vlucas / **valitron**

Valitron is a simple, elegant, stand-alone validation library with NO dependencies

350 commits	**2** branches	**24** releases	**70** contributors	BSD-3-Clause

Branch: master ▾ | New pull request | Create new file | Upload files | Find file | Clone or download ▾

vlucas Update composer license SPDX — Latest commit `4b32e60` 13 hours ago

lang	Translation to Lithuanian	4 days ago
src/Valitron	Merge pull request #231 from piotr-cz/patch-1	22 hours ago
tests	Merge pull request #237 from mangopeaches/nested-equals-different	4 days ago
.gitattributes	Add .gitattributes	a year ago
.gitignore	Add 'composer test' as a run script	2 years ago
.travis.yml	let's see if we can keep 5.3 still	3 months ago
LICENSE.txt	Working basic validations with tests	5 years ago
README.md	Merge pull request #238 from thinkjson/master	22 hours ago
composer.json	Update composer license SPDX	13 hours ago
phpunit.xml	Add a BaseTest class to reset static properties on setUp and tearDown	5 years ago

README.md

Valitron: Easy Validation That Doesn't Suck

Valitron is a simple, minimal and elegant stand-alone validation library with NO dependencies. Valitron uses simple, straightforward validation methods with a focus on readable and concise syntax. Valitron is the simple and pragmatic validation library you've been looking for.

build passing | stable v1.4.2 | downloads 349.52 k

Why Valitron?

Valitron was created out of frustration with other validation libraries that have dependencies on large components from other frameworks like Symfony's HttpFoundation, pulling in a ton of extra files that aren't really needed for basic validation. It also has purposefully simple syntax used to run all validations in one call instead of individually validating each value by instantiating new classes and validating values one at a time like some other validation libraries require.

In short, Valitron is everything you've been looking for in a validation library but haven't been able to find until now: simple pragmatic syntax, lightweight code that makes sense, extensible for custom callbacks and validations, well tested, and without dependencies. Let's get started.

Installation

Valitron uses Composer to install and update:

```
curl -s http://getcomposer.org/installer | php
php composer.phar require vlucas/valitron
```

The examples below use PHP 5.4 syntax, but Valitron works on PHP 5.3+.

Usage

Usage is simple and straightforward. Just supply an array of data you wish to validate, add some rules, and then call `validate()`. If there are any errors, you can call `errors()` to get them.

```
$v = new Valitron\Validator(array('name' => 'Chester Tester'));
$v->rule('required', 'name');
if($v->validate()) {
    echo "Yay! We're all good!";
} else {
    // Errors
    print_r($v->errors());
}
```

Using this format, you can validate `$_POST` data directly and easily, and can even apply a rule like `required` to an array of fields:

```
$v = new Valitron\Validator($_POST);
$v->rule('required', ['name', 'email']);
$v->rule('email', 'email');
if($v->validate()) {
    echo "Yay! We're all good!";
} else {
    // Errors
    print_r($v->errors());
}
```

You may use dot syntax to access members of multi-dimensional arrays, and an asterisk to validate each member of an array:

```
$v = new Valitron\Validator(array('settings' => array(
    array('threshold' => 50),
    array('threshold' => 90)
)));
$v->rule('max', 'settings.*.threshold', 100);
if($v->validate()) {
    echo "Yay! We're all good!";
} else {
    // Errors
    print_r($v->errors());
}
```

Or use dot syntax to validate all members of a numeric array:

```
$v = new Valitron\Validator(array('values' => array(50, 90)));
$v->rule('max', 'values.*', 100);
if($v->validate()) {
    echo "Yay! We're all good!";
} else {
    // Errors
    print_r($v->errors());
}
```

Setting language and language dir globally:

```

// boot or config file

use Valitron\Validator as V;

V::langDir(__DIR__.'/validator_lang'); // always set langDir before lang.
V::lang('ar');
```

Built-in Validation Rules

- `required` - Field is required
- `equals` - Field must match another field (email/password confirmation)
- `different` - Field must be different than another field
- `accepted` - Checkbox or Radio must be accepted (yes, on, 1, true)
- `numeric` - Must be numeric
- `integer` - Must be integer number

- `boolean` - Must be boolean
- `array` - Must be array
- `length` - String must be certain length
- `lengthBetween` - String must be between given lengths
- `lengthMin` - String must be greater than given length
- `lengthMax` - String must be less than given length
- `min` - Minimum
- `max` - Maximum
- `in` - Performs in_array check on given array values
- `notIn` - Negation of `in` rule (not in array of values)
- `ip` - Valid IP address
- `email` - Valid email address
- `emailDNS` - Valid email address with active DNS record
- `url` - Valid URL
- `urlActive` - Valid URL with active DNS record
- `alpha` - Alphabetic characters only
- `alphaNum` - Alphabetic and numeric characters only
- `slug` - URL slug characters (a-z, 0-9, -, _)
- `regex` - Field matches given regex pattern
- `date` - Field is a valid date
- `dateFormat` - Field is a valid date in the given format
- `dateBefore` - Field is a valid date and is before the given date
- `dateAfter` - Field is a valid date and is after the given date
- `contains` - Field is a string and contains the given string
- `creditCard` - Field is a valid credit card number
- `instanceOf` - Field contains an instance of the given class
- `optional` - Value does not need to be included in data array. If it is however, it must pass validation.

NOTE: If you are comparing floating-point numbers with min/max validators, you should install the BCMath extension for greater accuracy and reliability. The extension is not required for Valitron to work, but Valitron will use it if available, and it is highly recommended.

Required fields

the `required` rule checks if a field exists in the data array, and is not null or an empty string.

```
$v->rule('required', 'field_name');
```

Using an extra parameter, you can make this rule more flexible, and only check if the field exists in the data array.

```
$v->rule('required', 'field_name', true);
```

Credit Card Validation usage

Credit card validation currently allows you to validate a Visa `visa`, Mastercard `mastercard`, Dinersclub `dinersclub`, American Express `amex` or Discover `discover`

This will check the credit card against each card type

```
$v->rule('creditCard', 'credit_card');
```

To optionally filter card types, add the slug to an array as the next parameter:

```
$v->rule('creditCard', 'credit_card', ['visa', 'mastercard']);
```

If you only want to validate one type of card, put it as a string:

```
$v->rule('creditCard', 'credit_card', 'visa');
```

If the card type information is coming from the client, you might also want to still specify an array of valid card types:

```
$cardType = 'amex';
$v->rule('creditCard', 'credit_card', $cardType, ['visa', 'mastercard']);
$v->validate(); // false
```

Adding Custom Validation Rules

To add your own validation rule, use the `addRule` method with a rule name, a custom callback or closure, and a error message to display in case of an error. The callback provided should return boolean true or false.

```
Valitron\Validator::addRule('alwaysFail', function($field, $value, array $params, array $fields) {
    return false;
}, 'Everything you do is wrong. You fail.');
```

You can also use one-off rules that are only valid for the specified fields.

```
$v = new Valitron\Validator(array("foo" => "bar"));
$v->rule(function($field, $value, $params, $fields) {
    return true;
}, "foo")->message("{field} failed...");
```

This is useful because such rules can have access to variables defined in the scope where the `Validator` lives. The Closure's signature is identical to `Validator::addRule` callback's signature.

If you wish to add your own rules that are not static (i.e., your rule is not static and available to call `Validator` instances), you need to use `Validator::addInstanceRule`. This rule will take the same parameters as `Validator::addRule` but it has to be called on a `Validator` instance.

Alternate syntax for adding rules

As the number of rules grows, you may prefer the alternate syntax for defining multiple rules at once.

```
$rules = [
    'required' => 'foo',
    'accepted' => 'bar',
    'integer' =>  'bar'
];

$v = new Valitron\Validator(array('foo' => 'bar', 'bar' => 1));
$v->rules($rules);
$v->validate();
```

If your rule requires multiple parameters or a single parameter more complex than a string, you need to wrap the rule in an array.

```
$rules = [
    'required' => [
        ['foo'],
        ['bar']
    ],
    'length' => [
        ['foo', 3]
    ]
];
```

You can also specify multiple rules for each rule type.

```
$rules = [
    'length'   => [
        ['foo', 5],
        ['bar', 5]
    ]
];
```

Putting these techniques together, you can create a complete rule definition in a relatively compact data structure.

You can continue to add individual rules with the `rule` method even after specifying a rule definition via an array. This is especially useful if you are defining custom validation rules.

```
$rules = [
    'required' => 'foo',
    'accepted' => 'bar',
    'integer' =>  'bar'
];

$v = new Valitron\Validator(array('foo' => 'bar', 'bar' => 1));
$v->rules($rules);
$v->rule('min', 'bar', 0);
$v->validate();
```

You can also add rules on a per-field basis:

```
$rules = [
    'required',
    ['lengthMin', 4]
];

$v = new Valitron\Validator(array('foo' => 'bar'));
$v->mapFieldRules('foo', $rules);
$v->validate();
```

Or for multiple fields at once:

```
$rules = [
    'foo' => ['required', 'integer'],
    'bar'=>['email', ['lengthMin', 4]]
];

$v = new Valitron\Validator(array('foo' => 'bar', 'bar' => 'mail@example.com));
$v->mapFieldsRules($rules);
$v->validate();
```

Adding field label to messages

You can do this in two different ways, you can add a individual label to a rule or an array of all labels for the rules.

To add individual label to rule you simply add the `label` method after the rule.

```
$v = new Valitron\Validator(array());
$v->rule('required', 'name')->message('{field} is required')->label('Name');
$v->validate();
```

There is a edge case to this method, you wouldn't be able to use a array of field names in the rule definition, so one rule per field. So this wouldn't work:

```
$v = new Valitron\Validator(array());
$v->rule('required', array('name', 'email'))->message('{field} is required')->label('Name');
$v->validate();
```

However we can use a array of labels to solve this issue by simply adding the `labels` method instead:

```php
$v = new Valitron\Validator(array());
$v->rule('required', array('name', 'email'))->message('{field} is required');
$v->labels(array(
    'name' => 'Name',
    'email' => 'Email address'
));
$v->validate();
```

This introduces a new set of tags to your error language file which looks like `{field}`, if you are using a rule like `equals` you can access the second value in the language file by incrementing the field with a value like `{field1}`.

Re-use of validation rules

You can re-use your validation rules to quickly validate different data with the same rules by using the withData method:

```php
$v = new Valitron\Validator(array());
$v->rule('required', 'name')->message('{field} is required');
$v->validate(); //false

$v2 = $v->withData(array('name'=>'example'));
$v2->validate(); //true
```

Running Tests

The test suite depends on the Composer autoloader to load and run the Valitron files. Please ensure you have downloaded and installed Composer before running the tests:

1. Download Composer `curl -s http://getcomposer.org/installer | php`
2. Run 'install' `php composer.phar install`
3. Run the tests `phpunit`

Contributing

1. Fork it
2. Create your feature branch (`git checkout -b my-new-feature`)
3. Make your changes
4. Run the tests, adding new ones for your own code if necessary (`phpunit`)
5. Commit your changes (`git commit -am 'Added some feature'`)
6. Push to the branch (`git push origin my-new-feature`)
7. Create new Pull Request
8. Pat yourself on the back for being so awesome