Docs

Support Project

Derafu Support

Anonymous · 15/11/2026 · #php

Table of Contents

Support Project 3
- ***Introduction*** 4
- ***CSV*** 6
- ***Ip*** 8
- ***Objects*** 13
- ***Strings*** 17
- ***Url*** 20

Support Project

Derafu Support

Anonymous · 15/11/2026 · #php

Derafu Support

Last updated on 15/11/2026

Introduction

Essential PHP utilities of Derafu for strings, CSV, objects, IP addresses and URL paths

Anonymous · 15/11/2026

Derafu: Support



A collection of essential PHP utility classes for the common work of an application: strings, CSV, objects, IP addresses and URL paths.

Every class is `final` and has only static methods. They keep no state: what a call does depends on its arguments (and on the file system, in the classes that read or write files), so it gives the same answer anywhere and is easy to test.

The classes

Class	What it is for	Page
`Str`	Placeholders, case conversion, slugs, multibyte padding, UUIDs and random strings.	Strings
`Encoding`	UTF-8 and ISO-8859-1, also inside arrays and objects.	Strings
`Csv`	Reading, writing, generating and sending CSV.	CSV
`Obj`, `Factory`, `Hydrator`	Create objects from arrays and look at them.	Objects
`Serialization`, `JsonSerializer`	Is it serializable, and objects to JSON.	Objects
`Debug`	Information about a variable and where it was inspected.	Objects
`Ip`	IPv4 and IPv6: validation, normalization, ranges and networks.	Ip
`Url`	The canonical form of the paths of URLs, and prefixes by segments.	Url

For dates, months, periods, working days and date formatting, see Derafu Date. For files, directories, paths, ZIP archives and MIME types, see Derafu File. For typed array reading and array transformations, see Derafu Array. For value inspection, type validation, casting and parsing, see Derafu Value.

Why Derafu\Support?

It solves needs of business and data processing that the standard library leaves out:

- **CSV across systems:** separators, enclosures and encodings.
- **Addresses that are handled right:** IPv6, ranges and networks, with the same canonical form for the same address.

Install

```
composer require derafu/support
```

`derafu/array` provides the array flattening used by `Str::format()`. Its exceptions use `derafu/translation` (see Errors). The following integrations are optional; install the dependencies of the classes you use:

Package	Needed by
`league/csv` (9.27 or later)	`Csv` (all of it).

The string classes (`Str`, `Encoding`) use the functions of `mbstring`.

Errors

The errors that the classes throw are **translatable**: they are the standard exceptions of PHP (`InvalidArgumentException`, `RuntimeException`, `LogicException`) in the translatable version of `derafu/translation`, so catching the standard class works, and the message is also available as a message id with its parameters.

The Spanish translations ship with the package. To have them in your translator, import its services (or add `SupportTranslationResourceProvider` to the resource providers of the translator):

```
imports:
    - { resource: '../vendor/derafu/support/resources/config/support-services.yaml' }
```

`derafu/foundation` already does it.

Last updated on 15/11/2026

CSV

Csv of Derafu Support

Anonymous · 15/11/2026

CSV

Simplified reading and writing of CSV with `league/csv`. The defaults are the ones that spreadsheets in Spanish use: separator `;`, enclosure `"` and escape `\`.

Method	What it does
`Csv::read($file, $separator = ';', $enclosure = '"', $escape = '\\', $encoding = 'UTF-8')`	The rows of a file.
`Csv::load($content, ...)`	The rows of a text (same parameters).
`Csv::write($data, $file, $separator, $enclosure, $escape)`	Writes records to a file.
`Csv::generate($data, $separator, $enclosure, $escape, $encoding = 'UTF-8')`	The CSV text of some records.
`Csv::send($data, $filename, $separator, $enclosure, $escape, $sendHeaders = true)`	Sends the CSV as a download.
`Csv::validate($file, $separator, $enclosure, $escape, $sampleSize = 5)`	Whether a file looks like a CSV.

Reading

`read()` and `load()` give **a list of rows**, each one a list of texts. The first row is the header like any other, so take it off if the file has one:

```
use Derafu\Support\Csv;

$rows = Csv::read('file.csv');             // [['a', 'b'], ['1', '2'], ['3', '4']]
$rows = Csv::load("x,y\n1,2", ',');        // [['x', 'y'], ['1', '2']]

$header = array_shift($rows);
$records = array_map(fn (array $row) => array_combine($header, $row), $rows);
```

`$encoding` is the encoding of what is read (`'ISO-8859-1'` for a file from an old system): the rows come out in UTF-8.

Writing

`write()`, `generate()` and `send()` take **records**: a list of associative arrays. The first line is the header, with the keys of the first record.

```
$data = [['a' => 1, 'b' => 2], ['a' => 3, 'b' => 4]];

Csv::generate($data);                       // "a;b\n1;2\n3;4\n"
Csv::write($data, 'export.csv');            // The same, in a file.
Csv::send($data, 'export.csv');             // As a download.

Csv::generate($data, encoding: 'ISO-8859-1');  // For the system that wants it.
```

A value with the separator inside is put in the enclosure (`a;b` is written as `"a;b"`). `generate()` can write in another encoding; `send()` always sends UTF-8 (`Content-Type: text/csv; charset=UTF-8`, as an attachment, with no cache) and it throws if the headers were already sent and does not end the script; `$sendHeaders = false` only writes the content.

Errors

The failures are `RuntimeException`s (a `LogicException` when `write()` can not write the file), with the file and the reason in the message.

Validating

`validate()` says whether the file exists and can be read, whether its first `$sampleSize` rows can be read, and whether they all have the same number of columns. It does not say that the content is right, only that it looks like a CSV with that separator.

Last updated on 15/11/2026

Ip

Ip of Derafu Support, IPv4 and IPv6 addresses and ranges

Anonymous · 15/11/2026

Ip

`Ip` works with IP addresses (IPv4 and IPv6) and ranges of them (CIDR). It is made of pure functions: they depend only on their arguments, never on `$_SERVER`, so they give the same answer anywhere.

What it does **not** do is say which address is the one of the client of a request. That depends on the deployment (which proxies are trusted to say it), and it is not a property of an address: take it from where it is decided, and use `Ip` to check and compare it.

Two rules apply to the whole class:

- **An address that is not valid is a question that has an answer.** The functions that ask something about an address (`isPrivate()`, `inRange()`…) say `false`, and the ones that give an address (`normalize()`, `network()`…) say `null`.
- **A range or a prefix that is not valid is an error of the configuration.** It is an `InvalidArgumentException` (translatable, see Errors): a range that silently matches nothing is a hole that nobody sees.

An IPv4 address written as IPv6 (`::ffff:192.0.2.1`, which is what a socket of IPv6 gives for a client of IPv4) **is** the IPv4 address: `normalize()` gives it as IPv4, and the questions are asked about the IPv4 address.

Validating and normalizing

Method	What it does
`Ip::isValid($ip)`	Whether the text is an IP address.
`Ip::version($ip)`	`4`, `6` or `null`.
`Ip::isIpv4($ip)`, `Ip::isIpv6($ip)`	Whether it is of that version (an IPv4 written as IPv6 is IPv6).
`Ip::normalize($ip)`	The canonical form of the address, or `null`.
`Ip::expand($ip)`	The long form of an IPv6, with its eight groups.
`Ip::parse($address)`	The address inside a text with a port, brackets, a zone or quotes.

```
use Derafu\Support\Ip;
```

```
Ip::isValid('192.0.2.1');           // true
Ip::isValid('010.0.0.1');           // false: it could be read as octal.
Ip::isValid('192.0.2.1:80');        // false: it is not only the address.
Ip::version('2001:db8::1');         // 6

Ip::normalize('2001:DB8:0:0:0:0:0:1');   // "2001:db8::1"
Ip::normalize('::ffff:192.0.2.1');       // "192.0.2.1"
Ip::expand('2001:db8::1');               // "2001:0db8:0000:0000:0000:0000:0000:0001"
```

The canonical form makes the same address always the same text, so it is safe to use as a key, to compare or to store: IPv6 is in lowercase and compressed as RFC 5952 says (the longest run of zero groups becomes `::`, and a single zero group is not compressed), and an IPv4 written as IPv6 is IPv4. The text does not depend on the system.

`isValid()` accepts **only the address**. `parse()` takes it out of the forms that headers and logs use, and normalizes it:

```
Ip::parse('192.0.2.1:8080');          // "192.0.2.1"
Ip::parse('[2001:DB8::1]:443');       // "2001:db8::1"
Ip::parse('[2001:db8::1]');           // "2001:db8::1"
Ip::parse('fe80::1%eth0');            // "fe80::1": the zone is removed.
Ip::parse('"[2001:db8::1]:4711"');    // "2001:db8::1": quoted, as the header
Forwarded does.
Ip::parse('::1:80');                  // "::1:80": without brackets an IPv6 has no
port.
Ip::parse('unknown');                 // null
Ip::parse('example.com:80');          // null: it does not resolve names.
```

Kinds of address

Method	It is `true` for
`Ip::isPrivate($ip)`	The private networks (RFC 1918: `10.0.0.0/8`, `172.16.0.0/12`, `192.168.0.0/16`) and the unique local addresses of IPv6 (`fc00::/7`).
`Ip::isReserved($ip)`	The ranges that the IANA reserves, as PHP defines them: the loopback, `0.0.0.0/8` and `::`, the link-local ones, `240.0.0.0/4` and the others that PHP knows.
`Ip::isLoopback($ip)`	`127.0.0.0/8` and `::1`.
`Ip::isLinkLocal($ip)`	`169.254.0.0/16` and `fe80::/10`.
`Ip::isPublic($ip)`	A valid address that is neither private nor reserved.

```
Ip::isPrivate('10.0.0.1');        // true
Ip::isPrivate('127.0.0.1');       // false: the loopback is reserved, not private.
Ip::isReserved('127.0.0.1');      // true
Ip::isPublic('8.8.8.8');          // true
```

```
Ip::isPublic('::ffff:10.0.0.1');  // false: it is the private 10.0.0.1.
Ip::isPublic('nope');             // false
```

The addresses that are only for documentation (`192.0.2.0/24`, `203.0.113.0/24`, `2001:db8::/32`) are not excluded by PHP, so they are public here.

Ranges

A range is written in CIDR notation (`10.0.0.0/8`, `2001:db8::/32`), or it is an address alone (`192.0.2.1`, a range of one). The bits that the prefix does not use are ignored (`10.1.2.3/8` is `10.0.0.0/8`).

Method	What it does
`Ip::inRange($ip, $range)`	Whether the address is in the range.
`Ip::inAnyRange($ip, $ranges)`	Whether it is in any of the ranges (a list, or any iterable).
`Ip::isRange($range)`	Whether the text is a valid range.
`Ip::range($range)`	The first and the last address of the range.
`Ip::cidr($ip, $ipv4Prefix = 32, $ipv6Prefix = 64)`	The network of an address, written as a range (see Networks).

```
Ip::inRange('10.1.2.3', '10.0.0.0/8');                    // true
Ip::inRange('172.31.255.255', '172.16.0.0/12');           // true
Ip::inRange('2001:db8:1::5', '2001:db8::/32');            // true
Ip::inRange('192.0.2.1', '::/0');                         // false: an address is
never in a range of the other version.
Ip::inRange('::ffff:10.1.2.3', '10.0.0.0/8');             // true: it is the IPv4
address.
Ip::inRange('nope', '0.0.0.0/0');                         // false: it is not an
address, so it is in nothing.

Ip::inAnyRange($ip, ['10.0.0.0/8', '192.168.0.0/16', '::1']);

Ip::range('192.168.1.0/24');       // ['192.168.1.0', '192.168.1.255']
Ip::range('2001:db8::/32');        // ['2001:db8::',
'2001:db8:ffff:ffff:ffff:ffff:ffff:ffff']
```

A range that is not valid (`10.0.0.0/33`, `10.0.0.0/8/9`, `localhost`, `10.0.0.0/255.0.0.0`) throws, **even if the address is not an IP address**, and `inAnyRange()` checks all the ranges, not only the ones before the one that matches. So a wrong configuration shows the first time it is used, not when an address happens to fall after it.

Networks

`Ip::network($ip, $ipv4Prefix = 32, $ipv6Prefix = 64)` gives the **network** that an address belongs to: the address with the bits of the host set to zero (the network address, as RFC 4632 calls it). `Ip::cidr()` takes the same arguments and gives the network written as a range, with its prefix, so the result says what it is and can be used as one (`inRange()`, `range()`).

It is the key to count or to limit by network instead of by address. A client of IPv6 has a whole `/64` (or more), and changing its address within it costs nothing, so counting by address does not limit anything:

```
Ip::network('2001:db8:1:2:3:4:5:6');          // "2001:db8:1:2::": the /64 by default.
Ip::network('2001:db8:1:2:3:4:5:6', 32, 48);  // "2001:db8:1::"
Ip::network('192.0.2.77');                    // "192.0.2.77": an IPv4 is the address
by default.
Ip::network('192.0.2.77', 24);                // "192.0.2.0"
Ip::network('nope');                          // null

Ip::cidr('2001:db8:1:2:3:4:5:6');             // "2001:db8:1:2::/64"
Ip::cidr('192.0.2.77');                       // "192.0.2.77/32": a network of one
address.
Ip::cidr('192.0.2.77', 24);                   // "192.0.2.0/24"
Ip::cidr('100.100.100.100', 10);              // "100.64.0.0/10"
```

Each prefix is only used for the addresses of its version. A prefix that is not valid for the version of the address (`33` for IPv4, `129` for IPv6, a negative one) is an `InvalidArgumentException`.

Comparing and binary form

```
$ips = ['2001:db8::2', '10.0.0.10', '10.0.0.2', '2001:db8::1', '9.9.9.9'];
usort($ips, Ip::compare(...));
// ['9.9.9.9', '10.0.0.2', '10.0.0.10', '2001:db8::1', '2001:db8::2']

Ip::toBinary('192.0.2.1');        // "\xC0\x00\x02\x01" (4 bytes for IPv4, 16 for
IPv6).
Ip::fromBinary("\xC0\x00\x02\x01"); // "192.0.2.1"
```

`compare()` orders by value, not by text (`10.0.0.2` goes before `10.0.0.10`): every IPv4 goes before every IPv6, and the same address written in two ways is equal. An argument that is not an address is an `InvalidArgumentException`, because it can not be ordered.

`toBinary()` gives the bytes in which an address is stored or compared (`null` if it is not an address). `fromBinary()` goes back: it needs 4 or 16 bytes (`null` otherwise), and it gives an IPv4 written as IPv6 as IPv6 (`::ffff:c000:201`): the bytes say what it is. Use `normalize()` to have it as IPv4.

Last updated on 15/11/2026

Objects

Obj, Factory, Hydrator, Serialization, JsonSerializer and Debug of Derafu Support

Anonymous · 15/11/2026

Objects

Obj

Looks at an object and fills its public properties.

Method	What it does
`Obj::fill($instance, $data)`	Sets the public properties from an array. Gives the object back.
`Obj::getPublicProperties($instance)`	The public properties, as `ReflectionProperty` by name.
`Obj::getPublicValues($instance)`	The values of the public properties.
`Obj::hasProperty($instance, $property, $public = true)`	Whether the property exists (only public ones by default).
`Obj::getClassName($instance)`	The name of the class.
`Obj::getReflection($instance)`	A `ReflectionClass` of the object.

```
use Derafu\Support\Obj;

class Q
{
    public $x = 1;
    protected $y = 2;
    private $z = 3;
    public function __construct(public int $w = 4) {}
}

$q = new Q();

Obj::fill($q, ['x' => 10]);
Obj::getPublicValues($q);                // ['x' => 10, 'w' => 4]
array_keys(Obj::getPublicProperties($q)); // ['x', 'w']
Obj::hasProperty($q, 'y');               // false: it is not public.
Obj::hasProperty($q, 'y', false);        // true
Obj::getClassName($q);                   // "Q"
```

Factory and Hydrator

`Hydrator` fills an object from an array using its setters, and `Factory` creates the object and hydrates it.

```
use Derafu\Support\Factory;
use Derafu\Support\Hydrator;

$object = Factory::create(['a' => 1]);                        // A stdClass.
$object = Factory::create($data, MyClass::class);             // A MyClass.
$objects = Factory::createMany($rows, MyClass::class);        // A list of MyClass.
$object = Factory::createAndEnsureType($data, MyClass::class, MyInterface::class);

$object = Hydrator::hydrate($instance, $data);                // An object that already
exists.
$object = Hydrator::createAndHydrate(MyClass::class, $data);
```

For each attribute of the array, `Hydrator` tries, in this order:

1. The method `setXxx()` (`Xxx` is the attribute in StudlyCase: `first_name` is `setFirstName()`).
2. The method `setXxxAttribute()`.
3. The method `setAttribute($name, $value)`, if the class has it (it receives every attribute that the previous two did not take).
4. The property with that name. It is set directly, **also if it is not public**.

An attribute that none of them can take is a `LogicException` that says which attribute and which class. `Factory::createAndEnsureType()` also throws a `LogicException` if the object that it created is not of the expected type.

Because of the last step, do not hydrate an object with data that comes from a user as it is: the attributes are the names of its properties. Pick the fields first.

Serialization and JSON

```
use Derafu\Support\Serialization;
use Derafu\Support\JsonSerializer;

Serialization::isSerializable([1, 'a']);            // true
Serialization::isSerializable('text');              // true
Serialization::isSerializable(new stdClass());      // true
Serialization::isSerializable(fn () => 1);          // false: PHP does not serialize
closures.
Serialization::isSerializable([fopen('php://memory', 'r')]);  // false: a resource
inside.
```

```
Serialization::isSerialized(serialize([1]));        // true
Serialization::isSerialized('abc');                 // false
Serialization::isSerialized('b:0;');                // true: it is a serialized
`false`.
```

`isSerializable()` answers what PHP can do. A scalar and `null` can be serialized. An array or an object can be, if PHP can serialize it **and** what it has inside can be too. They are not serializable:

- **Resources**, open or closed: PHP would write them as `0`.
- **What PHP does not serialize:** closures, generators, anonymous classes, reflections, and the objects whose class forbids it or whose `__serialize()` or `__sleep()` fails (or warns).
- **An array or an object that has any of them inside**, in a property of any visibility, at any depth. For an object with `__serialize()`, what is searched is what that method gives, which is what PHP serializes.

Circular references are fine, PHP serializes them. `isSerialized()` says whether a text is the result of `serialize()` (it checks its format, it does not unserialize it).

`JsonSerializer::serialize($value, $flags = null, $depth = 512)` makes the JSON of a value that `json_encode()` would not handle well, going through arrays and objects recursively. An object becomes:

- what its `jsonSerialize()` gives, if it is a `JsonSerializable`;
- what its `toArray()` gives, if it has that method;
- a text in ISO 8601 if it is a `DateTimeInterface`;
- its text if it is a `Stringable`;
- an array of its public properties otherwise.

```
JsonSerializer::serialize([
    'date' => new DateTimeImmutable('2024-01-02 03:04:05'),
    'text' => $stringable,
    'data' => $objectWithToArray,
]);
// {"date": "2024-01-02T03:04:05-03:00", "text": "...", "data": {...}}
```

The flags by default make a readable output (pretty printed, without escaping unicode or slashes). A value that can not be encoded is a `JsonException`.

Debug

```
use Derafu\Support\Debug;

$info = Debug::inspect($var, 'myVar');
Debug::print($var, 'myVar');
```

`inspect()` gives an array with the `label`, the `type` (the class for an object), the `length` (of an array or a text), the `file` and `line` where it was called, the `caller`, the `timestamp`, the `memory_usage` and the `value` as text (`print_r()`, or `JSON` for `null` and booleans). The `caller` is the function or method that called it (`inAFunction()`, `Class::method()`), or `{main}` if it was called from the code that is outside of any function, as PHP says it in its traces. `print()` shows that array inside a `<pre>` (so it is for pages, not for the console), with the file, the line and the caller of the call to `print()`.

Last updated on 15/11/2026

Strings

Str and Encoding of Derafu Support

Anonymous · 15/11/2026

Strings

Str

String helpers. They handle UTF-8.

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

For scalar conversion and automatic value parsing, see Derafu Value.

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 `Ã±` instead of `ñ`. The limit is a text of ISO-8859-1 whose bytes also make a valid UTF-8 sequence (the text `Ã±` 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 15/11/2026

Url

Url of Derafu Support, the canonical form of the paths of URLs

Anonymous · 15/11/2026

Url

`Url` works with the **paths** of URLs. It is made of pure functions that depend only on their arguments, and it works on a path, not on a whole URL: a `?` or a `#` in it is a character of the path, as any other.

Why a canonical form

A path can be written in many ways that mean the same thing for whoever reads it: `/api//index`, `/api/./index`, `/api/%69ndex` and `/api/index/` are all `/api/index`. When a component decides by the **text** of the path (a rule that protects `/api/index`) and another one reads its **meaning** (a router, a program that serves files), they must agree on which is the path. If they do not, what the first one does not recognize gets to the second one: `/api//index` is not under the rule `/api/index`, but the API serves it as if it was.

The functions of `Url` give one canonical form for each path, so every component can use the same one.

The canonical form

```
use Derafu\Support\Url;

Url::normalizePath('/api/index');          // "/api/index"
Url::normalizePath('//api//index/');       // "/api/index"
Url::normalizePath('/api/./index');        // "/api/index"
Url::normalizePath('/api/%69ndex');        // "/api/index"
Url::normalizePath('/ma%c3%b1ana');        // "/ma%C3%B1ana"
Url::normalizePath('');                    // "/"
```

The canonical form:

- starts with `/`, and has no empty segments (`//`), no `.` segments and no `/` at the end (except for the root, `/`);
- has the escapes of the characters that need no escape (letters, numbers and `-`, `.`, `_`, `~`) decoded (`%69` is `i`), and the hexadecimal digits of the other escapes in capitals (`%c3%b1` is `%C3%B1`), as RFC 3986 says in section 6.2.2;
- keeps the case of the letters, and everything else as it is: `%20` stays encoded, and `%25` is not

decoded again, so `%252F` is not a `/`.

What has no safe form

A path that can not be written in a canonical form without changing what it means gives `null`:

The path has	Example
A `..` segment, also escaped	`/a/../b`, `/a/%2e%2e/b`
A separator hidden in an escape, or a backslash	`/a%2Fb`, `/a%5Cb`, `/a\b`
A control character, also escaped, the null byte among them	`/a%00b`, `/a%1Fb`
An escape that is not valid	`/a%zz`, `/a%4`, `/a%`

A separator in an escape is not accepted because a component that decodes it would see more segments than the one that did not. As with `Ip`, a path that has no safe form is a question that has an answer: the functions say `null` or `false`, they do not throw. Whoever receives requests decides what to do with it (refuse it, for example).

Functions

Method	What it does
`Url::normalizePath($path)`	The canonical form of the path, or `null`.
`Url::isNormalizedPath($path)`	Whether the path is its own canonical form (`false` if it has none).
`Url::pathSegments($path)`	The segments of the canonical form (none for the root), or `null`.
`Url::pathStartsWith($path, $prefix, $caseSensitive = true)`	Whether the path is the prefix or is below it, by segments.

```
Url::isNormalizedPath('/api/index');       // true
Url::isNormalizedPath('/api//index');      // false
Url::pathSegments('//api/./%69ndex/');     // ['api', 'index']
```

Prefixes

`pathStartsWith()` compares by **segments**, and both the path and the prefix in their canonical form, so the way they are written does not matter. `/api` is the prefix of `/api`, of `/api/index` and of `/api/index/x`, and it is **not** the prefix of `/apiary`. The root, `/`, is the prefix of every path.

```
Url::pathStartsWith('/api/index', '/api');        // true
Url::pathStartsWith('/api', '/api');              // true
```

```
Url::pathStartsWith('/apiary', '/api');           // false: not a segment boundary.
Url::pathStartsWith('/api//index', '/api/index'); // true: the same path, written
another way.
Url::pathStartsWith('/api/%69ndex', '/api/index'); // true
Url::pathStartsWith('/api/index', '/');           // true
Url::pathStartsWith('/api/../x', '/api');         // false: it has no safe form.
```

It is `false` if the path or the prefix has no safe form: a rule that is not valid does not match anything, so whoever uses it as a rule has to check that the prefix is valid when it is configured.

By default the case matters (`/API` is not `/api`), as it does for most servers. A rule that **protects** a path must not depend on it: a server that reads files from a file system that does not tell them apart (the default one of macOS and Windows) serves `/Academy/x` with the file of `/academy/x`. For that, use `$caseSensitive = false`, which keeps the boundary of the segments:

```
Url::pathStartsWith('/API/Index', '/api', caseSensitive: false);   // true
Url::pathStartsWith('/APIARY', '/api', caseSensitive: false);      // false
```

Last updated on 15/11/2026