## NAME

JSON::MaybeXS – Use Cpanel::JSON::XS with a fallback to JSON::XS and JSON::PP

## SYNOPSIS

```
  use JSON::MaybeXS;

  my $data_structure = decode_json($json_input);

  my $json_output = encode_json($data_structure);

  my $json = JSON()->new;

  my $json_with_args = JSON::MaybeXS->new(utf8 => 1); # or { utf8 => 1 }
```

## DESCRIPTION

This module first checks to see if either Cpanel::JSON::XS or JSON::XS is already loaded, in which case it uses that module. Otherwise it tries to load Cpanel::JSON::XS, then JSON::XS, then JSON::PP in order, and either uses the first module it finds or throws an error.

It then exports the `encode_json` and `decode_json` functions from the loaded module, along with a `JSON` constant that returns the class name for calling `new` on.

If you're writing fresh code rather than replacing JSON.pm usage, you might want to pass options as constructor args rather than calling mutators, so we provide our own `new` method that supports that.

## EXPORTS

`encode_json`, `decode_json` and `JSON` are exported by default; `is_bool` is exported on request.

To import only some symbols, specify them on the `use` line:

```
  use JSON::MaybeXS qw(encode_json decode_json is_bool); # functions only

  use JSON::MaybeXS qw(JSON); # JSON constant only
```

To import all available sensible symbols (`encode_json`, `decode_json`, and `is_bool`), use `:all`:

```
  use JSON::MaybeXS ':all';
```

To import all symbols including those needed by legacy apps that use JSON::PP:

```
  use JSON::MaybeXS ':legacy';
```

This imports the `to_json` and `from_json` symbols as well as everything in `:all`. NOTE: This is to support legacy code that makes extensive use of `to_json` and `from_json` which you are not yet in a position to refactor. DO NOT use this import tag in new code, in order to avoid the crawling horrors of getting UTF–8 support subtly wrong. See the documentation for JSON for further details.

### encode_json

This is the `encode_json` function provided by the selected implementation module, and takes a perl data structure which is serialised to JSON text.

```
  my $json_text = encode_json($data_structure);
```

### decode_json

This is the `decode_json` function provided by the selected implementation module, and takes a string of JSON text to deserialise to a perl data structure.

```
  my $data_structure = decode_json($json_text);
```

### to_json, from_json

See JSON for details. These are included to support legacy code **only**.

### JSON

The `JSON` constant returns the selected implementation module's name for use as a class name – so:

```
  my $json_obj = JSON()->new; # returns a Cpanel::JSON::XS or JSON::PP object
```

and that object can then be used normally:

```
  my $data_structure = $json_obj->decode($json_text); # etc.
```

The use of parentheses here is optional, and only used as a hint to the reader that this use of `JSON` is a *subroutine* call, *not* a class name.

**is_bool**

```
  $is_boolean = is_bool($scalar)
```

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

Since this is a bare sub in the various backend classes, it cannot be called as a class method like the other interfaces; it must be called as a function, with no invocant. It supports the representation used in all JSON backends.

## CONSTRUCTOR

**new**

With JSON::PP, JSON::XS and Cpanel::JSON::XS you are required to call mutators to set options, such as:

```
  my $json = $class->new->utf8(1)->pretty(1);
```

Since this is a trifle irritating and noticeably un-perlish, we also offer:

```
  my $json = JSON::MaybeXS->new(utf8 => 1, pretty => 1);
```

which works equivalently to the above (and in the usual tradition will accept a hashref instead of a hash, should you so desire).

The resulting object is blessed into the underlying backend, which offers (at least) the methods `encode` and `decode`.

## BOOLEANS

To include JSON-aware booleans (`true`, `false`) in your data, just do:

```
    use JSON::MaybeXS;
    my $true = JSON()->true;
    my $false = JSON()->false;
```

The booleans are also available as subs or methods on JSON::MaybeXS.

```
    use JSON::MaybeXS ();
    my $true = JSON::MaybeXS::true;
    my $true = JSON::MaybeXS->true;
    my $false = JSON::MaybeXS::false;
    my $false = JSON::MaybeXS->false;
```

## CONVERTING FROM JSON::Any

JSON::Any used to be the favoured compatibility layer above the various JSON backends, but over time has grown a lot of extra code to deal with legacy backends (e.g. JSON::Syck) that are no longer needed. This is a rough guide of translating such code:

Change code from:

```
    use JSON::Any;
    my $json = JSON::Any->new->objToJson($data);    # or to_json($data), or Dump($da
```

to:

```
    use JSON::MaybeXS;
    my $json = encode_json($data);
```

Change code from:

```
    use JSON::Any;
    my $data = JSON::Any->new->jsonToObj($json);     # or from_json($json), or Load($
```

to:

```
    use JSON::MaybeXS;
    my $json = decode_json($data);
```

## CAVEATS

The `new()` method in this module is technically a factory, not a constructor, because the objects it returns will *NOT* be blessed into the `JSON::MaybeXS` class.

If you are using an object returned by this module as a Moo(se) attribute, this type constraint code:

```
    is 'json' => ( isa => 'JSON::MaybeXS' );
```

will *NOT* do what you expect. Instead, either rely on the `JSON` class constant described above, as so:

```
    is 'json' => ( isa => JSON::MaybeXS::JSON() );
```

Alternatively, you can use duck typing:

```
    use Moose::Util::TypeConstraints 'duck_type';
    is 'json' => ( isa => Object , duck_type([qw/ encode decode /]));
```

## INSTALLATION

At installation time, *Makefile.PL* will attempt to determine if you have a working compiler available, and therefore whether you are able to run XS code. If so, Cpanel::JSON::XS will be added to the prerequisite list, unless JSON::XS is already installed at a high enough version. JSON::XS may also be upgraded to fix any incompatibility issues.

Because running XS code is not mandatory and JSON::PP (which is in perl core) is used as a fallback backend, this module is safe to be used in a suite of code that is fatpacked or installed into a restricted-resource environment.

You can also prevent any XS dependencies from being installed by setting `PUREPERL_ONLY=1` in *Makefile.PL* options (or in the `PERL_MM_OPT` environment variable), or using the `--pp` or `--pureperl` flags with the cpanminus client.

## AUTHOR

mst – Matt S. Trout (cpan:MSTROUT) <mst@shadowcat.co.uk>

## CONTRIBUTORS

- Clinton Gormley <drtech@cpan.org>
- Karen Etheridge <ether@cpan.org>
- Kieren Diment <diment@gmail.com>

## COPYRIGHT

Copyright (c) 2013 the `JSON::MaybeXS` "AUTHOR" and "CONTRIBUTORS" as listed above.

## LICENSE

This library is free software and may be distributed under the same terms as perl itself.