Validation

Introduction

Laravel provides several different approaches to validate your application's incoming data. It is most common to use the `validate` method available on all incoming HTTP requests. However, we will discuss other approaches to validation as well.

Laravel includes a wide variety of convenient validation rules that you may apply to data, even providing the ability to validate if values are unique in a given database table. We'll cover each of these validation rules in detail so that you are familiar with all of Laravel's validation features.

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.

By reading this high-level overview, you'll be able to gain a good general understanding of how to validate incoming request data using Laravel:

Defining The Routes

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

```
use App\Http\Controllers\PostController;

Route::get('/post/create', [PostController::class, 'create']);
Route::post('/post', [PostController::class, '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 incoming requests to 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 \Illuminate\View\View
     */
    public function create()
    {
        return view('post.create');
    }

    /**
     * Store a new blog post.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\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 `Illuminate\Validation\ValidationException` exception will be thrown and the proper error response will automatically be sent back to the user.

If validation fails during a traditional HTTP request, a redirect response to the previous URL will be generated. If the incoming request is an XHR request, a JSON response containing the validation error messages will be returned.

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

```
/**
```

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

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

As you can see, the validation rules are passed into the `validate` method. Don't worry - all available validation rules are documented. 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'],
]);
```

In addition, you may use the `validateWithBag` method to validate a request and store any error messages within a named error bag:

```
$validatedData = $request->validateWithBag('post', [
    '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 the incoming HTTP request contains "nested" field data, you may specify these fields in your validation rules using "dot" syntax:

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

On the other hand, if your field name contains a literal period, you can explicitly prevent this from being interpreted as "dot" syntax by escaping the period with a backslash:

```
$request->validate([
    'title' => 'required|unique:posts|max:255',
```

```
    'v1\.0' => 'required',
]);
```

Displaying The Validation Errors

So, what if the incoming request fields 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 and request input will automatically be flashed to the session.

An `$errors` variable is shared with all of your application's views by the `Illuminate\View\Middleware\ShareErrorsFromSession` middleware, which is provided by the `web` middleware group. When this middleware is applied an `$errors` variable will always be available in your views, allowing you to conveniently assume the `$errors` variable is always defined and can be safely used. The `$errors` variable will be an instance of `Illuminate\Support\MessageBag`. For more information on working with this object, check out its documentation.

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

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

Customizing The Error Messages

Laravel's built-in validation rules each has an error message that is located in your application's `lang/en/validation.php` file. Within this file, you will find a translation entry for each validation rule. You are free to change or modify these messages based on the needs of your application.

In addition, you may copy this file to another translation language directory to translate the messages for your application's language. To learn more about Laravel localization, check out the complete localization documentation.

XHR Requests & Validation

In this example, we used a traditional form to send data to the application. However, many applications receive XHR requests from a JavaScript powered frontend. When using the `validate` method during an XHR 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.

The `@error` Directive

You may use the `@error` Blade directive to quickly determine 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"
    name="title"
    class="@error('title') is-invalid @enderror">

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

If you are using named error bags, you may pass the name of the error bag as the second argument to the `@error` directive:

```
<input ... class="@error('title', 'post') is-invalid @enderror">
```

Repopulating Forms

When Laravel generates a redirect response due to a validation error, the framework will automatically flash all of the request's input to the session. This is done so that you may conveniently access the input during the next request and repopulate the form that the user attempted to submit.

To retrieve flashed input from the previous request, invoke the `old` method on an instance of `Illuminate\Http\Request`. The `old` method will pull the previously flashed input data from the session:

```
$title = $request->old('title');
```

Laravel also provides a global `old` helper. If you are displaying old input within a Blade template, it is more convenient to use the `old` helper to repopulate the form. If no old input exists for the given field, `null` will be returned:

```
<input type="text" name="title" value="{{ old('title') }}">
```

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.

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 encapsulate their own validation and authorization logic. To create a form request class, you may use the `make:request` Artisan CLI command:

```
php artisan make:request StorePostRequest
```

The generated form request 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. Each form request generated by Laravel has two methods: `authorize` and `rules`.

As you might have guessed, the `authorize` method is responsible for determining if the currently authenticated user can perform the action represented by the request, while the `rules` method returns the validation rules that should apply to the request's data:

```
/**
 * 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 require 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 a new blog post.
 *
 * @param  \App\Http\Requests\StorePostRequest  $request
 * @return Illuminate\Http\Response
 */
public function store(StorePostRequest $request)
{
    // The incoming request is valid...

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

    // Retrieve a portion of the validated input data...
    $validated = $request->safe()->only(['name', 'email']);
    $validated = $request->safe()->except(['name', 'email']);
}
```

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 XHR request, an 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" validation 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 fi
            }
        });
    }
```

Stopping On First Validation Failure Attribute

By adding a `stopOnFirstFailure` property to your request class, you may inform the validator that it should stop validating all attributes once a single validation failure has occurred:

```
    /**
     * Indicates if the validator should stop on the first rule failure.
     *
     * @var bool
     */
    protected $stopOnFirstFailure = true;
```

Customizing The Redirect Location

As previously discussed, a redirect response will be generated to send the user back to their previous location when form request validation fails. However, you are free to customize this behavior. To do so, define a `$redirect` property on your form request:

```
    /**
     * The URI that users should be redirected to if validation fails.
     *
     * @var string
     */
    protected $redirect = '/dashboard';
```

Or, if you would like to redirect users to a named route, you may define a `$redirectRoute` property instead:

```
    /**
     * The route that users should be redirected to if validation fails.
     *
     * @var string
     */
    protected $redirectRoute = 'dashboard';
```

Authorizing Form Requests

The form request class also contains an `authorize` method. Within this method, you may determine 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. Most likely, you will interact with your authorization gates and policies within this method:

```
use App\Models\Comment;

    /**
     * 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}');
```

Therefore, if your application is taking advantage of route model binding, your code may be made even more succinct by accessing the resolved model as a property of the request:

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

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

If you plan to handle authorization logic for the request in another part of your application, you may simply 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

Many of Laravel's built-in validation rule error messages contain an `:attribute` placeholder. If you would like the `:attribute` placeholder 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:

```php
/**
 * Get custom attributes for validator errors.
 *
 * @return array
 */
public function attributes()
{
    return [
        'email' => 'email address',
    ];
}
```

Preparing Input For Validation

If you need to prepare or sanitize any data from the request before you apply your validation rules, you may use the `prepareForValidation` method:

```php
use Illuminate\Support\Str;

/**
 * Prepare the data for validation.
 *
 * @return void
 */
protected function prepareForValidation()
{
    $this->merge([
        'slug' => Str::slug($this->slug),
    ]);
}
```

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
<?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();
        }
```

```
        // Retrieve the validated input...
        $validated = $validator->validated();

        // Retrieve a portion of the validated input...
        $validated = $validator->safe()->only(['name', 'email']);
        $validated = $validator->safe()->except(['name', 'email']);

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

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

After determining whether 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`.

Stopping On First Validation Failure

The `stopOnFirstFailure` method will inform the validator that it should stop validating all attributes once a single validation failure has occurred:

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

Automatic Redirection

If you would like to create a validator instance manually but still take advantage of the automatic redirection offered by the HTTP 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 XHR request, a JSON response will be returned:

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

You may use the `validateWithBag` method to store the error messages in a named error bag if validation fails:

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

Named Error Bags

If you have multiple forms on a single page, you may wish to name the `MessageBag` containing the validation errors, allowing you to retrieve the error messages for a specific form. To achieve this, 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') }}
```

Customizing The Error Messages

If needed, you may provide custom error messages that a validator instance should use instead of the default error messages provided by Laravel. There are several ways to specify custom messages. First, you may pass the custom messages as the third argument to the `Validator::make` method:

```
$validator = Validator::make($input, $rules, $messages = [
    'required' => 'The :attribute field is required.',
]);
```

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 attribute. 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 email address!',
];
```

Specifying Custom Attribute Values

Many of Laravel's built-in error messages include an `:attribute` placeholder that is replaced with the name of the field or attribute under validation. To customize the values used to replace these placeholders for specific fields, you may pass an array of custom attributes as the fourth argument to the `Validator::make` method:

```
$validator = Validator::make($input, $rules, $messages, [
    'email' => 'email address',
]);
```

After Validation Hook

You may also 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, call 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 Validated Input

After validating incoming request data using a form request or a manually created validator instance, you may wish to retrieve the incoming request data that actually underwent validation. This can be accomplished in several ways. First, you may call the `validated` method on a form request or validator instance. This method returns an array of the data that was validated:

```
$validated = $request->validated();

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

Alternatively, you may call the `safe` method on a form request or validator instance. This method returns an instance of `Illuminate\Support\ValidatedInput`. This object exposes `only`, `except`, and `all` methods to retrieve a subset of the validated data or the entire array of validated data:

```
$validated = $request->safe()->only(['name', 'email']);

$validated = $request->safe()->except(['name', 'email']);

$validated = $request->safe()->all();
```

In addition, the `Illuminate\Support\ValidatedInput` instance may be iterated over and accessed like an array:

```
// Validated data may be iterated...
foreach ($request->safe() as $key => $value) {
    //
}

// Validated data may be accessed as an array...
$validated = $request->safe();

$email = $validated['email'];
```

If you would like to add additional fields to the validated data, you may call the `merge` method:

```
$validated = $request->safe()->merge(['name' => 'Taylor Otwell']);
```

If you would like to retrieve the validated data as a collection instance, you may call the `collect` method:

```
$collection = $request->safe()->collect();
```

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')) {
    //
}
```

Specifying Custom Messages In Language Files

Laravel's built-in validation rules each has an error message that is located in your application's `lang/en/validation.php` file. Within this file, you will find a translation entry for each validation rule. You are free to change or modify these messages based on the needs of your application.

In addition, you may copy this file to another translation language directory to translate the messages for your application's language. To learn more about Laravel localization, check out the complete localization documentation.

Custom Messages For Specific Attributes

You may customize the error messages used for specified attribute and rule combinations within your application's validation language files. To do so, add your message customizations to the `custom` array of your application's `lang/xx/validation.php` language file:

```
'custom' => [
    'email' => [
        'required' => 'We need to know your email address!',
        'max' => 'Your email address is too long!'
    ],
],
```

Specifying Attributes In Language Files

Many of Laravel's built-in error messages include an `:attribute` placeholder that is replaced with the name of the field or attribute under validation. If you would like the `:attribute` portion of your validation message to be replaced with a custom value, you may specify the custom attribute name in the `attributes` array of your `lang/xx/validation.php` language file:

```
'attributes' => [
```

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

Specifying Values In Language Files

Some of Laravel's built-in validation rule error messages contain a `:value` placeholder that is replaced with the current value of the request attribute. However, you may occasionally 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`:

```
Validator::make($request->all(), [
    '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 more user-friendly value representation in your `lang/xx/validation.php` language file by defining a `values` array:

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

After defining this value, the validation rule will produce the following error 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
Accepted If
Active URL
After (Date)
After Or Equal (Date)
Alpha
Alpha Dash
Alpha Numeric
Array
Bail
Before (Date)
Before Or Equal (Date)
Between
Boolean
Confirmed
Current Password
Date
Date Equals
Date Format
Declined
Declined If
Different
Digits
Digits Between
Dimensions (Image Files)
Distinct
Email
Ends With
Enum
Exclude
Exclude If
Exclude Unless
Exclude With
Exclude Without
Exists (Database)
File
Filled
Greater Than
Greater Than Or Equal
Image (File)
In
In Array
Integer
IP Address
JSON
Less Than
Less Than Or Equal
MAC Address
Max
MIME Types
MIME Type By File
Extension
Min
Multiple Of
Not In
Not Regex
Nullable
Numeric
Password
Present
Prohibited
Prohibited If
Prohibited Unless
Prohibits
Regular Expression
Required

accepted

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

accepted_if:anotherfield,value,...

The field under validation must be `"yes"`, `"on"`, `1`, or `true` if another field under validation is equal to a specified value. This is useful for validating "Terms of Service" acceptance or similar fields.

active_url

The field under validation must have a valid A or AAAA record according to the `dns_get_record` PHP function. The hostname of the provided URL is extracted using the `parse_url` PHP function before being passed to `dns_get_record`.

after:*date*

The field under validation must be a value after a given date. The dates will be passed into the `strtotime` PHP function in order to be converted to a valid `DateTime` instance:

```
'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`.

When additional values are provided to the `array` rule, each key in the input array must be present within the list of values provided to the rule. In the following example, the `admin` key in the input array is invalid since it is not contained in the list of values provided to the `array` rule:

```
use Illuminate\Support\Facades\Validator;

$input = [
    'user' => [
        'name' => 'Taylor Otwell',
        'username' => 'taylorotwell',
        'admin' => true,
```

```
    ],
];

Validator::make($input, [
    'user' => 'array:username,locale',
]);
```

In general, you should always specify the array keys that are allowed to be present within your array.

bail

Stop running validation rules for the field after the first validation failure.

While the `bail` rule will only stop validating a specific field when it encounters a validation failure, the `stopOnFirstFailure` method will inform the validator that it should stop validating all attributes once a single validation failure has occurred:

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

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 order to be converted into a valid `DateTime` instance. 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 order to be converted into a valid `DateTime` instance. 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 `{field}_confirmation`. For example, if the field under validation is `password`, a matching `password_confirmation` field must be present in the input.

current_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' => 'current_password:api'
```

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 in order to be converted into a valid `DateTime` instance.

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.

declined

The field under validation must be `"no"`, `"off"`, `0`, or `false`.

declined_if:anotherfield,value,...

The field under validation must be `"no"`, `"off"`, `0`, or `false` if another field under validation is equal to a specified value.

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 be *numeric* and 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 fraction 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\Support\Facades\Validator;
use Illuminate\Validation\Rule;

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

distinct

When validating arrays, the field under validation must not have any duplicate values:

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

Distinct uses loose variable comparisons by default. To use strict comparisons, you may add the `strict` parameter to your validation rule definition:

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

You may add `ignore_case` to the validation rule's arguments to make the rule ignore capitalization differences:

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

email

The field under validation must be formatted as an email address. This validation rule utilizes 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, ships with Laravel and was Laravel's default email validation behavior prior to Laravel version 5.8.

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.

enum

The `Enum` rule is a class based rule that validates whether the field under validation contains a valid enum value. The `Enum` rule accepts the name of the enum as its only constructor argument:

```
use App\Enums\ServerStatus;
use Illuminate\Validation\Rules\Enum;

$request->validate([
    'status' => [new Enum(ServerStatus::class)],
]);
```

Enums are only available on PHP 8.1+.

exclude

The field under validation will be excluded from the request data returned by the `validate` and `validated` methods.

exclude_if:*anotherfield,value*

The field under validation will be excluded from the request data returned by the `validate` and `validated` methods if the *anotherfield* field is equal to *value*.

If complex conditional exclusion logic is required, you may utilize the `Rule::excludeIf` method. This method accepts a boolean or a closure. When given a closure, the closure should return `true` or `false` to indicate if the field under validation should be excluded:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

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

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

exclude_unless:*anotherfield*,*value*

The field under validation will be excluded from the request data returned by the `validate` and `validated` methods unless *anotherfield*'s field is equal to *value*. If *value* is `null` (`exclude_unless:name,null`), the field under validation will be excluded unless the comparison field is `null` or the comparison field is missing from the request data.

exclude_with:*anotherfield*

The field under validation will be excluded from the request data returned by the `validate` and `validated` methods if the *anotherfield* field is present.

exclude_without:*anotherfield*

The field under validation will be excluded from the request data returned by the `validate` and `validated` methods if the *anotherfield* field is not present.

exists:*table*,*column*

The field under validation must exist in 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. So, in this case, the rule will validate that the `states` database table contains a record with a `state` column value matching the request's `state` attribute value.

Specifying A Custom Column Name

You may explicitly specify the database column name that should be used by the validation rule by placing it after the database table 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:

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

Instead of specifying the table name directly, you may specify the Eloquent model which should be used to determine the table name:

```
'user_id' => 'exists:App\Models\User,id'
```

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:

```php
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

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

You may explicitly specify the database column name that should be used by the `exists` rule generated by the `Rule::exists` method by providing the column name as the second argument to the `exists` method:

```php
'state' => Rule::exists('states', 'abbreviation'),
```

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 (jpg, 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:

```php
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

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

When the `in` rule is combined with the `array` rule, each value in the input array must be present within the list of values provided to the `in` rule. In the following example, the `LAS` airport code in the input array is invalid since it is not contained in the list of airports provided to the `in` rule:

```php
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

$input = [
    'airports' => ['NYC', 'LAS'],
];
```