NAME

JSON::XS − JSON serialising/deserialising, done correctly and fast

JSON::XS − JSON /
(http://fleur.hio.jp/perldoc/mix/lib/JSON/XS.html)

SYNOPSIS

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

# Note that JSON version 2.0 and above will automatically use JSON::XS
# if available, at virtually no speed overhead either, so you should
# be able to just:

use JSON;

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

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.

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

FEATURES

- correct Unicode handling

  This module knows how to handle Unicode, documents how and when it does so, and even documents what ‘‘correct’’ means.

- round-trip integrity

  When you serialise a perl data structure using only data types supported by JSON and Perl, the deserialised 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 serialisers 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.

FUNCTIONAL INTERFACE

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

`$json_text` = encode_json `$perl_scalar`

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:

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

Except being faster.

`$perl_scalar` = decode_json `$json_text`

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

This function call is functionally identical to:

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

Except being faster.

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.
   Just ignore that flag unless you debug a Perl bug, a module written in XS or want to dive into the internals of perl. Otherwise it will only confuse you, as, despite the name, it says nothing about how your string is encoded. You can have Unicode strings with that flag set, with that flag clear, and you can have binary data with that flag set and that flag clear. Other possibilities exist, too.

   If you didn't know about that flag, just the better, pretend it doesn't exist.

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.
   It's a fact. Learn to live with it.

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 JSON::XS

Creates a new JSON::XS object that can be used to de/encode JSON strings. All boolean flags described below are by default *disabled* (with the exception of `allow_nonref`, which defaults to *enabled* since version `4.0`).

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

```
my $json = 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.

```
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.

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

`$json` = `$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 handed a 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 bytewise/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;
$jsontext = encode "UTF-16BE", JSON::XS->new->encode ($object);
```

Example, decode UTF−32LE−encoded JSON:

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

`$json` = `$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 = 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`−>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 any way. *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`).

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

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

Unlike other boolean options, this opotion is enabled by default beginning with version `4.0`. See "SECURITY CONSIDERATIONS" for the gory details.

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 without enabled `allow_nonref`, resulting in an error:

```
JSON::XS->new->allow_nonref (0)->encode ("Hello, World!")
=> hash- or arrayref expected...
```

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

See "OBJECT SERIALISATION" for details.

If `$enable` is true (or missing), then the `encode` method will not barf when it encounters a blessed reference that it cannot convert otherwise. Instead, a JSON `null` value is encoded instead of the object.

If `$enable` is false (the default), then `encode` will throw an exception when it encounters a blessed object that it cannot convert otherwise.

This setting has no effect on `decode`.

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

See "OBJECT SERIALISATION" for details.

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.

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 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 SERIALISATION" 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 serialise the object into a nonstandard tagged JSON value (that JSON decoders cannot decode).

It also causes `decode` to parse such tagged JSON values and deserialise 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.

`$json`->boolean_values ([$false, `$true`])
($false, `$true`) = `$json`->get_boolean_values
By default, JSON booleans will be decoded as overloaded `$Types::Serialiser::false` and `$Types::Serialiser::true` objects.

With this method you can specify your own boolean values for decoding - on decode, JSON `false` will be decoded as a copy of `$false`, and JSON `true` will be decoded as `$true` ("copy" here is the same thing as assigning a value to another variable, i.e. `$copy = $false`).

Calling this method without any arguments will reset the booleans to their default values.

`get_boolean_values` will return both `$false` and `$true` values, or the empty list when they are set to the default.

`$json` = `$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 reference returns a single scalar (which need not be a reference), this value (or rather a copy of it) is inserted into the deserialised data structure. If it returns an empty list (NOTE: *not* `undef`, which is a valid scalar), the original deserialised 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 deserialised hash in any way.

Example, convert all JSON objects into the integer 5:

```
my $js = 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 then the `filter_json_object` one, decoding speed will not usually suffer as much. Therefore, single-key objects make excellent targets to serialise 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 serialised 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}:
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 serialisation 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 ([$enable])
`$enabled` = `$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 then 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.

`$json_text` = `$json->`encode ($perl_scalar)
Converts the given Perl value or data structure to its JSON representation. Croaks on error.

`$perl_scalar` = `$json->`decode ($json_text)
The opposite of `encode`: expects a JSON text and tries to parse it, returning the resulting simple scalar or reference. Croaks on error.

($perl_scalar, `$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.

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

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).

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 if 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 (other than whitespace) 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 = JSON::XS->new->incr_parse ("[5][7][1,2]");
```

`$lvalue_string = $json->`incr_text

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. 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.

That means you can only use this function to look at or manipulate text before or after complete JSON objects, not while the parser is in the middle of parsing a JSON object.

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

The incremental parser is a non-exact parser: it works by gathering as much text as possible that *could* be a valid JSON text, followed by trying to decode it.

That means it sometimes needs to read more data than strictly necessary to diagnose an invalid JSON text. For example, after parsing the following fragment, the parser *could* stop with an error, as this fragment *cannot* be the beginning of a valid JSON text:

```
[,
```

In reality, hopwever, the parser might continue to read data until a length limit is exceeded or it finds a closing bracket.

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 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 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 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 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 $json = new 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: $!";
   $json->incr_parse ($buf); # void context, so no parsing

   # Exit the loop once we found and removed(!) the initial "[".
   # In essence, we are (ab-)using the $json object as a simple scalar
   # we append data to.
   last if $json->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 = $json->incr_parse) {
         # do something with $obj
         last;
      }

      # add more data
      sysread $fh, my $buf, 65536
         or die "read error: $!";
      $json->incr_parse ($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
      $json->incr_text =~ s/^\s*//;

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

      # if we find ",", we can continue with the next element
      if ($json->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 :).

MAPPING

This section describes how 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, 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, JSON::XS only guarantees precision up to but not including the least significant bit.

true, false
: These JSON atoms become `Types::Serialiser::true` and `Types::Serialiser::false`, respectively. They 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 `Types::Serialiser::is_bool` function (after `use Types::Serialier`, of course).

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 SERIALISATION", 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. JSON::XS can optionally sort the hash keys (determined by the *canonical* flag), so the same datastructure will serialise to the same JSON text (given same settings and version of 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.

Since `JSON::XS` uses the boolean model from Types::Serialiser, you can also `use Types::Serialiser` and then use `Types::Serialiser::false` and `Types::Serialiser::true` to improve readability.

```
use Types::Serialiser;
encode_json [\0, Types::Serialiser::true]      # yields [false,true]
```

Types::Serialiser::true, Types::Serialiser::false

These special values from the Types::Serialiser module become JSON true and JSON false values, respectively. You can also use `\1` and `\0` directly if you want.

blessed objects

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

simple scalars

Simple Perl scalars (any scalar that is not a reference) are the most difficult objects to encode: JSON::XS will encode undefined scalars as JSON `null` values, 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, so dump as string
print $value;
encode_json [$value]                 # yields ["5"]

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

You can not currently force the type in other, less obscure, ways. Tell me if you need this capability (but don't forget to explain why it's needed :).

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 it is an error to pass those in.

OBJECT SERIALISATION

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

SERIALISATION

What happens when `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, `JSON::XS` uses the Types::Serialiser object serialisation 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 serialise, and the second argument being the constant string `JSON` to distinguish it from other serialisers.

   The `FREEZE` method can return any number of values (i.e. zero or more). These values and the paclkage/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, $serialiser) = @_;

      ($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 serialised. The fatc that these values originally were URI objects is lost.

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

3. `allow_blessed` is enabled.
   The object will be serialised as a JSON null value.

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

DESERIALISATION

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

This section only considers the tagged value case: I 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, `JSON::XS` will look up the `THAW` method of the package/classname used during serialisation (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, $serialiser, $type, $id) = @_;

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

ENCODING/CODESET FLAG NOTES

The interested reader might have seen a number of flags that signify encodings or codesets – `utf8`, `latin1` 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 a UTF-8 encoded octet/binary string in Perl.

`latin1` 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.

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` 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` 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-standardised 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 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 JSON::XS;

print 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 = 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.

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

JSON and YAML

You often hear that JSON is a subset of YAML. This is, however, a mass hysteria(*) and very far from the truth (as of the time of this writing), so let me state it clearly: *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 JSON::XS to generate YAML, you should use this algorithm (subject to change in future versions):

```
my $to_yaml = 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. Please note that YAML has hardcoded limits on (simple) object key lengths that JSON doesn't have and also has different and incompatible unicode character escape syntax, so you should make sure that your hash keys are noticeably shorter than the 1024 "stream characters" YAML allows and that you do not have characters with codepoint values outside the Unicode BMP (basic multilingual page). YAML also does not allow `\/` sequences in strings (which JSON::XS does not *currently* generate, but other JSON generators might).

There might be other incompatibilities that I am not aware of (or the YAML specification has been changed yet again - it does so quite often). In general you should not try to generate YAML with a JSON generator or vice versa, or try to parse JSON with a YAML parser or vice versa: chances are high that you will run into severe interoperability problems when you least expect it.

(*) I have been pressured multiple times by Brian Ingerson (one of the authors of the YAML specification) to remove this paragraph, despite him acknowledging that the actual incompatibilities exist. As I was personally bitten by this "JSON is YAML" lie, I refused and said I will continue to educate people about these issues, so others do not run into the same problem again and again. After this, Brian called me a (quote)*complete and worthless idiot*(unquote).

In my opinion, instead of pressuring and insulting people who actually clarify issues with YAML and

the wrong statements of some of its proponents, I would kindly suggest reading the JSON spec (which is not that difficult or long) and finally make YAML compatible to it, and educating users about the changes, instead of spreading lies about the real compatibility for many *years* and trying to silence people who point out that it isn't true.

Addendum/2009: the YAML 1.2 spec is still incompatible with JSON, even though the incompatibilities have been documented (and are known to Brian) for many years and the spec makes explicit claims that YAML is a superset of JSON. It would be so easy to fix, but apparently, bullying people and corrupting userdata is so much easier.

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.

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 JSON::XS/2 uses the OO interface with pretty-printing and hashkey sorting enabled, JSON::XS/3 enables shrink. JSON::DWIW/DS uses the deserialise 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.

SECURITY CONSIDERATIONS

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 and I am trying hard on making that true, but you never know.

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, 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.

Something else could bomb you, too, that I forgot to think of. In that case, you get to keep the pieces. I am always open for hints, though...

Also keep in mind that JSON::XS might leak contents of your Perl data structures in its error messages, so when you serialise 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 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).

"OLD" VS. "NEW" JSON (RFC4627 VS. RFC7159)

JSON originally required JSON texts to represent an array or object − scalar values were explicitly not allowed. This has changed, and versions of JSON::XS beginning with `4.0` reflect this by allowing scalar values by default.

One reason why one might not want this is that this removes a fundamental property of JSON texts, namely that they are self-delimited and self-contained, or in other words, you could take any number of ''old'' JSON texts and paste them together, and the result would be unambiguously parseable:

```
[1,3]{"k":5}[][null] # four JSON texts, without doubt
```

By allowing scalars, this property is lost: in the following example, is this one JSON text (the number 12) or two JSON texts (the numbers 1 and 2):

```
12    # could be 12, or 1 and 2
```

Another lost property of ''old'' JSON is that no lookahead is required to know the end of a JSON text, i.e. the JSON text definitely ended at the last `]` or `}` character, there was no need to read extra characters.

For example, a viable network protocol with ''old'' JSON was to simply exchange JSON texts without delimiter. For ''new'' JSON, you have to use a suitable delimiter (such as a newline) after every JSON text or ensure you never encode/decode scalar values.

Most protocols do work by only transferring arrays or objects, and the easiest way to avoid problems with the ''new'' JSON definition is to explicitly disallow scalar values in your encoder and decoder:

```
$json_coder = JSON::XS->new->allow_nonref (0)
```

This is a somewhat unhappy situation, and the blame can fully be put on JSON's inmventor, Douglas Crockford, who unilaterally changed the format in 2006 without consulting the IETF, forcing the IETF to either fork the format or go with it (as I was told, the IETF wasn't amused).

RELATIONSHIP WITH I–JSON

JSON is a somewhat sloppily-defined format – it carries around obvious Javascript baggage, such as not really defining number range, probably because Javascript only has one type of numbers: IEEE 64 bit floats (''binary64'').

For this reaosn, RFC7493 defines ''Internet JSON'', which is a restricted subset of JSON that is supposedly more interoperable on the internet.

While `JSON::XS` does not offer specific support for I–JSON, it of course accepts valid I–JSON and by default implements some of the limitations of I–JSON, such as parsing numbers as perl numbers, which are usually a superset of binary64 numbers.

To generate I–JSON, follow these rules:

- always generate UTF–8

  I–JSON must be encoded in UTF–8, the default for `encode_json`.

- numbers should be within IEEE 754 binary64 range

  Basically all existing perl installations use binary64 to represent floating point numbers, so all you need to do is to avoid large integers.

- objects must not have duplicate keys

  This is trivially done, as `JSON::XS` does not allow duplicate keys.

- do not generate scalar JSON texts, use `->allow_nonref (0)`

  I–JSON strongly requests you to only encode arrays and objects into JSON.

- times should be strings in ISO 8601 format

  There are a myriad of modules on CPAN dealing with ISO 8601 – search for `ISO8601` on CPAN and use one.

- encode binary data as base64

  While it's tempting to just dump binary data as a string (and let `JSON::XS` do the escaping), for I–JSON, it's *recommended* to encode binary data as base64.

There are some other considerations – read RFC7493 for the details if interested.

INTEROPERABILITY WITH OTHER MODULES

`JSON::XS` uses the Types::Serialiser module to provide boolean constants. That means that the JSON true and false values will be compatible to true and false values of other modules that do the same, such as JSON::PP and CBOR::XS.

INTEROPERABILITY WITH OTHER JSON DECODERS

As long as you only serialise data that can be directly expressed in JSON, `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)).

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` will allow you to encode and decode Perl objects, at the cost of not outputting valid JSON 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 serialised objects, and you still want to decode the generated When you want to serialise 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 serialised 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*\[/["XU1peReLzT4ggEllLanBYq4G9Vzli
```

And after decoding the JSON text, you could walk the data structure looking for arrays with a first element of `XU1peReLzT4ggEllLanBYq4G9VzliwKF`.

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*"XU1peReLzT4ggEllLanBYq4G9VzliwKF"\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.

(I–)THREADS

This module is *not* guaranteed to be ithread (or MULTIPLICITY–) safe and there are no plans to change this. Note that perl's builtin so-called threads/ithreads are officially deprecated and should not be used.

THE PERILS OF SETLOCALE

Sometimes people avoid the Perl locale support and directly call the system's setlocale function with `LC_ALL`.

This breaks both perl and modules such as JSON::XS, as stringification of numbers no longer works correctly (e.g. `$x = 0.1; print "$x"+1` might print `1`, and JSON::XS might output illegal JSON as JSON::XS relies on perl to stringify numbers).

The solution is simple: don't call `setlocale`, or use it for only those categories you need, such as `LC_MESSAGES` or `LC_CTYPE`.

If you need `LC_NUMERIC`, you should enable it only around the code that actually needs it (avoiding stringification of numbers), and restore it afterwards.

SOME HISTORY

At the time this module was created there already were a number of JSON modules available on CPAN, so what was the reason to write yet another JSON module? While it seems there are many JSON modules, none of them correctly handled all corner cases, and in most cases their maintainers are unresponsive, gone missing, or not listening to bug reports for other reasons.

Beginning with version 2.0 of the JSON module, when both JSON and JSON::XS are installed, then JSON will fall back on JSON::XS (this can be overridden) with no overhead due to emulation (by inheriting constructor and methods). If JSON::XS is not available, it will fall back to the compatible JSON::PP module as backend, so using JSON instead of JSON::XS gives you a portable JSON API that can be fast when you

need it and doesn't require a C compiler when that is a problem.

Somewhere around version 3, this module was forked into `Cpanel::JSON::XS`, because its maintainer had serious trouble understanding JSON and insisted on a fork with many bugs "fixed" that weren't actually bugs, while spreading FUD about this module without actually giving any details on his accusations. You be the judge, but in my personal opinion, if you want quality, you will stay away from dangerous forks like that.

BUGS

While the goal of this module is to be correct, that unfortunately does not mean it's bug-free, only that I think its design is bug-free. If you keep reporting bugs they will be fixed swiftly, though.

Please refrain from using rt.cpan.org or any other bug reporting service. I put the contact address into my modules for a reason.

SEE ALSO

The *json_xs* command line utility for quick experiments.

AUTHOR

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