~ s/\xe2\x80\xa9/\u2029/g; # escape U+2029
print $json;
```

Note that *this is a bad idea*: the above only works for U+2028 and U+2029 and thus only for fully ECMAScript-compliant parsers. Many existing javascript implementations, however, have issues with other characters as well – using `eval` naively simply *will* cause problems.

Another problem is that some javascript implementations reserve some property names for their own purposes (which probably makes them non-ECMAScript-compliant). For example, Iceweasel reserves the `__proto__` property name for its own purposes.

If that is a problem, you could parse try to filter the resulting JSON output for these property strings, e.g.:

```
$json =~ s/"__proto__"\s*/"__proto__renamed"/g;
```

This works because `__proto__` is not valid outside of strings, so every occurrence of `"__proto__"\s*` must be a string used as property name.

Unicode non-characters between U+FFFD and U+10FFFF are decoded either to the recommended U+FFFD REPLACEMENT CHARACTER (see Unicode PR #121: Recommended Practice for Replacement

Characters), or in the binary or relaxed mode left as is, keeping the illegal non-characters as before.

Raw non-Unicode characters outside the valid unicode range fail now to parse, because “A string is a sequence of zero or more Unicode characters” RFC 7159 section 1 and "JSON text SHALL be encoded in Unicode RFC 7159 section 8.1. We use now the UTF8_DISALLOW_SUPER flag when parsing unicode.

If you know of other incompatibilities, please let me know.

JSON and YAML

You often hear that JSON is a subset of YAML. *in general, there is no way to configure JSON::XS to output a data structure as valid YAML that works in all cases.* If you really must use Cpanel::JSON::XS to generate YAML, you should use this algorithm (subject to change in future versions):

```
my $to_yaml = Cpanel::JSON::XS->new->utf8->space_after (1);
my $yaml = $to_yaml->encode ($ref) . "\n";
```

This will *usually* generate JSON texts that also parse as valid YAML.

SPEED

It seems that JSON::XS is surprisingly fast, as shown in the following tables. They have been generated with the help of the `eg/bench` program in the JSON::XS distribution, to make it easy to compare on your own system.

JSON::XS is with Data::MessagePack and Sereal one of the fastest serializers, because JSON and JSON::XS do not support backrefs (no graph structures), only trees. Storable supports backrefs, i.e. graphs. Data::MessagePack encodes its data binary (as Storable) and supports only very simple subset of JSON.

First comes a comparison between various modules using a very short single-line JSON string (also available at <http://dist.schmorp.de/misc/json/short.json>).

```
{"method": "handleMessage", "params": ["user1",
"we were just talking"], "id": null, "array": [1, 11, 234, -5, 1e5, 1e7,
1, 0]}
```

It shows the number of encodes/decodes per second (JSON::XS uses the functional interface, while Cpanel::JSON::XS/2 uses the OO interface with pretty-printing and hash key sorting enabled, Cpanel::JSON::XS/3 enables shrink. JSON::DWIW/DS uses the deserialize function, while JSON::DWIW:FJ uses the `from_json` method). Higher is better:

module	encode	decode
JSON::DWIW/DS	86302.551	102300.098
JSON::DWIW/FJ	86302.551	75983.768
JSON::PP	15827.562	6638.658
JSON::Syck	63358.066	47662.545
JSON::XS	511500.488	511500.488
JSON::XS/2	291271.111	388361.481
JSON::XS/3	361577.931	361577.931
Storable	66788.280	265462.278

That is, JSON::XS is almost six times faster than JSON::DWIW on encoding, about five times faster on decoding, and over thirty to seventy times faster than JSON's pure perl implementation. It also compares favourably to Storable for small amounts of data.

Using a longer test string (roughly 18KB, generated from Yahoo! Locals search API (<http://dist.schmorp.de/misc/json/long.json>)).

module	encode	decode
JSON::DWIW/DS	1647.927	2673.916
JSON::DWIW/FJ	1630.249	2596.128
JSON::PP	400.640	62.311
JSON::Syck	1481.040	1524.869
JSON::XS	20661.596	9541.183
JSON::XS/2	10683.403	9416.938
JSON::XS/3	20661.596	9400.054
Storable	19765.806	10000.725

Again, JSON::XS leads by far (except for Storable which non-surprisingly decodes a bit faster).

On large strings containing lots of high Unicode characters, some modules (such as JSON::PC) seem to decode faster than JSON::XS, but the result will be broken due to missing (or wrong) Unicode handling. Others refuse to decode or encode properly, so it was impossible to prepare a fair comparison table for that case.

For updated graphs see <<https://github.com/Sereal/Sereal/wiki/Sereal-Comparison-Graphs>>

INTEROP with JSON and JSON::XS and other JSON modules

As long as you only serialize data that can be directly expressed in JSON, Cpanel::JSON::XS is incapable of generating invalid JSON output (modulo bugs, but JSON::XS has found more bugs in the official JSON testsuite (1) than the official JSON testsuite has found in JSON::XS (0)). Cpanel::JSON::XS is currently the only known JSON decoder which passes all <http://seriot.ch/parsing_json.html> tests, while being the fastest also.

When you have trouble decoding JSON generated by this module using other decoders, then it is very likely that you have an encoding mismatch or the other decoder is broken.

When decoding, JSON::XS is strict by default and will likely catch all errors. There are currently two settings that change this: `relaxed` makes JSON::XS accept (but not generate) some non-standard extensions, and `allow_tags` or `allow_blessed` will allow you to encode and decode Perl objects, at the cost of being totally insecure and not outputting valid JSON anymore.

JSON-XS-3.01 broke interoperability with JSON-2.90 with booleans. See JSON.

Cpanel::JSON::XS needs to know the JSON and JSON::XS versions to be able work with those objects, especially when encoding a booleans like `{"is_true":true}`. So you need to load these modules before.

`true/false` overloading and boolean representations are supported.

JSON::XS and JSON::PP representations are accepted and older JSON::XS accepts Cpanel::JSON::XS booleans. All JSON modules JSON, JSON, PP, JSON::XS, Cpanel::JSON::XS produce JSON::PP::Boolean objects, just Mojo and JSON::YAJL not. Mojo produces Mojo::JSON::_Bool and JSON::YAJL::Parser just an unblessed IV.

Cpanel::JSON::XS accepts JSON::PP::Boolean and Mojo::JSON::_Bool objects as booleans.

I cannot think of any reason to still use JSON::XS anymore.

TAGGED VALUE SYNTAX AND STANDARD JSON EN/DECODERS

When you use `allow_tags` to use the extended (and also nonstandard and invalid) JSON syntax for serialized objects, and you still want to decode the generated serialize objects, you can run a regex to replace the tagged syntax by standard JSON arrays (it only works for “normal” package names without comma, newlines or single colons). First, the readable Perl version:

```
# if your FREEZE methods return no values, you need this replace first:
$json =~ s/\( \s* (" (? : [^\\":,]+|\\.|:)* ") \s* \) \s* \[ \s* \] /[$1]/gx;

# this works for non-empty constructor arg lists:
```

```
$json =~ s/\( \s* (" (? : [^\\"":,]+|\\".|:)* ") \s* \) \s* \[/[$1,/gx;
```

And here is a less readable version that is easy to adapt to other languages:

```
$json =~ s/\( \s* (" ([^\\"":,]+|\\".|:)*") \s* \) \s* \[/[$1,/g;
```

Here is an ECMAScript version (same regex):

```
json = json.replace (/ \( \s* (" ([^\\"":,]+|\\".|:)*") \s* \) \s* \[/g, "[ $1, "];
```

Since this syntax converts to standard JSON arrays, it might be hard to distinguish serialized objects from normal arrays. You can prepend a “magic number” as first array element to reduce chances of a collision:

```
$json =~ s/\( \s* (" ([^\\"":,]+|\\".|:)*") \s* \) \s* \[/["XU1peReLzT4ggE11LanBYq4G9Vz1i
```

And after decoding the JSON text, you could walk the data structure looking for arrays with a first element of XU1peReLzT4ggE11LanBYq4G9Vz1iwKF.

The same approach can be used to create the tagged format with another encoder. First, you create an array with the magic string as first member, the classname as second, and constructor arguments last, encode it as part of your JSON structure, and then:

```
$json =~ s/\[ \s* "XU1peReLzT4ggE11LanBYq4G9Vz1iwKF" \s* , \s* (" ([^\\"":,]+|\\".|:)*") \s* \
```

Again, this has some limitations – the magic string must not be encoded with character escapes, and the constructor arguments must be non-empty.

RFC7159

Since this module was written, Google has written a new JSON RFC, RFC 7159 (and RFC7158). Unfortunately, this RFC breaks compatibility with both the original JSON specification on www.json.org and RFC4627.

As far as I can see, you can get partial compatibility when parsing by using `->allow_nonref`. However, consider the security implications of doing so.

I haven’t decided yet when to break compatibility with RFC4627 by default (and potentially leave applications insecure) and change the default to follow RFC7159, but application authors are well advised to call `->allow_nonref(0)` even if this is the current default, if they cannot handle non-reference values, in preparation for the day when the default will change.

SECURITY CONSIDERATIONS

JSON::XS and Cpanel::JSON::XS are not only fast. JSON is generally the most secure serializing format, because it is the only one besides `Data::MessagePack`, which does not deserialize objects per default. For all languages, not just perl. The binary variant BSON (MongoDB) does more but is unsafe.

It is trivial for any attacker to create such serialized objects in JSON and trick perl into expanding them, thereby triggering certain methods. Watch <https://www.youtube.com/watch?v=Gzx6KlqiIZE> for an exploit demo for “CVE–2015–1592 SixApart MovableType Storable Perl Code Execution” for a deserializer which expands objects. Deserializing even coderefs (methods, functions) or external data would be considered the most dangerous.

Security relevant overview of serializers regarding deserializing objects by default:

	Objects	Coderefs	External Data
<code>Data::Dumper</code>	YES	YES	YES
<code>Storable</code>	YES	NO (def)	NO
<code>Sereal</code>	YES	NO	NO
YAML	YES	NO	NO
<code>B::C</code>	YES	YES	YES
<code>B::Bytecode</code>	YES	YES	YES
BSON	YES	YES	NO
<code>JSON::SL</code>	YES	NO	YES
JSON	NO (def)	NO	NO
<code>Data::MessagePack</code>	NO	NO	NO

XML	NO	NO	YES
Pickle	YES	YES	YES
PHP Deserialize	YES	NO	NO

When you are using JSON in a protocol, talking to untrusted potentially hostile creatures requires relatively few measures.

First of all, your JSON decoder should be secure, that is, should not have any buffer overflows. Obviously, this module should ensure that.

Second, you need to avoid resource-starving attacks. That means you should limit the size of JSON texts you accept, or make sure then when your resources run out, that's just fine (e.g. by using a separate process that can crash safely). The size of a JSON text in octets or characters is usually a good indication of the size of the resources required to decode it into a Perl structure. While `JSON::XS` can check the size of the JSON text, it might be too late when you already have it in memory, so you might want to check the size before you accept the string.

Third, `Cpanel::JSON::XS` recurses using the C stack when decoding objects and arrays. The C stack is a limited resource: for instance, on my amd64 machine with 8MB of stack size I can decode around 180k nested arrays but only 14k nested JSON objects (due to perl itself recursing deeply on croak to free the temporary). If that is exceeded, the program crashes. To be conservative, the default nesting limit is set to 512. If your process has a smaller stack, you should adjust this setting accordingly with the `max_depth` method.

Also keep in mind that `Cpanel::JSON::XS` might leak contents of your Perl data structures in its error messages, so when you serialize sensitive information you might want to make sure that exceptions thrown by `JSON::XS` will not end up in front of untrusted eyes.

If you are using `Cpanel::JSON::XS` to return packets to consumption by JavaScript scripts in a browser you should have a look at <http://blog.archive.jpsykes.com/47/practical-csrf-and-json-security/> to see whether you are vulnerable to some common attack vectors (which really are browser design bugs, but it is still you who will have to deal with it, as major browser developers care only for features, not about getting security right). You might also want to also look at `Mojo::JSON` special escape rules to prevent from XSS attacks.

“OLD” VS. “NEW” JSON (RFC 4627 VS. RFC 7159)

TL;DR: Due to security concerns, `Cpanel::JSON::XS` will not allow scalar data in JSON texts by default – you need to create your own `Cpanel::JSON::XS` object and enable `allow_nonref`:

```
my $json = JSON::XS->new->allow_nonref;

$text = $json->encode ($data);
$data = $json->decode ($text);
```

The long version: JSON being an important and supposedly stable format, the IETF standardized it as RFC 4627 in 2006. Unfortunately the inventor of JSON Douglas Crockford unilaterally changed the definition of JSON in javascript. Rather than create a fork, the IETF decided to standardize the new syntax (apparently, so I as told, without finding it very amusing).

The biggest difference between the original JSON and the new JSON is that the new JSON supports scalars (anything other than arrays and objects) at the top-level of a JSON text. While this is strictly backwards compatible to older versions, it breaks a number of protocols that relied on sending JSON back-to-back, and is a minor security concern.

For example, imagine you have two banks communicating, and on one side, the JSON coder gets upgraded. Two messages, such as 10 and 1000 might then be confused to mean 101000, something that couldn't happen in the original JSON, because neither of these messages would be valid JSON.

If one side accepts these messages, then an upgrade in the coder on either side could result in this becoming exploitable.

This module has always allowed these messages as an optional extension, by default disabled. The security concerns are the reason why the default is still disabled, but future versions might/will likely upgrade to the newer RFC as default format, so you are advised to check your implementation and/or override the default with `->allow_nonref (0)` to ensure that future versions are safe.

THREADS

Cpanel::JSON::XS has proper ithreads support, unlike JSON::XS. If you encounter any bugs with thread support please report them.

BUGS

While the goal of the Cpanel::JSON::XS module is to be correct, that unfortunately does not mean it's bug-free, only that the author thinks its design is bug-free. If you keep reporting bugs and tests they will be fixed swiftly, though.

Since the JSON::XS author refuses to use a public bugtracker and prefers private emails, we use the tracker at **github**, so you might want to report any issues twice. Once in private to MLEHMANN to be fixed in JSON::XS and one to our the public tracker. Issues fixed by JSON::XS with a new release will also be backported to Cpanel::JSON::XS and 5.6.2, as long as cPanel relies on 5.6.2 and Cpanel::JSON::XS as our serializer of choice.

<<https://github.com/rurban/Cpanel-JSON-XS/issues>>

LICENSE

This module is available under the same licences as perl, the Artistic license and the GPL.

SEE ALSO

The `cpanel_json_xs` command line utility for quick experiments.

JSON, JSON::XS, JSON::MaybeXS, Mojo::JSON, Mojo::JSON::MaybeXS, JSON::SL, JSON::DWIW, JSON::YAJL, JSON::Any, Test::JSON, Locale::Wolowitz, <<https://metacpan.org/search?q=JSON>>

<<https://tools.ietf.org/html/rfc7159>>

<<https://tools.ietf.org/html/rfc4627>>

AUTHOR

Reini Urban <rurban@cpan.org>

Marc Lehmann <schmorp@schmorp.de>, <http://home.schmorp.de/>

MAINTAINER

Reini Urban <rurban@cpan.org>