NAME

Cpanel::JSON::XS::Type − Type support for JSON encode

SYNOPSIS

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


encode_json([10, "10", 10.25], [JSON_TYPE_INT, JSON_TYPE_INT, JSON_TYPE_STRING]);
# '[10,10,"10.25"]'

encode_json([10, "10", 10.25], json_type_arrayof(JSON_TYPE_INT));
# '[10,10,10]'

encode_json(1, JSON_TYPE_BOOL);
# 'true'

my $perl_struct = { key1 => 1, key2 => "2", key3 => 1 };
my $type_spec = { key1 => JSON_TYPE_STRING, key2 => JSON_TYPE_INT, key3 => JSON_TYP
my $json_string = encode_json($perl_struct, $type_spec);
# '{"key1":"1","key2":2,"key3":true}'

my $perl_struct = { key1 => "value1", key2 => "value2", key3 => 0, key4 => 1, key5
my $type_spec = json_type_hashof(JSON_TYPE_STRING);
my $json_string = encode_json($perl_struct, $type_spec);
# '{"key1":"value1","key2":"value2","key3":"0","key4":"1","key5":"string","key6":"s

my $perl_struct = { key1 => { key2 => [ 10, "10", 10.6 ] }, key3 => "10.5" };
my $type_spec = { key1 => json_type_anyof(JSON_TYPE_FLOAT, json_type_hashof(json_ty
my $json_string = encode_json($perl_struct, $type_spec);
# '{"key1":{"key2":[10,10,10]},"key3":10.5}'


my $value = decode_json('false', 1, my $type);
# $value is 0 and $type is JSON_TYPE_BOOL

my $value = decode_json('0', 1, my $type);
# $value is 0 and $type is JSON_TYPE_INT

my $value = decode_json('"0"', 1, my $type);
# $value is 0 and $type is JSON_TYPE_STRING

my $json_string = '{"key1":{"key2":[10,"10",10.6]},"key3":"10.5"}';
my $perl_struct = decode_json($json_string, 0, my $type_spec);
# $perl_struct is { key1 => { key2 => [ 10, 10, 10.6 ] }, key3 => 10.5 }
# $type_spec is { key1 => { key2 => [ JSON_TYPE_INT, JSON_TYPE_STRING, JSON_TYPE_FL
```

DESCRIPTION

This module provides stable JSON type support for the Cpanel::JSON::XS encoder which doesn't depend on any internal perl scalar flags or characteristics. Also it provides real JSON types for Cpanel::JSON::XS decoder.

In most cases perl structures passed to encode_json come from other functions or from other modules and caller of Cpanel::JSON::XS module does not have control of internals or they are subject of change. So it is not easy to support enforcing types as described in the simple scalars section.

For services based on JSON contents it is sometimes needed to correctly process and enforce JSON types.

The function decode_json takes optional third scalar parameter and fills it with specification of json types.

The function encode_json takes a perl structure as its input and optionally also a json type specification in the second parameter.

If the specification is not provided (or is undef) internal perl scalar flags are used for the resulting JSON type. The internal flags can be changed by perl itself, but also by external modules. Which means that types in resulting JSON string aren't stable. Specially it does not work reliable for dual vars and scalars which were used in both numeric and string operations. See simple scalars.

To enforce that specification is always provided use `require_types`. In this case when `encode` is called without second argument (or is undef) then it croaks. It applies recursively for all sub-structures.

JSON type specification for scalars:

JSON_TYPE_BOOL
: It enforces JSON boolean in resulting JSON, i.e. either `true` or `false`. For determining whether the scalar passed to the encoder is true, standard perl boolean logic is used.

JSON_TYPE_INT
: It enforces JSON number without fraction part in the resulting JSON. Equivalent of perl function int is used for conversion.

JSON_TYPE_FLOAT
: It enforces JSON number with fraction part in the resulting JSON. Equivalent of perl operation `+0` is used for conversion.

JSON_TYPE_STRING
: It enforces JSON string type in the resulting JSON.

JSON_TYPE_NULL
: It represents JSON `null` value. Makes sense only when passing perl's `undef` value.

For each type, there also exists a type with the suffix `_OR_NULL` which encodes perl's `undef` into JSON `null`. Without type with suffix `_OR_NULL` perl's `undef` is converted to specific type according to above rules.

JSON type specification for arrays:

[...] The array must contain the same number of elements as in the perl array passed for encoding. Each element of the array describes the JSON type which is enforced for the corresponding element of the perl array.

json_type_arrayof
: This function takes a JSON type specification as its argument which is enforced for every element of the passed perl array.

JSON type specification for hashes:

{...}
: Each hash value for corresponding key describes the JSON type specification for values of passed perl hash structure. Keys in hash which are not present in passed perl hash structure are simple ignored and not used.

json_type_hashof
: This function takes a JSON type specification as its argument which is enforced for every value of passed perl hash structure.

JSON type specification for alternatives:

json_type_anyof
: This function takes a list of JSON type alternative specifications (maximally one scalar, one array, and one hash) as its input and the JSON encoder chooses one that matches.

json_type_null_or_anyof
: Like `json_type_anyof`, but scalar can be only perl's `undef`.

Recursive specifications

json_type_weaken

This function can be used as an argument for "json_type_arrayof", "json_type_hashof" or "json_type_anyof" functions to create weak references suitable for complicated recursive structures. It depends on the weaken function from Scalar::Util module. See following example:

```
my $struct = {
    type => JSON_TYPE_STRING,
    array => json_type_arrayof(JSON_TYPE_INT),
};
$struct->{recursive} = json_type_anyof(
    json_type_weaken($struct),
    json_type_arrayof(JSON_TYPE_STRING),
);
```

If you want to encode all perl scalars to JSON string types despite how complicated is input perl structure you can define JSON type specification for alternatives recursively. It could be defined as:

```
my $type = json_type_anyof();
$type->[0] = JSON_TYPE_STRING_OR_NULL;
$type->[1] = json_type_arrayof(json_type_weaken($type));
$type->[2] = json_type_hashof(json_type_weaken($type));

print encode_json([ 10, "10", { key => 10 } ], $type);
# ["10","10",{"key":"10"}]
```

An alternative solution for encoding all scalars to JSON strings is to use `type_all_string` method of Cpanel::JSON::XS itself:

```
my $json = Cpanel::JSON::XS->new->type_all_string;
print $json->encode([ 10, "10", { key => 10 } ]);
# ["10","10",{"key":"10"}]
```

AUTHOR

Pali <pali@cpan.org>

COPYRIGHT & LICENSE

Copyright (c) 2017, GoodData Corporation. All rights reserved.

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