IPUtil

Ákos Takács

Mar 22, 2023

CONTENTS:

Your are reading the documentation of IPUtil which is a library written in PHP to help you to handle IP addresses and IP ranges. It supports IPv4 and IPv6 addresses and IP ranges for both versions. You can create IP address instances from multiple formats and then convert them to other formats as well. Probably one of the most useful features is generating multiple CIDR notations from any two IP addresses (lowest and highest) so you can use it in a software that requires CIDR notations to restrict access to an interface and regenerate it any time using the library.

CHAPTER

ONE

INTRODUCTION

1.1 The goal of the library

Sometimes you need to work with IP addresses. For example you want to restrict access to a website, but the restriction depends on a list of IP addresses from a database or other sources. Imagine you are a developer in an institution that has a large network with many subnets or individual PCs. You want to make sure the service is available only for coworkers in the subnets of the institution even if the server is accessible publicly. So anyone can manage the addresses in the database and set a minimum and maximum IP addresses of one or many IP ranges and you can provide an api to get CIDR notations or find overlapping IP ranges and fix them.

This is the goal of the library. To make it simple and give you a tool to work with IPv4 and IPv6 addresses.

1.2 Requirements

- PHP 5.6
- Composer

1.3 License

This library is under MIT license

1.4 DEMO

If you want to see some live examples what the library can do for you, check the following page on IT sziget: https://app.it-sziget.hu/en/iputil

CHAPTER

TWO

INSTALLATION

You can install it easily via Composer

```
composer require rimelek/ip-util:1.0.0
```

Make sure you have PHP 5.6 at least. If you wish to test if the library works as intended before using it, read about testing in the section *Unit testing*.

CHAPTER

THREE

UNIT TESTING

3.1 Intro

This library is fully unit-tested and does not have any special dependency, however, you can run unit test if you wish. It is mainly useful when you fork the library and want to improve something.

The source code contains everything you need to run the tests, but it is optimized for using it in PHPStorm with Docker. Using the custom Docker-based PHP interpreter and PHPUnit script you can test the source code with multiple PHP versions and compatible PHPUnit versions.

These scripts are based on the following information:

- Supported PHP versions by PHPUnit
- Supported PHP versions by XDebug

PHP 5.6 is the oldest PHP version supported by the Library, but only because the project started in 2017 without releasing a stable version so I kept the compatibility in the first stable release and I will drop it in the next release.

3.2 Requirements

Installing Docker is required if you want to use the custom PHP interpreters which are based on Docker containers. You can follow the official documentation to install Docker. If you would like to read a short explanation about the variant of Docker check out the Install Docker in my "Learn Docker" tutorial.

3.3 Custom PHP interpreters

- **./php/bin/php.sh**: Wrapper script to run a container for any supported PHP version. It also contains some logic for running PHPUnit with customized configuration files and parameters. The first parameter of the script is the major and minor part of the PHP version and the rest of the parameters are the parameters of the PHP interpreter.

```
./php/bin/php.sh 8.2 --version
```

- `./php/bin/php-<PHP_VERSION>.sh`: Wrapper scripts for `php.sh` configured in PHPStorm as PHP interpreters.
- `./php/bin/php-all.sh`: A special interpreter which runs the command with each custom PHP interpreter. Using this script you can see the test result for each PHP version where the test suite name contains the version number.

- `./php/bin/phpunit.php`: This PHP script runs inside the container of the PHP interpreter and changes some of the parameters which was not handled in `php.sh`. It is responsible for downloading a compatible version of PHPUnit as a PHP Archive (phar).

Then run the following command to test in terminal:

```
./php/bin/php-8.2.sh "$PWD/php/bin/phpunit.php" --configuration phpunit.xml
```

It actually doesn't matter what you pass as configuration file. It is just a placeholder so `phpunit.php` can replace it with the required and compatible configuration file. `phpunit.php` however must be written as is, because the interpreter will compare it with an expected value to add PHPUnit-related arguments.

At the end, every line should be tested. If you are not sure if it happened, use code coverage.

```
./php/bin/php-8.2.sh "$PWD/php/bin/phpunit.php" --configuration phpunit.xml --coverage-
↪clover "$PWD/phpunit.clover"
```

Note: Starting the path of the coverage file with `$PWD` or `$(pwd)` is important, since it will be recognized by the custom php interpreters and mounted into the containers so as a bind mount, the source path must be absolute.

If you want to run coverage test using PHP 5.6, you need to add an extra parameter to enable code coverage.

```
./php/bin/php-5.6.sh -dxdebug.coverage_enable=1 "$PWD/php/bin/phpunit.php" --
↪configuration phpunit.xml --coverage-clover "$PWD/phpunit.coverage.xml"
```

You can also use the PHPStorm GUI to run these tests, although PHP 5.6 unit test is not compatible with the latest PHPStorm. You can see the result in the terminal of the unit test, but code coverage works with PHP 5.6 as well.

Downloading the base image and installing the necessary extensions can be slow. It can be a good idea to run a simple php command before the actual test just for downloading the base images and building the local image:

```
./php/bin/php-all.sh --version
```

This custom interpreters was created to use with PHPStorm. PHPStorm supports to choose Docker container as interpreter, but in that case, you could would need much more manual configurations. These custom interpreters can be set as local interpreters and will give the same output as any PHP interpreter would do.

CHAPTER

FOUR

USAGE AND EXAMPLES

4.1 IP conversions

IP addresses are stored in the objects as binary strings, but there are other formats you can convert to and from. Here are all off them:

- **binary string**: IPv4 addresses are 4, IPv6 addresses are 16 bytes long strings. This is the best format to work with them and the worst to show them. You cannot just echo a binary string. You need to use unpack() for example. You can however create a binary string using hex numbers: "\x7F\x00\x00\x01". It is the binary version of "127.0.0.1"
- **bit string**: It is made of series of 1 and 0.
- **string**: Well, each version is a string, but this is the general representation of an IP address which is for example 127.0.0.1 or ::FFFF:7F00:1.

First we need the autoloader

```
<?php
require_once 'vendor/autoload.php';
```

Let's see how to convert addresses

```
<?php
// ...
use Rimelek\IPUtil\IPv4Address;

// from IPv4 binary string to string
$ip4 = IPv4Address::fromBinary("\x7f\x00\x00\x01");
echo $ip4->toString() . "\n";
// output: 127.0.0.1

// from IPv4 string to bit string
$ip4 = IPv4Address::fromBinary("127.0.0.1");
echo $ip4->toBitString() . "\n";
// output: 01111111000000000000000000000001

$ip4 = IPv4Address::fromBitString("01111111000000000000000000000001");
echo bin2hex($ip4->toBinary()) . "\n";
```

You can do the same with IPv6

```php
<?php
// ..
use Rimelek\IPUtil\IPv6Address;

// from IPv6 binary string to string
$ip6 = IPv4Address::fromBinary("\0\0\0\0\0\0\0\0\0\0\xFF\xFF\x7F\x00\x00\x01");
echo $ip6->toString() . "\n"; // This is an alias of $ipv6->toString(IPv6Address::LENGTH_
↪SHORT);
// output: ::ffff:7f00:1
echo $ip6->toString(IPv6Address::LENGTH_MEDIUM) . "\n";
// output: 0:0:0:0:0:ffff:7f00:1
echo $ip6->toString(IPv6Address::LENGTH_LONG) . "\n";
// output: 0000:0000:0000:0000:0000:ffff:7f00:0001

// from IPv6 string to bit string
$ip6 = IPv6Address::fromBinary("::ffff:7f00:1");
echo $ip6->toBitString() . "\n";
// output:␣
↪00000000000000000000000000000000000000000000000000000000000000000000000000000000000000001111111111111111011111

$ip6 = IPv6Address::fromBitString(
↪"00000000000000000000000000000000000000000000000000000000000000000000000000000000000000001111111111111111011111
↪");
echo bin2hex($ip6->toBinary()) . "\n";
// output: 00000000000000000000ffff7f000001
```

And you can convert an IPv4 address to IPv6 or IPv6 address to IPv4

```php
<?php
// ...
$ip4 = IPv4Address::fromString('192.168.1.1');
$ip6 = $ip4->toIPv6();
echo $ip6->toString() . "\n";
// output: ::ffff:c0a8:101

$ip4 = $ip6->toIPv4();
echo $ip4->toString() . "\n";
// output: ::ffff:c0a8:101
```

or

```php
<?php
// ...
$ip6 = IPv6Address::fromIPv4($ip4);
echo $ip6->toString() . "\n";
// output: ::ffff:c0a8:101

$ip4 = IPv4Address::fromIPv6($ip6);
echo $ip4->toString() . "\n";
// output: ::ffff:c0a8:101
```

CIDR prefix is a number that tells you how many bits are set in the IP mask from left to right followed by zeros only. If you need IP mask, you can create it from CIDR prefix.

```php
<?php
// ...
$ip4Mask = IPv4Address::fromCIDRPrefix(24);
echo $ip4Mask->toString() . "\n";
// output: 255.255.255.0

$ip6Mask = IPv6Address::fromCIDRPrefix(40);
echo $ip6Mask->toString() . "\n";
// output: ffff:ffff:ff00::
```

Not all IPv6 addresses are compatible with IPv4. If you do not want to get an exception when you call $ip6->toIPv4(), use isCompatibleWithIPv4() to check if it is compatible.

```php
<?php
// ...
$ip6 = IPv6Address::fromString('2620:2d:4000:1::16');
echo $ip6->isCompatibleWithIPv4() ? 'Compatible' : 'Incompatible';
echo "\n";
// output: Incompatible

$ip6 = IPv6Address::fromString('::ffff:c0a8:101');
echo $ip6->isCompatibleWithIPv4() ? 'Compatible' : 'Incompatible';
echo "\n";
// output: Compatible
```

You can get some additional information about an IPv4 address like IP class and high order bits

```php
<?php
// ...
$ip4 = IPv4Address::fromString('192.168.1.1');
echo $ip4->getIPClass() . "\n";
// output: C
$ip4 = IPv4Address::fromString('172.17.1.1');
echo $ip4->getIPClass() . "\n";
// output: B
```

The above check based on high order bits. You can get the high order bits of an IP class by calling getHighOrderBitsOfIPv4Class()

```php
<?php
// ...
echo IPv4Address::getHighOrderBitsOfIPv4Class('C') . "\n";
// output: 110
```

4.2 IP ranges

You can create an IP range instance three ways. Directly giving minimum and maximum IP address instances or binary strings or using one IP address instance and a CIDR prefix.

```
<?php
// ...
use Rimelek\IPUtil\IPv4Range;
// ...
$ip4min = IPv4Address::fromString('192.168.1.0');
$ip4max = IPv4Address::fromString('192.168.1.255');

$ip4Range = IPv4Range::fromIPInterval($ip4min, $ip4max);
echo $ip4Range->toString() . "\n";
// output: 192.168.1.0 - 192.168.1.255

$ip4range = IPv4Range::fromBinaryInterval("\xC0\xA8\x01\x00", "\xC0\xA8\x01\xFF");
echo $ip4Range->toString() . "\n";
// output: 192.168.1.0 - 192.168.1.255

$ip4Range = IPv4Range::fromIPWithCIDRPrefix($ip4min, 24);
echo $ip4Range->toString() . "\n";
// output: 192.168.1.0 - 192.168.1.255
```

Using the CIDR notation you do not even need to pass the minimum address. An address between minimum and maximum is appropriate.

```
<?php
// ...
$ip4min = IPv4Address::fromString('192.168.1.18');
$ip4Range = IPv4Range::fromIPWithCIDRPrefix($ip4min, 24);
echo $ip4Range->toString() . "\n";
// output: 192.168.1.0 - 192.168.1.255
```

IPv6Range works the same way with IPv6Address of course.

You can also get the minimum and maximum IP addresses without `toString()`:

```
<?php
// ...
echo $ip4Range->getMinIP()->toString();
echo " - ";
echo $ip4Range->getMaxIP()->toString();
// output: 192.168.1.0 - 192.168.1.255
```

This only for testing and debugging. Do not use it to check if two ranges are equal!

If a range was instantiated by `fromIPWithCIDRPrefix()`, you can get the prefix any time. Otherwise, it will be null.

```
<?php
// ...
echo $ip4Range->getCIDRPrefix() . "\n";
// output: 24
```

You can check if a range is in another range:

```php
<?php
// ...
if ($ip4Range->in($largeIP4Range)) {
    echo 'ip4Range is in largerIp4Range';
}
```

When a range was created by `fromIPInterval()` or `fromBinaryInterval()`, converting it to one CIDR notation is not always possible. Use `toCIDRNotations()` on the instance of the range. This returns an array with new IP range instances. They are all created by `fromIPWithCIDRPrefix()`

```php
<?php
$ip4min = IPv4Address::fromString('192.168.0.3');
$ip4max = IPv4Address::fromString('192.168.2.18');

$ip4Range = IPv4Range::fromIPInterval($ip4min, $ip4max);
foreach ($ip4Range->toCIDRNotations() as $CIDRNotation) {
    echo $CIDRNotation->getMinIP() . '/' . $CIDRNotation->getCIDRPrefix() . "\n";
}
// output:
// 192.168.0.3/32
// 192.168.0.4/30
// 192.168.0.8/29
// 192.168.0.16/28
// 192.168.0.32/27
// 192.168.0.64/26
// 192.168.0.128/25
// 192.168.1.0/24
// 192.168.2.0/28
// 192.168.2.16/31
// 192.168.2.18/32
```