

Strings

Str, Caster and Encoding of Derafu Support

Anonymous · 08/10/2026

Strings

Str

String helpers. They handle UTF-8.

Method	What it does
<code>Str::format(\$template, \$data, \$style = 'mustache')</code>	Replaces the placeholders of a text.
<code>Str::studly(\$value), camel(\$value), snake(\$value, \$delimiter = '_')</code>	HelloWorld, helloWorld and hello_world.
<code>Str::slug(\$string, \$separator = '-', \$encoding = 'UTF-8')</code>	A text for a URL.
<code>Str::pad(\$string, \$length, \$padStr = ' ', \$padType = STR_PAD_RIGHT, \$encoding = null)</code>	<code>str_pad()</code> that counts characters, not bytes.
<code>Str::wordWrap(\$string, \$characters = 64, \$break = "\n", \$cutLongWords = true)</code>	Wraps a text at a width.
<code>Str::extract(\$text, \$startDelimiter, \$endDelimiter, \$offset = 0)</code>	The text between two delimiters.
<code>Str::splitParameters(\$parameters, \$delimiter = ',')</code>	Splits a list of parameters.
<code>Str::uuid4()</code>	A UUID v4 (RFC 4122).
<code>Str::random(\$length = 12, \$useUppercase = true, \$useNumbers = true, \$useSpecial = false)</code>	A random string.
<code>Str::utf8decode(\$input), Str::utf8encode(\$input)</code>	The ones of Encoding , for a string.

Placeholders

```
use Derafu\Support\Str;

Str::format('Hello {{name}}!', ['name' => 'John']); // "Hello John!"
Str::format('Hi { name }, you are { age }', $data, 'simple'); // "Hi Ana, you are 3"
Str::format('WHERE id = :id', ['id' => 5], 'sql'); // "WHERE id = 5"
Str::format('Hi %(name)s', ['name' => 'Ana'], 'python'); // "Hi Ana"
```

```
Str::format('Hi $name', ['name' => 'Ana'], 'dollar'); // "Hi Ana"
Str::format('{{a}} :b', ['a' => 1, 'b' => 2], 'mustache|sql'); // "1 2"
```

The styles are `mustache` (`{{ key }}`), `simple` (`{ key }`), `sql` (`:key`), `python` (`%(key)s`) and `dollar` (`$key`), and several can be used together separated by `|`. A placeholder that has no value in `$data` stays as it is (`{{x}}`). A style that does not exist is an `InvalidArgumentException`.

`Str::format()` replaces text: it does not escape anything. Do not use it to build SQL or HTML with data of a user.

Case, slugs and padding

```
Str::studly('hello_world-foo bar'); // "HelloWorldFooBar"
Str::camel('hello_world-foo bar'); // "helloWorldFooBar"
Str::snake('HelloWorldFoo'); // "hello_world_foo"
Str::snake('HelloWorld', '-'); // "hello-world"

Str::slug('Ñandú y Camión!! '); // "nandu-y-camion"
Str::slug('Hello World', '_'); // "hello_world"

Str::pad('ñu', 6, '.', STR_PAD_BOTH); // "...ñu..."
```

A slug is lowercase, has only letters, numbers and the separator, has no repeated or leading or trailing separators, and the accents are removed.

Pieces of a text

```
Str::extract('a [b c] d', '[', ']');
// ['string' => 'b c', 'start' => 3, 'end' => 5, 'length' => 3]
Str::extract('a b', '[', ']'); // null

Str::splitParameters('a, b\, c, d'); // ['a', 'b, c', 'd']

Str::wordWrap('The quick brown fox', 10); // "The quick\nbrown fox"
```

`extract()` gives the text between the delimiters (without the spaces around it), the position where it starts, the position of its last character and its length, or `null` if either delimiter is not found.

`splitParameters()` trims each parameter, and a delimiter escaped with a backslash is part of the parameter.

UUIDs and random strings

`Str::uuid4()` gives a UUID v4 made with `random_bytes()`.

`Str::random()` gives 12 characters by default: lowercase letters (always), and uppercase letters and numbers (the options remove them). `$useSpecial` adds special characters (`!@#$$%^&*()_+-`).

`=[]{}|;:,<.>?)`. The characters are chosen with `random_int()`, and the string has one of each kind that is enabled when the length has room for them (a length of 2 can not have four kinds). A length less than 1 is an `InvalidArgumentException`.

Caster

Turns a value into the scalar that it says, the way a configuration file or a spreadsheet is read.

```
use Derafu\Support\Caster;

Caster::cast('42');      // 42
Caster::cast('3.14');    // 3.14
Caster::cast('1e3');     // 1000.0
Caster::cast('yes');     // true
Caster::cast('null');    // null

Caster::castWithType('42'); // ['value' => 42, 'type' => 'integer']
```

The rules for a text (it is trimmed to decide, and it is given back **as it came** if it says nothing):

The text says	It gives
An empty text, <code>null</code> , <code>nil</code> (in any case)	<code>null</code>
<code>true</code> , <code>yes</code> , <code>on</code> (in any case)	<code>true</code>
<code>false</code> , <code>no</code> , <code>off</code> (in any case)	<code>false</code>
An integer: 42, -3, +5, 1_000, 007	The <code>int</code> (007 is 7)
A decimal or scientific number: 3.14, .5, 1e3, 1_000.50	The <code>float</code>
Anything else (abc, 1,5, 0x1A)	The text, as it came

A value that is already `null`, a boolean, an integer or a float stays as it is. Any other value is turned into a string, without warnings: an array into its JSON (`Caster::cast(['a' => 1])` is `'{"a":1}'`), an object that has `__toString()` into its text, any other object into the name of its class (`stdClass`), and a resource as PHP writes it (`Resource id #5`).

An empty text is `null`, and the leading zeros of a number are lost: do not cast the values where they matter, like the codes, the postal codes or the identifiers that look like numbers.

`castWithType()` gives the value and its type as `gettype()` says it (`integer`, `double`, `boolean`, `string`, `NULL`).

Encoding

Converts between UTF-8 and ISO-8859-1, also inside arrays and objects.

```
use Derafu\Support\Encoding;

Encoding::utf8decode('ñ'); // The ISO-8859-1 text ("\xF1").
Encoding::utf8encode("\xF1"); // "ñ"
Encoding::utf8encode('ñ'); // "ñ": it was already UTF-8.
Encoding::utf8decode(['name' => 'Ñandú']); // The same array, with its texts
converted.
```

- **utf8decode()** (UTF-8 to ISO-8859-1) only converts a text that is valid UTF-8; any other is given back as it came.
- **utf8encode()** (ISO-8859-1 to UTF-8) converts a text that is **not** already valid UTF-8, so it can be applied twice, or to text that may be either, without writing $\tilde{\pm}$ instead of ñ. The limit is a text of ISO-8859-1 whose bytes also make a valid UTF-8 sequence (the text $\tilde{\pm}$ in ISO-8859-1, for example): it can not be told from UTF-8, and it is not converted.
- **Arrays and objects** are converted recursively, and the values that are not texts stay as they are. An **object is changed in place** (the same instance is given back).

Last updated on 08/10/2026