Validation

Introduction

Laravel provides several different approaches to validate your application's incoming data. By default, Laravel's base controller class uses a `ValidatesRequests` trait which provides a convenient method to validate incoming HTTP request with a variety of powerful validation rules.

Validation Quickstart

To learn about Laravel's powerful validation features, let's look at a complete example of validating a form and displaying the error messages back to the user.

Defining The Routes

First, let's assume we have the following routes defined in our `routes/web.php` file:

```
Route::get('post/create', 'PostController@create');

Route::post('post', 'PostController@store');
```

The `GET` route will display a form for the user to create a new blog post, while the `POST` route will store the new blog post in the database.

Creating The Controller

Next, let's take a look at a simple controller that handles these routes. We'll leave the `store` method empty for now:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;

class PostController extends Controller
{
    /**
     * Show the form to create a new blog post.
     *
     * @return Response
     */
    public function create()
    {
        return view('post.create');
    }
```

```
    /**
     * Store a new blog post.
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        // Validate and store the blog post...
    }
}
```

Writing The Validation Logic

Now we are ready to fill in our `store` method with the logic to validate the new blog post. To do this, we will use the `validate` method provided by the `Illuminate\Http\Request` object. If the validation rules pass, your code will keep executing normally; however, if validation fails, an exception will be thrown and the proper error response will automatically be sent back to the user. In the case of a traditional HTTP request, a redirect response will be generated, while a JSON response will be sent for AJAX requests.

To get a better understanding of the `validate` method, let's jump back into the `store` method:

```
/**
 * Store a new blog post.
 *
 * @param  Request  $request
 * @return Response
 */
public function store(Request $request)
{
    $validatedData = $request->validate([
        'title' => 'required|unique:posts|max:255',
        'body' => 'required',
    ]);

    // The blog post is valid...
}
```

As you can see, we pass the desired validation rules into the `validate` method. Again, if the validation fails, the proper response will automatically be generated. If the validation passes, our controller will continue executing normally.

Alternatively, validation rules may be specified as arrays of rules instead of a single `|` delimited string:

```
$validatedData = $request->validate([
    'title' => ['required', 'unique:posts', 'max:255'],
    'body' => ['required'],
]);
```

Stopping On First Validation Failure

Sometimes you may wish to stop running validation rules on an attribute after the first validation failure. To do so, assign the `bail` rule to the attribute:

```
$request->validate([
    'title' => 'bail|required|unique:posts|max:255',
    'body' => 'required',
]);
```

In this example, if the `unique` rule on the `title` attribute fails, the `max` rule will not be checked. Rules will be validated in the order they are assigned.

A Note On Nested Attributes

If your HTTP request contains "nested" parameters, you may specify them in your validation rules using "dot" syntax:

```
$request->validate([
    'title' => 'required|unique:posts|max:255',
    'author.name' => 'required',
    'author.description' => 'required',
]);
```

Displaying The Validation Errors

So, what if the incoming request parameters do not pass the given validation rules? As mentioned previously, Laravel will automatically redirect the user back to their

previous location. In addition, all of the validation errors will automatically be flashed to the session.

Again, notice that we did not have to explicitly bind the error messages to the view in our `GET` route. This is because Laravel will check for errors in the session data, and automatically bind them to the view if they are available. The `$errors` variable will be an instance of `Illuminate\Support\MessageBag`. For more information on working with this object, check out its documentation.

The `$errors` variable is bound to the view by the `Illuminate\View\Middleware\ShareErrorsFromSession` middleware, which is provided by the `web` middleware group. **When this middleware is applied an `$errors` variable will always be available in your views**, allowing you to conveniently assume the `$errors` variable is always defined and can be safely used.

So, in our example, the user will be redirected to our controller's `create` method when validation fails, allowing us to display the error messages in the view:

```
<!-- /resources/views/post/create.blade.php -->

<h1>Create Post</h1>

@if ($errors->any())
    <div class="alt alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif

<!-- Create Post Form -->
```

The `@error` Directive

You may also use the `@error` Blade directive to quickly check if validation error messages exist for a given attribute. Within an `@error` directive, you may echo the `$message` variable to display the error message:

```
<!-- /resources/views/post/create.blade.php -->

<label for="title">Post Title</label>

<input id="title" type="text" class="@error('title') is-invalid @enderror">

@error('title')
    <div class="alt alert-danger">{{ $message }}</div>
@enderror
```

A Note On Optional Fields

By default, Laravel includes the `TrimStrings` and `ConvertEmptyStringsToNull` middleware in your application's global middleware stack. These middleware are listed in the stack by the `App\Http\Kernel` class. Because of this, you will often need to mark your "optional" request fields as `nullable` if you do not want the validator to consider `null` values as invalid. For example:

```
$request->validate([
    'title' => 'required|unique:posts|max:255',
    'body' => 'required',
    'publish_at' => 'nullable|date',
]);
```

In this example, we are specifying that the `publish_at` field may be either `null` or a valid date representation. If the `nullable` modifier is not added to the rule definition, the validator would consider `null` an invalid date.

AJAX Requests & Validation

In this example, we used a traditional form to send data to the application. However, many applications use AJAX requests. When using the `validate` method during an AJAX request, Laravel will not generate a redirect response. Instead, Laravel generates a JSON response containing all of the validation errors. This JSON response will be sent with a 422 HTTP status code.

Form Request Validation

Creating Form Requests

For more complex validation scenarios, you may wish to create a "form request". Form requests are custom request classes that contain validation logic. To create a form request class, use the `make:request` Artisan CLI command:

```
php artisan make:request StoreBlogPost
```

The generated class will be placed in the `app/Http/Requests` directory. If this directory does not exist, it will be created when you run the `make:request` command. Let's add a few validation rules to the `rules` method:

```
/**
 * Get the validation rules that apply to the request.
 *
 * @return array
 */
public function rules()
{
    return [
        'title' => 'required|unique:posts|max:255',
        'body' => 'required',
    ];
}
```

You may type-hint any dependencies you need within the `rules` method's signature. They will automatically be resolved via the Laravel service container.

So, how are the validation rules evaluated? All you need to do is type-hint the request on your controller method. The incoming form request is validated before the controller method is called, meaning you do not need to clutter your controller with any validation logic:

```
/**
 * Store the incoming blog post.
 *
 * @param  StoreBlogPost  $request
 * @return Response
 */
public function store(StoreBlogPost $request)
{
    // The incoming request is valid...

    // Retrieve the validated input data...
    $validated = $request->validated();
}
```

If validation fails, a redirect response will be generated to send the user back to their previous location. The errors will also be flashed to the session so they are available for display. If the request was an AJAX request, a HTTP response with a 422 status code will be returned to the user including a JSON representation of the validation errors.

Adding After Hooks To Form Requests

If you would like to add an "after" hook to a form request, you may use the `withValidator` method. This method receives the fully constructed validator, allowing you to call any of its methods before the validation rules are actually evaluated:

```
/**
 * Configure the validator instance.
 *
 * @param  \Illuminate\Validation\Validator  $validator
 * @return void
 */
public function withValidator($validator)
{
    $validator->after(function ($validator) {
        if ($this->somethingElseIsInvalid()) {
            $validator->errors()->add('field', 'Something is wrong with this field
        }
    });
}
```

Authorizing Form Requests

The form request class also contains an `authorize` method. Within this method, you may check if the authenticated user actually has the authority to update a given resource. For example, you may determine if a user actually owns a blog comment they are attempting to update:

```
/**
 * Determine if the user is authorized to make this request.
 *
 * @return bool
 */
public function authorize()
{
    $comment = Comment::find($this->route('comment'));

    return $comment && $this->user()->can('update', $comment);
}
```

Since all form requests extend the base Laravel request class, we may use the `user` method to access the currently authenticated user. Also note the call to the `route` method in the example above. This method grants you access to the URI parameters defined on the route being called, such as the `{comment}` parameter in the example below:

```
Route::post('comment/{comment}');
```

If the `authorize` method returns `false`, a HTTP response with a 403 status code will automatically be returned and your controller method will not execute.

If you plan to have authorization logic in another part of your application, return `true` from the `authorize` method:

```
/**
 * Determine if the user is authorized to make this request.
 *
 * @return bool
 */
public function authorize()
{
    return true;
}
```

You may type-hint any dependencies you need within the `authorize` method's signature. They will automatically be resolved via the Laravel service container.

Customizing The Error Messages

You may customize the error messages used by the form request by overriding the `messages` method. This method should return an array of attribute / rule pairs and their corresponding error messages:

```
/**
 * Get the error messages for the defined validation rules.
 *
 * @return array
 */
public function messages()
{
    return [
        'title.required' => 'A title is required',
        'body.required'  => 'A message is required',
    ];
}
```

Customizing The Validation Attributes

If you would like the `:attribute` portion of your validation message to be replaced with a custom attribute name, you may specify the custom names by overriding the `attributes` method. This method should return an array of attribute / name pairs:

```
/**
 * Get custom attributes for validator errors.
 *
 * @return array
 */
public function attributes()
{
```

```
    return [
        'email' => 'email address',
    ];
}
```

Manually Creating Validators

If you do not want to use the `validate` method on the request, you may create a validator instance manually using the `Validator` facade. The `make` method on the facade generates a new validator instance:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Validator;

class PostController extends Controller
{
    /**
     * Store a new blog post.
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        $validator = Validator::make($request->all(), [
            'title' => 'required|unique:posts|max:255',
            'body' => 'required',
        ]);

        if ($validator->fails()) {
            return redirect('post/create')
                        ->withErrors($validator)
                        ->withInput();
        }

        // Store the blog post...
    }
}
```

The first argument passed to the `make` method is the data under validation. The second argument is the validation rules that should be applied to the data.

After checking if the request validation failed, you may use the `withErrors` method to flash the error messages to the session. When using this method, the `$errors` variable will automatically be shared with your views after redirection, allowing you to easily display them back to the user. The `withErrors` method accepts a validator, a `MessageBag`, or a PHP `array`.

Automatic Redirection

If you would like to create a validator instance manually but still take advantage of the automatic redirection offered by the request's `validate` method, you may call the `validate` method on an existing validator instance. If validation fails, the user will automatically be redirected or, in the case of an AJAX request, a JSON response will be returned:

```
Validator::make($request->all(), [
    'title' => 'required|unique:posts|max:255',
    'body' => 'required',
])->validate();
```

Named Error Bags

If you have multiple forms on a single page, you may wish to name the `MessageBag` of errors, allowing you to retrieve the error messages for a specific form. Pass a name as the second argument to `withErrors`:

```
return redirect('register')
            ->withErrors($validator, 'login');
```

You may then access the named `MessageBag` instance from the `$errors` variable:

```
{{ $errors->login->first('email') }}
```

After Validation Hook

The validator also allows you to attach callbacks to be run after validation is completed. This allows you to easily perform further validation and even add more error messages to the message collection. To get started, use the `after` method on a validator instance:

```
$validator = Validator::make(...);

$validator->after(function ($validator) {
    if ($this->somethingElseIsInvalid()) {
        $validator->errors()->add('field', 'Something is wrong with this field!');
    }
});

if ($validator->fails()) {
    //
}
```

Working With Error Messages

After calling the `errors` method on a `Validator` instance, you will receive an `Illuminate\Support\MessageBag` instance, which has a variety of convenient methods for working with error messages. The `$errors` variable that is automatically made available to all views is also an instance of the `MessageBag` class.

Retrieving The First Error Message For A Field

To retrieve the first error message for a given field, use the `first` method:

```
$errors = $validator->errors();

echo $errors->first('email');
```

Retrieving All Error Messages For A Field

If you need to retrieve an array of all the messages for a given field, use the `get` method:

```
foreach ($errors->get('email') as $message) {
    //
}
```

If you are validating an array form field, you may retrieve all of the messages for each of the array elements using the `*` character:

```
foreach ($errors->get('attachments.*') as $message) {
    //
}
```

Retrieving All Error Messages For All Fields

To retrieve an array of all messages for all fields, use the `all` method:

```
foreach ($errors->all() as $message) {
    //
}
```

Determining If Messages Exist For A Field

The `has` method may be used to determine if any error messages exist for a given field:

```
if ($errors->has('email')) {
    //
}
```

Custom Error Messages

If needed, you may use custom error messages for validation instead of the defaults. There are several ways to specify custom messages. First, you may pass the custom messages as the third argument to the `Validator::make` method:

```
$messages = [
    'required' => 'The :attribute field is required.',
];

$validator = Validator::make($input, $rules, $messages);
```

In this example, the `:attribute` placeholder will be replaced by the actual name of the field under validation. You may also utilize other placeholders in validation messages. For example:

```
$messages = [
    'same'    => 'The :attribute and :other must match.',
    'size'    => 'The :attribute must be exactly :size.',
    'between' => 'The :attribute value :input is not between :min - :max.',
    'in'      => 'The :attribute must be one of the following types: :values',
];
```

Specifying A Custom Message For A Given Attribute

Sometimes you may wish to specify a custom error message only for a specific field. You may do so using "dot" notation. Specify the attribute's name first, followed by the rule:

```
$messages = [
    'email.required' => 'We need to know your e-mail address!',
];
```

Specifying Custom Messages In Language Files

In most cases, you will probably specify your custom messages in a language file instead of passing them directly to the `Validator`. To do so, add your messages to `custom` array in the `resources/lang/xx/validation.php` language file.

```
'custom' => [
    'email' => [
        'required' => 'We need to know your e-mail address!',
    ],
],
```

Specifying Custom Attributes In Language Files

If you would like the `:attribute` portion of your validation message to be replaced with a custom attribute name, you may specify the custom name in the `attributes` array of your `resources/lang/xx/validation.php` language file:

```
'attributes' => [
    'email' => 'email address',
],
```

Specifying Custom Values In Language Files

Sometimes you may need the `:value` portion of your validation message to be replaced with a custom representation of the value. For example, consider the following rule that specifies that a credit card number is required if the `payment_type` has a value of `cc`:

```
$request->validate([
    'credit_card_number' => 'required_if:payment_type,cc'
]);
```

If this validation rule fails, it will produce the following error message:

```
The credit card number field is required when payment type is cc.
```

Instead of displaying `cc` as the payment type value, you may specify a custom value representation in your `validation` language file by defining a `values` array:

```
'values' => [
    'payment_type' => [
        'cc' => 'credit card'
    ],
],
```

Now if the validation rule fails it will produce the following message:

```
The credit card number field is required when payment type is credit card.
```

Available Validation Rules

Below is a list of all available validation rules and their function:

accepted

The field under validation must be *yes*, *on*, *1*, or *true*. This is useful for validating "Terms of Service" acceptance.

active_url

The field under validation must have a valid A or AAAA record according to the `dns_get_record` PHP function.

after:*date*

The field under validation must be a value after a given date. The dates will be passed into the `strtotime` PHP function:

```
'start_date' => 'required|date|after:tomorrow'
```

Instead of passing a date string to be evaluated by `strtotime`, you may specify another field to compare against the date:

```
'finish_date' => 'required|date|after:start_date'
```

after_or_equal:*date*

The field under validation must be a value after or equal to the given date. For more information, see the after rule.

alpha

The field under validation must be entirely alphabetic characters.

alpha_dash

The field under validation may have alpha-numeric characters, as well as dashes and underscores.

alpha_num

The field under validation must be entirely alpha-numeric characters.

array

The field under validation must be a PHP `array`.

bail

Stop running validation rules after the first validation failure.

before:*date*

The field under validation must be a value preceding the given date. The dates will be passed into the PHP `strtotime` function. In addition, like the `after` rule, the name of another field under validation may be supplied as the value of `date`.

before_or_equal:*date*

The field under validation must be a value preceding or equal to the given date. The dates will be passed into the PHP `strtotime` function. In addition, like the `after` rule, the name of another field under validation may be supplied as the value of `date`.

between:*min*,*max*

The field under validation must have a size between the given *min* and *max*. Strings, numerics, arrays, and files are evaluated in the same fashion as the `size` rule.

boolean

The field under validation must be able to be cast as a boolean. Accepted input are `true`, `false`, `1`, `0`, `"1"`, and `"0"`.

confirmed

The field under validation must have a matching field of `foo_confirmation`. For example, if the field under validation is `password`, a matching `password_confirmation` field must be present in the input.

date

The field under validation must be a valid, non-relative date according to the `strtotime` PHP function.

date_equals:*date*

The field under validation must be equal to the given date. The dates will be passed into the PHP `strtotime` function.

date_format:*format*

The field under validation must match the given *format*. You should use **either** `date` or `date_format` when validating a field, not both. This validation rule supports all formats supported by PHP's DateTime class.

different:*field*

The field under validation must have a different value than *field*.

digits:*value*

The field under validation must be *numeric* and must have an exact length of *value*.

digits_between:*min*,*max*

The field under validation must have a length between the given *min* and *max*.

dimensions

The file under validation must be an image meeting the dimension constraints as specified by the rule's parameters:

```
'avatar' => 'dimensions:min_width=100,min_height=200'
```

Available constraints are: *min_width*, *max_width*, *min_height*, *max_height*, *width*, *height*, *ratio*.

A *ratio* constraint should be represented as width divided by height. This can be specified either by a statement like `3/2` or a float like `1.5`:

```
'avatar' => 'dimensions:ratio=3/2'
```

Since this rule requires several arguments, you may use the `Rule::dimensions` method to fluently construct the rule:

```
use Illuminate\Validation\Rule;

Validator::make($data, [
    'avatar' => [
        'required',
        Rule::dimensions()->maxWidth(1000)->maxHeight(500)->ratio(3 / 2),
    ],
]);
```

distinct

When working with arrays, the field under validation must not have any duplicate values.

```
'foo.*.id' => 'distinct'
```

email

The field under validation must be formatted as an e-mail address. Under the hood,

this validation rule makes use of the `egulias/email-validator` package for validating the email address. By default the `RFCValidation` validator is applied, but you can apply other validation styles as well:

```
'email' => 'email:rfc,dns'
```

The example above will apply the `RFCValidation` and `DNSCheckValidation` validations. Here's a full list of validation styles you can apply:

- `rfc`: `RFCValidation`
- `strict`: `NoRFCWarningsValidation`
- `dns`: `DNSCheckValidation`
- `spoof`: `SpoofCheckValidation`
- `filter`: `FilterEmailValidation`

The `filter` validator, which uses PHP's `filter_var` function under the hood, ships with Laravel and is Laravel's pre-5.8 behavior. The `dns` and `spoof` validators require the PHP `intl` extension.

ends_with:*foo,bar,...*

The field under validation must end with one of the given values.

exists:*table,column*

The field under validation must exist on a given database table.

Basic Usage Of Exists Rule

```
'state' => 'exists:states'
```

If the `column` option is not specified, the field name will be used.

Specifying A Custom Column Name

```
'state' => 'exists:states,abbreviation'
```

Occasionally, you may need to specify a specific database connection to be used for the `exists` query. You can accomplish this by prepending the connection name to the table name using "dot" syntax:

```
'email' => 'exists:connection.staff,email'
```

If you would like to customize the query executed by the validation rule, you may use the `Rule` class to fluently define the rule. In this example, we'll also specify the validation rules as an array instead of using the `|` character to delimit them:

```
use Illuminate\Validation\Rule;

Validator::make($data, [
    'email' => [
        'required',
        Rule::exists('staff')->where(function ($query) {
            $query->where('account_id', 1);
        }),
    ],
]);
```

file

The field under validation must be a successfully uploaded file.

filled

The field under validation must not be empty when it is present.

gt:*field*

The field under validation must be greater than the given *field*. The two fields must be of the same type. Strings, numerics, arrays, and files are evaluated using the same conventions as the `size` rule.

gte:*field*

The field under validation must be greater than or equal to the given *field*. The two fields must be of the same type. Strings, numerics, arrays, and files are evaluated using the same conventions as the `size` rule.

image

The file under validation must be an image (jpeg, png, bmp, gif, svg, or webp)

in:*foo,bar,...*

The field under validation must be included in the given list of values. Since this rule often requires you to `implode` an array, the `Rule::in` method may be used to fluently construct the rule:

```
use Illuminate\Validation\Rule;

Validator::make($data, [
    'zones' => [
        'required',
        Rule::in(['first-zone', 'second-zone']),
    ],
]);
```

in_array:*anotherfield*.*

The field under validation must exist in *anotherfield*'s values.

integer

The field under validation must be an integer.

> This validation rule does not verify that the input is of the "integer" variable type, only that the input is a string or numeric value that contains an integer.

ip

The field under validation must be an IP address.

ipv4

The field under validation must be an IPv4 address.

ipv6

The field under validation must be an IPv6 address.

json

The field under validation must be a valid JSON string.

lt:*field*

The field under validation must be less than the given *field*. The two fields must be of the same type. Strings, numerics, arrays, and files are evaluated using the same conventions as the `size` rule.

lte:*field*

The field under validation must be less than or equal to the given *field*. The two fields must be of the same type. Strings, numerics, arrays, and files are evaluated using the same conventions as the `size` rule.

max:*value*

The field under validation must be less than or equal to a maximum *value*. Strings, numerics, arrays, and files are evaluated in the same fashion as the `size` rule.

mimetypes:*text/plain,...*

The file under validation must match one of the given MIME types:

```
'video' => 'mimetypes:video/avi,video/mpeg,video/quicktime'
```

To determine the MIME type of the uploaded file, the file's contents will be read and the framework will attempt to guess the MIME type, which may be different from the client provided MIME type.

mimes:*foo,bar,...*

The file under validation must have a MIME type corresponding to one of the listed extensions.

Basic Usage Of MIME Rule

```
'photo' => 'mimes:jpeg,bmp,png'
```

Even though you only need to specify the extensions, this rule actually validates against the MIME type of the file by reading the file's contents and guessing its MIME type.

A full listing of MIME types and their corresponding extensions may be found at the following location: https://svn.apache.org/repos/asf/httpd/httpd/trunk/docs/conf/mime.types

min:*value*

The field under validation must have a minimum *value*. Strings, numerics, arrays, and files are evaluated in the same fashion as the `size` rule.

not_in:*foo,bar,...*

The field under validation must not be included in the given list of values. The `Rule::notIn` method may be used to fluently construct the rule:

```
use Illuminate\Validation\Rule;

Validator::make($data, [
    'toppings' => [
        'required',
        Rule::notIn(['sprinkles', 'cherries']),
    ],
]);
```

not_regex:*pattern*

The field under validation must not match the given regular expression.

Internally, this rule uses the PHP `preg_match` function. The pattern specified should obey the same formatting required by `preg_match` and thus also include valid delimiters. For example: `'email' => 'not_regex:/^.+$/i'`.

Note: When using the `regex` / `not_regex` patterns, it may be necessary to specify rules in an array instead of using pipe delimiters, especially if the regular expression contains a pipe character.

nullable

The field under validation may be `null`. This is particularly useful when validating primitive such as strings and integers that can contain `null` values.

numeric

The field under validation must be numeric.

password

The field under validation must match the authenticated user's password. You may specify an authentication guard using the rule's first parameter:

```
'password' => 'password:api'
```

present

The field under validation must be present in the input data but can be empty.

regex:*pattern*

The field under validation must match the given regular expression.

Internally, this rule uses the PHP `preg_match` function. The pattern specified should obey the same formatting required by `preg_match` and thus also include valid delimiters. For example: `'email' => 'regex:/^.+@.+$/i'`.

Note: When using the `regex` / `not_regex` patterns, it may be necessary to specify rules in an array instead of using pipe delimiters, especially if the regular expression contains a pipe character.

required

The field under validation must be present in the input data and not empty. A field is considered "empty" if one of the following conditions are true:

- The value is `null`.
- The value is an empty string.

- The value is an empty array or empty `Countable` object.
- The value is an uploaded file with no path.

required_if:*anotherfield*,*value*,...

The field under validation must be present and not empty if the *anotherfield* field is equal to any *value*.

If you would like to construct a more complex condition for the `required_if` rule, you may use the `Rule::requiredIf` method. This methods accepts a boolean or a Closure. When passed a Closure, the Closure should return `true` or `false` to indicate if the field under validation is required:

```
use Illuminate\Validation\Rule;

Validator::make($request->all(), [
    'role_id' => Rule::requiredIf($request->user()->is_admin),
]);

Validator::make($request->all(), [
    'role_id' => Rule::requiredIf(function () use ($request) {
        return $request->user()->is_admin;
    }),
]);
```

required_unless:*anotherfield*,*value*,...

The field under validation must be present and not empty unless the *anotherfield* field is equal to any *value*.

required_with:*foo*,*bar*,...

The field under validation must be present and not empty *only if* any of the other specified fields are present.

required_with_all:*foo*,*bar*,...

The field under validation must be present and not empty *only if* all of the other specified fields are present.

required_without:*foo*,*bar*,...

The field under validation must be present and not empty *only when* any of the other specified fields are not present.

required_without_all:*foo*,*bar*,...

The field under validation must be present and not empty *only when* all of the other specified fields are not present.

same:*field*

The given *field* must match the field under validation.

size:*value*

The field under validation must have a size matching the given *value*. For string data, *value* corresponds to the number of characters. For numeric data, *value* corresponds to a given integer value. For an array, *size* corresponds to the `count` of the array. For files, *size* corresponds to the file size in kilobytes.

starts_with:*foo*,*bar*,...

The field under validation must start with one of the given values.

string

The field under validation must be a string. If you would like to allow the field to also be `null`, you should assign the `nullable` rule to the field.

timezone

The field under validation must be a valid timezone identifier according to the `timezone_identifiers_list` PHP function.

unique:*table*,*column*,*except*,*idColumn*

The field under validation must not exist within the given database table.

Specifying A Custom Column Name:

The `column` option may be used to specify the field's corresponding database column. If the `column` option is not specified, the field name will be used.

```
'email' => 'unique:users,email_address'
```

Custom Database Connection

Occasionally, you may need to set a custom connection for database queries made by the Validator. As seen above, setting `unique:users` as a validation rule will use the default database connection to query the database. To override this, specify the connection and the table name using "dot" syntax:

```
'email' => 'unique:connection.users,email_address'
```

Forcing A Unique Rule To Ignore A Given ID:

Sometimes, you may wish to ignore a given ID during the unique check. For example, consider an "update profile" screen that includes the user's name, e-mail address, and location. You will probably want to verify that the e-mail address is unique. However, if the user only changes the name field and not the e-mail field, you do not want a validation error to be thrown because the user is already the owner of the e-mail address.

To instruct the validator to ignore the user's ID, we'll use the `Rule` class to fluently define the rule. In this example, we'll also specify the validation rules as an array instead of using the `|` character to delimit the rules:

```
use Illuminate\Validation\Rule;

Validator::make($data, [
    'email' => [
        'required',
        Rule::unique('users')->ignore($user->id),
    ],
]);
```

You should never pass any user controlled request input into the `ignore` method. Instead, you should only pass a system generated unique ID such as an auto-incrementing ID or UUID from an Eloquent model instance. Otherwise, your application will be vulnerable to an SQL injection attack.

Instead of passing the model key's value to the `ignore` method, you may pass the entire model instance. Laravel will automatically extract the key from the model:

```
Rule::unique('users')->ignore($user)
```

If your table uses a primary key column name other than `id`, you may specify the name of the column when calling the `ignore` method:

```
Rule::unique('users')->ignore($user->id, 'user_id')
```

By default, the `unique` rule will check the uniqueness of the column matching the name of the attribute being validated. However, you may pass a different column name as the second argument to the `unique` method:

```
Rule::unique('users', 'email_address')->ignore($user->id),
```

Adding Additional Where Clauses:

You may also specify additional query constraints by customizing the query using the `where` method. For example, let's add a constraint that verifies the `account_id` is `1`:

```
'email' => Rule::unique('users')->where(function ($query) {
    return $query->where('account_id', 1);
})
```

url

The field under validation must be a valid URL.

uuid

The field under validation must be a valid RFC 4122 (version 1, 3, 4, or 5) universally unique identifier (UUID).

Conditionally Adding Rules

Validating When Present

In some situations, you may wish to run validation checks against a field **only** if that field is present in the input array. To quickly accomplish this, add the `sometimes` rule to your rule list:

```
$v = Validator::make($data, [
    'email' => 'sometimes|required|email',
]);
```

In the example above, the `email` field will only be validated if it is present in the `$data` array.

If you are attempting to validate a field that should always be present but may be empty, check out this note on optional fields

Complex Conditional Validation

Sometimes you may wish to add validation rules based on more complex conditional logic. For example, you may wish to require a given field only if another field has a greater value than 100. Or, you may need two fields to have a given value only when another field is present. Adding these validation rules doesn't have to be a pain. First, create a `Validator` instance with your *static rules* that never change:

```
$v = Validator::make($data, [
    'email' => 'required|email',
    'games' => 'required|numeric',
]);
```

Let's assume our web application is for game collectors. If a game collector registers with our application and they own more than 100 games, we want them to explain why they own so many games. For example, perhaps they run a game resale shop, or maybe they just enjoy collecting. To conditionally add this requirement, we can use the `sometimes` method on the `Validator` instance.

```
$v->sometimes('reason', 'required|max:500', function ($input) {
    return $input->games >= 100;
});
```

The first argument passed to the `sometimes` method is the name of the field we are conditionally validating. The second argument is the rules we want to add. If the `Closure` passed as the third argument returns `true`, the rules will be added. This method makes it a breeze to build complex conditional validations. You may even add conditional validations for several fields at once:

```
$v->sometimes(['reason', 'cost'], 'required', function ($input) {
    return $input->games >= 100;
});
```

The `$input` parameter passed to your `Closure` will be an instance of `Illuminate\Support\Fluent` and may be used to access your input and files.

Validating Arrays

Validating array based form input fields doesn't have to be a pain. You may use "dot notation" to validate attributes within an array. For example, if the incoming HTTP request contains a `photos[profile]` field, you may validate it like so:

```
$validator = Validator::make($request->all(), [
    'photos.profile' => 'required|image',
]);
```

You may also validate each element of an array. For example, to validate that each e-mail in a given array input field is unique, you may do the following:

```
$validator = Validator::make($request->all(), [
    'person.*.email' => 'email|unique:users',
    'person.*.first_name' => 'required_with:person.*.last_name',
]);
```

Likewise, you may use the `*` character when specifying your validation messages in your language files, making it a breeze to use a single validation message for array based fields:

```
'custom' => [
    'person.*.email' => [
        'unique' => 'Each person must have a unique e-mail address',
    ]
],
```

Custom Validation Rules

Using Rule Objects

Laravel provides a variety of helpful validation rules; however, you may wish to specify some of your own. One method of registering custom validation rules is using rule objects. To generate a new rule object, you may use the `make:rule` Artisan command. Let's use this command to generate a rule that verifies a string is uppercase. Laravel will place the new rule in the `app/Rules` directory:

```
php artisan make:rule Uppercase
```

Once the rule has been created, we are ready to define its behavior. A rule object contains two methods: `passes` and `message`. The `passes` method receives the attribute value and name, and should return `true` or `false` depending on whether the attribute value is valid or not. The `message` method should return the validation error message that should be used when validation fails:

```
<?php

namespace App\Rules;

use Illuminate\Contracts\Validation\Rule;

class Uppercase implements Rule
{
    /**
     * Determine if the validation rule passes.
     *
     * @param  string  $attribute
     * @param  mixed  $value
     * @return bool
     */
    public function passes($attribute, $value)
    {
        return strtoupper($value) === $value;
    }

    /**
     * Get the validation error message.
     *
     * @return string
     */
    public function message()
    {
        return 'The :attribute must be uppercase.';
    }
}
```

You may call the `trans` helper from your `message` method if you would like to return an error message from your translation files:

```
/**
 * Get the validation error message.
 *
 * @return string
 */
public function message()
{
    return trans('validation.uppercase');
}
```

Once the rule has been defined, you may attach it to a validator by passing an instance of the rule object with your other validation rules:

```php
use App\Rules\Uppercase;

$request->validate([
    'name' => ['required', 'string', new Uppercase],
]);
```

Using Closures

If you only need the functionality of a custom rule once throughout your application, you may use a Closure instead of a rule object. The Closure receives the attribute's name, the attribute's value, and a `$fail` callback that should be called if validation fails:

```php
$validator = Validator::make($request->all(), [
    'title' => [
        'required',
        'max:255',
        function ($attribute, $value, $fail) {
            if ($value === 'foo') {
                $fail($attribute.' is invalid.');
            }
        },
    ],
]);
```

Using Extensions

Another method of registering custom validation rules is using the `extend` method on the `Validator` facade. Let's use this method within a service provider to register a custom validation rule:

```php
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Illuminate\Support\Facades\Validator;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Validator::extend('foo', function ($attribute, $value, $parameters, $valid
            return $value == 'foo';
        });
    }
}
```

The custom validator Closure receives four arguments: the name of the `$attribute` being validated, the `$value` of the attribute, an array of `$parameters` passed to the rule, and the `Validator` instance.

You may also pass a class and method to the `extend` method instead of a Closure:

```php
Validator::extend('foo', 'FooValidator@validate');
```

Defining The Error Message

You will also need to define an error message for your custom rule. You can do so either using an inline custom message array or by adding an entry in the validation language file. This message should be placed in the first level of the array, not within the `custom` array, which is only for attribute-specific error messages:

```php
"foo" => "Your input was invalid!",

"accepted" => "The :attribute must be accepted.",
```

```
// The rest of the validation error messages...
```

When creating a custom validation rule, you may sometimes need to define custom placeholder replacements for error messages. You may do so by creating a custom Validator as described above then making a call to the `replacer` method on the `Validator` facade. You may do this within the `boot` method of a service provider:

```
/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Validator::extend(...);

    Validator::replacer('foo', function ($message, $attribute, $rule, $parameters)
        return str_replace(...);
    });
}
```

Implicit Extensions

By default, when an attribute being validated is not present or contains an empty string, normal validation rules, including custom extensions, are not run. For example, the `unique` rule will not be run against an empty string:

```
$rules = ['name' => 'unique:users,name'];

$input = ['name' => ''];

Validator::make($input, $rules)->passes(); // true
```

For a rule to run even when an attribute is empty, the rule must imply that the attribute is required. To create such an "implicit" extension, use the `Validator::extendImplicit()` method:

```
Validator::extendImplicit('foo', function ($attribute, $value, $parameters, $valid
    return $value == 'foo';
});
```

An "implicit" extension only *implies* that the attribute is required. Whether it actually invalidates a missing or empty attribute is up to you.

Implicit Rule Objects

If you would like a rule object to run when an attribute is empty, you should implement the `Illuminate\Contracts\Validation\ImplicitRule` interface. This interface serves as a "marker interface" for the validator; therefore, it does not contain any methods you need to implement.

Become a Laravel Partner

Laravel Partners are elite shops providing top-notch Laravel development and consulting. Each of our partners can help you craft a beautiful, well-architected project.

Our Partners

Laravel

Highlights

- Release Notes
- Getting Started
- Routing
- Blade Templates
- Authentication
- Authorization
- Artisan Console
- Database
- Eloquent ORM
- Testing

Resources

- Laracasts
- Laravel News
- Laracon
- Laracon EU
- Laracon AU
- Jobs
- Certification
- Forums

Partners

- Vehikl
- Tighten Co.
- Kirschbaum
- Byte 5
- 64Robots
- Cubet
- DevSquad
- Ideil
- Cyber-Duck
- ABOUT YOU
- Become A Partner

Ecosystem

- Vapor
- Forge
- Envoyer
- Horizon
- Lumen
- Nova
- Echo
- Valet
- Mix
- Spark
- Cashier
- Homestead
- Dusk
- Passport
- Scout
- Socialite

Laravel is a web application framework with expressive, elegant syntax. We believe development must be an enjoyable and creative experience to be truly fulfilling. Laravel attempts to take the pain out of development by easing common tasks used in most web projects.