

Ip

Ip of Derafu Support, IPv4 and IPv6 addresses and ranges

Anonymous · 08/10/2026

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
<code>Ip::isValid(\$ip)</code>	Whether the text is an IP address.
<code>Ip::version(\$ip)</code>	4, 6 or null.
<code>Ip::isIpv4(\$ip)</code> , <code>Ip::isIpv6(\$ip)</code>	Whether it is of that version (an IPv4 written as IPv6 is IPv6).
<code>Ip::normalize(\$ip)</code>	The canonical form of the address, or null.
<code>Ip::expand(\$ip)</code>	The long form of an IPv6, with its eight groups.
<code>Ip::parse(\$address)</code>	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
<code>Ip::isPrivate(\$ip)</code>	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).
<code>Ip::isReserved(\$ip)</code>	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.
<code>Ip::isLoopback(\$ip)</code>	127.0.0.0/8 and ::1.
<code>Ip::isLinkLocal(\$ip)</code>	169.254.0.0/16 and fe80::/10.
<code>Ip::isPublic(\$ip)</code>	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
<code>Ip::inRange(\$ip, \$range)</code>	Whether the address is in the range.
<code>Ip::inAnyRange(\$ip, \$ranges)</code>	Whether it is in any of the ranges (a list, or any iterable).
<code>Ip::isRange(\$range)</code>	Whether the text is a valid range.
<code>Ip::range(\$range)</code>	The first and the last address of the range.
<code>Ip::cidr(\$ip, \$ipv4Prefix = 32, \$ipv6Prefix = 64)</code>	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 08/10/2026