Course - Laravel Framework

Giám định Form

Laravel cung cấp một số cách để giám định dữ liệu đầu vào từ request gửi đến cho ứng dụng của bạn.

Tags: form validate, laravel

Giới thiệu

Phổ biến nhất là sử dụng phương thức `validate` có sẵn trên tất cả các HTTP request gửi đến. Tuy nhiên, chúng ta cũng sẽ thảo luận về các cách tiếp cận khác để giám định form.

Laravel bao gồm nhiều quy tắc kiểm tra giám định thuận tiện mà bạn có thể áp dụng để giám định dữ liệu đầu vào, thậm chí cung cấp khả năng kiểm tra giám định cho các giá trị là duy nhất trong một bảng cơ sở dữ liệu nào đó. Chúng tôi sẽ trình bày chi tiết từng quy tắc kiểm tra giám định này để bạn làm quen với tất cả các tính năng giám định của Laravel.

Bắt đầu

Để tìm hiểu về các tính năng giám định mạnh mẽ của Laravel, hãy xem một ví dụ hoàn chỉnh về kiểm tra giám định form và hiển thị thông báo lỗi trở lại người dùng. Bằng cách đọc bài tổng quát này, bạn sẽ có thể hiểu biết thêm về cách giám định dữ liệu request gửi đến bằng Laravel:

Khai báo route

Trước tiên, hãy giả sử chúng ta có các route sau được khai báo trong tập tin *routes/web.php* của chúng ta:

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

Route::get('/post/create', [PostController::class, 'create']);
Route::post('/post', [PostController::class, 'store']);
```

Route `GET` sẽ hiển thị một form để người dùng tạo một bài đăng blog mới, trong khi route `POST` sẽ lưu trữ bài đăng blog mới trong cơ sở dữ liệu.

Tạo Controller

Tiếp theo, chúng ta hãy xem một controller đơn giản xử lý các request gửi đến các tuyến này. Bây giờ chúng ta sẽ để trống phương thức `store`:

```
<?php
namespace App\Http\Controllers;
use App\Http\Controllers\Controller;
```

```php
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...
  }
}
```

Viết logic kiểm tra

Bây giờ chúng ta đã sẵn sàng điền vào phương thức `store` của chúng ta với logic để giám định bài đăng blog mới. Để làm điều này, chúng ta sẽ sử dụng phương thức `validate` được cung cấp bởi đối tượng `Illuminate\Http\Request`. Nếu các quy tắc giám định được thông qua, mã của bạn sẽ tiếp tục thực thi bình thường; tuy nhiên, nếu kết quả giám định không thành công, một exception của `Illuminate\Validation\ValidationException` sẽ được đưa ra và response lỗi thích hợp sẽ tự động được gửi lại cho người dùng.

Nếu kết quả giám định không thành công trong một HTTP request truyền thống, thì một response chuyển hướng đến URL trước đó sẽ được tạo ra. Nếu request gửi đến là một XHR request, thì một response JSON chứa các thông báo lỗi kiểm tra giám định sẽ được trả về.

Để hiểu rõ hơn về phương thức `validate`, hãy quay lại phương thức `store`:

```
/**
 * 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...
}
```

Như bạn có thể thấy, các quy tắc giám định được truyền vào phương thức `validate`. Đừng lo lắng - tất cả các quy tắc giám định hiện có đều được ghi lại. Một lần nữa, nếu kết quả giám định không thành công, thì response thích hợp sẽ tự động được tạo. Nếu quá trình giám định vượt qua, thì controller của chúng ta sẽ tiếp tục thực thi bình thường.

Ngoài ra, các quy tắc giám định có thể được chỉ định dưới dạng mảng các quy tắc giám định thay vì một chuỗi phân cách với dấu `|`:

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

Ngoài ra, bạn có thể sử dụng phương thức `validateWithBag` để giám định một request và lưu trữ bất kỳ thông báo lỗi nào trong một named error bag (túi lỗi được đặt tên):

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

Dừng tại lỗi đầu tiên

Đôi khi bạn có thể muốn ngừng chạy các quy tắc giám định (rule) trên một *attribute* sau lần giám định đầu tiên không thành công. Để làm như vậy, hãy khai báo quy tắc `bail` cho thuộc tính:

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

Trong ví dụ này, nếu quy tắc rule `unique` trên thuộc tính `title` không thành công, quy tắc tối đa `max` sẽ không được chọn. Các quy tắc sẽ được giám định theo thứ tự mà chúng được bố trí.

Node trên các thuộc tính lồng ghép

Nếu HTTP request gửi đến chứa dữ liệu field lồng ghép vào nhau, bạn có thể khai báo các field này trong quy tắc giám định của mình bằng cú pháp chấm:

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

Mặt khác, nếu tên trường của bạn chứa dấu chấm theo nghĩa đen, thì bạn có thể ngăn điều này được hiểu theo cú pháp giám định bằng cách thoát dấu chấm của field với dấu gạch chéo ngược:

```
$request->validate([
  'title' => 'required|unique:posts|max:255',
  'v1\.0' => 'required',
]);
```

Hiển thị lỗi giám định form

Vì vậy, điều gì sẽ xảy ra nếu các field trên request gửi đến không vượt qua các quy tắc giám định đã cho? Như đã đề cập trước đây, Laravel sẽ tự động chuyển hướng người dùng trở lại vị trí trước đó của họ. Ngoài ra, tất cả các lỗi giám định và dữ liệu đầu vào sẽ tự động được chuyển vào trong session.

Biến `$error` được middleware `Illuminate\View\Middleware\ShareErrorsFromSession` chia sẻ với tất cả các bản view của ứng dụng của bạn, được cung cấp bởi nhóm middleware web. Khi middleware này được áp dụng, một biến `$error` sẽ luôn có sẵn trong các bảng view của bạn, cho phép bạn thuận tiện giả định rằng biến `$error` luôn được xác định và có thể được sử dụng một cách an toàn. Biến `$error` sẽ là một đối tượng của `Illuminate\Support\MessageBag`. Để biết thêm thông tin về cách làm việc với đối tượng này, hãy xem tài liệu của nó.

Vì vậy, trong ví dụ của chúng tôi, người dùng sẽ được chuyển hướng đến phương thức `create` của bộ điều khiển của chúng tôi khi giám định không thành công, cho phép chúng tôi hiển thị thông báo lỗi trong bản 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 -->
```

Tuỳ chỉnh thông điệp lỗi

Mỗi quy tắc giám định tích hợp của Laravel đều có một thông báo lỗi nằm trong tập tin tài nguyên */lang/en/validation.php* của ứng dụng của bạn. Trong tập tin này, bạn sẽ tìm thấy một mục thông dịch ngôn ngữ cho từng quy tắc giám định. Bạn có thể tự do thay đổi hoặc sửa đổi các thông báo này dựa trên nhu cầu của ứng dụng của bạn.

Ngoài ra, bạn có thể sao chép tập tin này sang một thư mục thông dịch ngôn ngữ khác để dịch các thông báo cho ngôn ngữ ứng dụng của bạn. Để tìm hiểu thêm về việt hoá Laravel, hãy xem toàn bộ tài liệu việt hóa.

Giám định XHR request

Trong ví dụ này, chúng tôi đã sử dụng một form truyền thống để gửi dữ liệu đến ứng dụng. Tuy nhiên, nhiều ứng dụng nhận được XHR request từ giao diện người dùng sử dụng JavaScript. Khi sử dụng phương thức `validate` cho XHR request, Laravel sẽ không tạo response chuyển hướng. Thay vào đó, Laravel tạo một JSON response chứa tất cả các lỗi giám định. JSON response này sẽ được gửi với mã trạng thái `422` HTTP.

Chỉ thị @error

Bạn có thể sử dụng chỉ thị `@error` của Blade để nhanh chóng xác định xem thông báo lỗi giám định có tồn tại cho một *attribute* nào đó hay không. Trong chỉ thị `@error`, bạn có thể lặp lại biến `$message` để hiển thị thông báo lỗi:

```
<!-- /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
```

Nếu bạn đang sử dụng các lỗi đã được đặt tên (named error bag), bạn có thể truyền tên của lỗi đó làm đối số thứ hai cho chỉ thị `@error`:

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

Tái hiện form cũ

Khi Laravel tạo response chuyển hướng do lỗi giám định form, framework sẽ tự động chuyển tất cả đầu vào của request vào session. Điều này được thực hiện để bạn có thể thuận tiện truy cập đầu vào trong lần request tiếp theo và tạo lại form mà người dùng đã cố gắng gửi trước đó.

Để truy xuất đầu vào đã được lưu trữ từ request trước đó, hãy gọi phương thức `old` trên đối tượng của `Illuminate\Http\Request`. Phương thức `old` sẽ lấy dữ liệu đầu vào đã được lưu trước đó từ session:

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

Laravel cung cấp một hàm trợ giúp `old`. Nếu bạn đang hiển thị đầu vào đã nhập trong template Blade ở request trước đó, sẽ thuận tiện hơn khi sử dụng hàm `old` để tạo lại form. Nếu không có đầu vào nào tồn tại cho field đã cho, thì giá trị `null` sẽ được trả về:

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

Các field tuỳ chọn

Mặc định, Laravel đưa sẵn vào middleware `TrimStrings` và `ConvertEmptyStringsToNull` trong stack middleware tổng bộ của ứng dụng của bạn. Các middleware này được liệt kê trong stack bởi class `App\Http\Kernel`. Do đó, bạn thường sẽ cần phải đánh dấu các field với "optional" là `nullable` nếu bạn không muốn chương trình giám định coi các giá trị `null` là không hợp lệ. Ví dụ:

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

Trong ví dụ này, chúng ta đang chỉ định các field `publish_at` có thể là rỗng hoặc là đại diện ngày hợp lệ. Nếu modifier `nullable` không được thêm vào khi khai báo quy tắc giám định, thì chương trình giám định sẽ coi `null` là một ngày không hợp lệ.

Giám định request form

Tạo form request

Đối với các tình huống giám định phức tạp hơn, bạn có thể muốn tạo một "form request". Form request là các class Request tùy chỉnh đính kèm logic giám định và ủy quyền của riêng chúng. Để tạo một class form request, bạn có thể sử dụng lệnh Artisan `make:request`:

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

Class form request đã tạo sẽ được đặt trong thư mục *app/Http/Request*. Nếu thư mục này không tồn tại, nó sẽ được tạo khi bạn chạy lệnh `make:request`. Từng form request do Laravel tạo ra sẽ có hai phương thức: `authorize` và `rules`.

Như bạn có thể đoán, phương thức `authorize` chịu trách nhiệm xác định xem liệu người dùng hiện được giám định có thể thực hiện hành động được đại diện bởi request hay không, trong khi phương thức `rules` trả về các quy tắc giám định sẽ áp dụng cho dữ liệu của request:

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

> Bạn có thể khai kiểu cho bất kỳ thư viện nào bạn yêu cầu trong khai báo của phương thức `rules`. Chúng sẽ tự động được giải quyết thông qua service container của Laravel.

Vì vậy, các quy tắc giám định được đánh giá như thế nào? Tất cả những gì bạn cần làm là khai kiểu thư viện được yêu cầu trên phương thức controller của bạn. Form request gửi đến sẽ được giám định trước khi phương thức controller được gọi, có nghĩa là bạn không cần phải làm lộn xộn controller của mình với bất kỳ logic giám định nào:

```
/**
 * 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']);
}
```

Nếu kết quả giám định không thành công, một response chuyển hướng sẽ được tạo để đưa người dùng trở lại vị trí trước đó của họ. Các lỗi cũng sẽ được hiển thị trong session để chúng có sẵn để hiển thị. Nếu request là một XHR request, thì HTTP response với mã trạng thái `422` sẽ được trả lại cho người dùng bao gồm dữ liệu JSON về các lỗi giám định.

Thêm after hook vào form request

Hook "after" (được ví như một khâu hậu kỳ cho quá trình giám định). Nếu bạn muốn thêm hook "after" vào form request, bạn có thể sử dụng phương thức `withValidator`. Phương thức này cho phép truyền đầy đủ chương trình giám định (validator), qua đó nó cho phép bạn gọi được đến bất kỳ phương thức nào của validator (chương trình giám định) trước khi các quy tắc giám định thực sự được đánh giá:

```
/**
```

```
     * 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!');
            }
        });
    }
```

Dừng tại lỗi đầu tiên

Bằng cách thêm thuộc tính `stopOnFirstFailure` vào class request của mình, bạn có thể thông báo cho chương trình giám định rằng nó sẽ ngừng giám định tất cả các *attribute* sau khi một lỗi giám định xảy ra:

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

Tuỳ biến chuyển hướng

Như đã thảo luận trước đây, một response chuyển hướng sẽ được tạo để đưa người dùng trở lại vị trí trước đó của họ khi giám định form không thành công. Tuy nhiên, bạn có thể tự do tùy chỉnh chức năng này. Để làm như vậy, hãy chỉ định thuộc tính `$redirect` cho form request của bạn:

```
    /**
     * The URI that users should be redirected to if validation fails.
```

```
     *
     * @var string
     */
    protected $redirect = '/dashboard';
```

Hoặc, nếu bạn muốn chuyển hướng người dùng đến một route đã đặt tên, bạn có thể chỉ định thuộc tính `$redirectRoute` để thay thế:

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

Form request giám định

Class của form request cũng chứa một phương thức `authorize`. Trong phương thức này, bạn có thể xác định xem người dùng được xác thực có thực sự có quyền cập nhật một tài nguyên nào đó hay không. Ví dụ: bạn có thể xác minh xem người dùng có thực sự sở hữu một bình luận nào đó mà họ đang cố gắng cập nhật hay không. Rất có thể, bạn sẽ tương tác với các cổng và chính sách xác thực của mình trong phương pháp này:

```
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);
}
```

Vì tất cả các form request đều mở rộng từ class Request của Laravel, nên chúng ta có thể sử dụng phương thức `user` để truy cập người dùng đã được xác thực hiện tại. Ngoài ra, hãy lưu ý lệnh gọi phương thức `route` trong ví dụ trên. Phương thức này cấp cho bạn quyền truy cập vào các tham số URI được chỉ định trên route đang được gọi, chẳng hạn như tham số `{comment}` trong ví dụ bên dưới:

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

Do đó, nếu ứng dụng của bạn đang tận dụng lợi thế của route model binding (một kiểu truy xuất trực tiếp model trên route), code của bạn có thể ngắn gọn hơn nữa bằng cách truy cập model đã nhận được dưới dạng thuộc tính của request:

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

Nếu phương thức `authorize` trả về `false`, HTTP response có mã trạng thái `403` sẽ tự động được trả về và phương thức controller của bạn sẽ không thực thi.

Nếu bạn đang định xử lý logic xác minh cho request trong một phần khác của ứng dụng của mình, bạn có thể chỉ cần trả về `true` từ phương thức `authorize`:

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

> Bạn có thể khai kiểu cho bất kỳ thư viện nào bạn cần trong khai báo của phương thức `authorize`. Chúng sẽ tự động được giải quyết thông qua service container Laravel.

Tuỳ chỉnh thông báo lỗi

Bạn có thể tùy chỉnh các thông báo lỗi được sử dụng bởi form request bằng cách sửa lại

phương thức `messages`. Phương thức này sẽ trả về một mảng các cặp *attribute/rule* và các thông báo lỗi tương ứng của chúng:

```
/**
 * 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',
  ];
}
```

Tùy chỉnh thuộc tính giám định

Nhiều thông báo lỗi quy tắc giám định được tích hợp sẵn của Laravel có chứa `:attribute`. Nếu bạn muốn thay thế `:attribute` của thông báo giám định bằng một tên thuộc tính tùy chọn, bạn có thể chỉ định các tên tùy chọn bằng cách ghi đè phương thức `attributes`. Phương thức này sẽ trả về một mảng các cặp attribute/name:

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

Chuẩn bị input để giám định

Nếu bạn cần chuẩn bị hoặc quy chuẩn bất kỳ dữ liệu nào từ request trước khi áp dụng các quy tắc giám định của mình, bạn có thể sử dụng phương thức `prepareForValidation`:

```
use Illuminate\Support\Str;

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

Tạo thủ công chương trình giám định

Nếu bạn không muốn sử dụng phương thức `validate` theo request, thì bạn có thể tạo đối tượng giám định theo cách thủ công bằng cách sử dụng facade `Validator`. Phương thức `make` trên facade này tạo ra một đối tượng giám định mới:

```
<?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...
    }
}
```

Đối số đầu tiên được truyền cho phương thức `make` là dữ liệu đang được giám định. Đối số thứ hai là một mảng các quy tắc giám định sẽ được áp dụng để kiểm tra dữ liệu đầu vào.

Sau khi xác định xem việc giám định request có thất bại hay không, bạn có thể sử dụng phương thức `withErrors` để đưa thông báo lỗi cho session. Khi sử dụng phương thức này, biến `$error` sẽ tự động được chia sẻ với các bản view của bạn sau khi chuyển hướng, cho phép bạn dễ dàng hiển thị lại cho người dùng. Phương thức `withErrors` chấp nhận một chương trình giám định, một `MessageBag` hoặc một mảng PHP.

Dừng tại giám định lỗi đầu tiên

Phương thức `stopOnFirstFailure` sẽ thông báo cho chương trình giám định rằng nó sẽ ngừng giám định tất cả các thuộc tính sau khi một lỗi giám định xảy ra:

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

Chuyển hướng tự động

Nếu bạn muốn tạo một đối tượng giám định theo cách thủ công nhưng vẫn tận dụng lợi thế của chuyển hướng tự động được cung cấp bởi phương thức `validate` của HTTP request, thì bạn có thể gọi phương thức `validate` trên một đối tượng giám định hiện có. Nếu giám định không thành công, người dùng sẽ tự động được chuyển hướng hoặc trong trường hợp có XHR request, thì JSON response sẽ được trả lại:

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

Bạn có thể sử dụng phương thức `validateWithBag` để lưu trữ các thông báo lỗi trong một túi chứa lỗi được đặt tên (named error bag) nếu quá trình giám định không thành công:

Túi lỗi được đặt tên (named error bag)

Nếu bạn có nhiều form trên một trang, bạn có thể sẽ muốn dùng `MessageBag` chứa các lỗi giám định, nó cho phép bạn truy xuất các thông báo lỗi cho một form cụ thể nào đó. Để làm được điều này, bạn hãy truyền vào một cái tên làm đối số thứ hai cho phương thức `withErrors`:

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

Sau đó, bạn có thể truy cập đối tượng `MessageBag` được cài đặt như ở trên từ biến `$error`:

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

Tùy chỉnh nội dung lỗi

Nếu cần, bạn có thể cung cấp các thông báo lỗi tùy chỉnh mà một đối tượng giám định sẽ sử dụng thay cho các thông báo lỗi mặc định do Laravel cung cấp. Có một số cách để chỉ định thông báo tùy chỉnh. Đầu tiên, bạn có thể truyền các thông báo tùy chỉnh làm đối số thứ ba cho phương thức `Validator::make`:

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

Trong ví dụ này, biến thuộc tính `:attribute` sẽ được thay thế bằng tên thực của field đang được giám định, vì `:attribute` đại diện cho tên field. Bạn cũng có thể sử dụng thêm các biến khác trong các thông báo giám định. Ví dụ:

```
$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',
];
```

Tùy chỉnh thuộc tính có sẵn

Đôi khi bạn có thể chỉ muốn tự tạo một thông báo lỗi cho một field nào đó. Bạn có thể thực hiện điều này bằng cách sử dụng ký hiệu "`.`". Chỉ định tên của thuộc tính trước, sau đó là quy tắc:

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

Chỉ định giá trị của thuộc tính tuỳ chỉnh

Nhiều thông báo lỗi có sẵn của Laravel bao gồm biến thuộc tính `:attribute` được dùng để gán tên của field đang được giám định. Để tùy chỉnh giá trị của field cụ thể nào đó, bạn

có thể truyền một mảng các cặp *name/value* làm đối số thứ tư cho phương thức `Validator::make`:

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

After hook trong chương trình giám định

Bạn cũng có thể đính kèm các hàm callback sẽ được chạy sau khi quá trình giám định hoàn tất. Điều này cho phép bạn dễ dàng thực hiện giám định thêm một cái gì đó bạn muốn và thậm chí thêm được nhiều chi tiết thông báo lỗi hơn vào dữ kiện thông báo. Để bắt đầu, hãy gọi phương thức `after` trên đối tượng `Validator`:

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

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

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

Làm việc với dữ liệu đã được giám định

Sau khi giám định dữ liệu trong request gửi đến bằng form request (form request được mở rộng từ đối tượng Request của Laravel) hoặc bằng đối tượng `Validator` được tạo thủ công, bạn có thể muốn truy xuất dữ liệu request gửi đến khi chúng thực sự vượt qua quá trình giám định. Điều này có thể được thực hiện bằng một số cách. Đầu tiên, bạn có thể gọi phương thức `validated` trên một form request hoặc đối tượng `Vallidator`. Phương thức này trả về một mảng dữ liệu đã được giám định:

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

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

Ngoài ra, bạn có thể gọi phương thức `safe` trên một form request hoặc đối tượng `Validator`. Phương thức này trả về một đối tượng của `Illuminate\Support\ValidatedInput`. Đối tượng này tiết lộ các phương thức `only`, `except` và `all` để truy xuất một phần nào đó của dữ liệu đã được giám định hoặc toàn bộ mảng dữ liệu đã được giám định:

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

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

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

Ngoài ra, đối tượng `Illuminate\Support\ValidatedInput` có thể được đặt vào trong một vòng lặp và truy cập tuần tự giống như một mảng dữ liệu bình thường:

```php
// 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'];
```

Nếu bạn muốn thêm các field bổ sung vào dữ liệu đã được giám định, bạn có thể gọi phương thức `merge`:

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

Taylor Otwell là người đã viết ra Laravel thần thánh.

Nếu bạn muốn truy xuất dữ liệu đã được giám định dưới dạng một đối tượng Collection, thì bạn có thể gọi phương thức `collect`:

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

Làm việc với nội dung lỗi

Sau khi gọi phương thức `errors` trên đối tượng `Validator`, bạn sẽ nhận được một đối tượng `Illuminate\Support\MessageBag`, nó sẽ có nhiều phương thức thuận tiện để làm việc với các thông báo lỗi. Biến `$errors` được cung cấp tự động cho tất cả các bản view trong Laravel cũng là một đối tượng `MessageBag`.

Nhận thông báo lỗi đầu tiên của một field

Để truy xuất thông báo lỗi đầu tiên cho một field nhất định nào đó, thì hãy sử dụng phương thức `first`:

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

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

Nhận tất cả thông báo lỗi của một field

Nếu bạn cần truy xuất một mảng gồm tất cả các thông báo cho một field nhất định nào đó, thì hãy sử dụng phương thức `get`:

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

Nếu bạn đang giám định một field dạng mảng, bạn có thể truy xuất tất cả các thông báo cho từng phần tử của mảng bằng cách sử dụng ký tự *:

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

```
}
```

Nhận tất cả thông báo lỗi cho tất cả các field

Để truy xuất một mảng gồm tất cả các thông báo cho tất cả các field, thì hãy sử dụng phương thức `all`:

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

Kiểm tra field có thông báo lỗi tương ứng

Phương thức `has` có thể được sử dụng để xác định xem có bất kỳ thông báo lỗi nào tồn tại cho một field nhất định nào đó hay không:

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

Chỉ định thông báo lỗi cho các tập tin language

Mỗi quy tắc giám định được tích hợp sẵn trong Laravel đều có một thông báo lỗi nằm trong tập tin *resources/lang/en/validation.php* của ứng dụng của bạn. Trong tập tin này, bạn sẽ tìm thấy một mục thông dịch cho từng quy tắc giám định. Bạn có thể tự do thay đổi hoặc sửa đổi các thông báo này dựa trên nhu cầu của ứng dụng của bạn.

Ngoài ra, bạn có thể sao chép tập tin này sang một thư mục thông dịch ngôn ngữ khác để phiên dịch các thông báo cho ngôn ngữ của ứng dụng của bạn. Để tìm hiểu thêm về Việt hóa Laravel, hãy xem toàn bộ tài liệu việt hóa.

Tuỳ chỉnh thông báo cho các field cụ thể

Bạn có thể tùy chỉnh các thông báo lỗi được sử dụng cho các field và quy tắc được khai báo trong các tập tin ngôn ngữ về giám định dữ liệu đầu vào. Để làm được như vậy, hãy tùy chỉnh thông báo của bạn trong mảng tùy chỉnh của tập tin ngôn ngữ *resources/lang*

/xx/validation.php của ứng dụng:

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

Thông dịch tên của field với tập tin ngôn ngữ

Nhiều thông báo lỗi tích hợp sẵn của Laravel bao gồm biến thuộc tính `:attribute` đại diện tên của field đang được giám định. Nếu bạn muốn `:attribute` của thông báo giám định của mình được thay thế bằng một giá trị tùy chỉnh, thì bạn có thể chỉ định tên *attribute* tùy chỉnh trong mảng các *attributes* trong tập tin ngôn ngữ *resources/lang/xx/validation.php* của bạn:

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

Chỉ định các giá trị trong tập tin ngôn ngữ

Một số thông báo lỗi quy tắc giám định có sẵn trong Laravel có chứa một biến :value được thay thế bằng giá trị hiện tại của field đã request. Tuy nhiên, đôi khi bạn có thể cần biến `:value` của thông báo giám định của mình được thay thế bằng nội dung tùy chỉnh. Ví dụ: hãy xem xét quy tắc rule giám định sau đây chỉ định rằng số thẻ tín dụng là bắt buộc nếu `payment_type` (kiểu thanh toán) có giá trị là `cc` (credit card):

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

Nếu quy tắc giám định này không thành công, nó sẽ tạo ra thông báo lỗi sau:

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

Thay vì hiển thị `cc` dưới dạng giá trị loại thanh toán, bạn có thể chỉ định một hình thức thân thiện hơn với người dùng trong tệp ngôn ngữ `resources/lang/xx/validation.php` của mình bằng cách xác định mảng giá trị:

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

Sau khi xác định giá trị này, quy tắc giám định sẽ tạo ra thông báo lỗi sau:

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

Các rule trong giám định có sẵn

accepted

Field được giám định phải có giá trị là "`yes`", "`on`", `1` hoặc `true`. Điều này rất hữu ích để giám định việc chấp nhận "Điều khoản dịch vụ" hoặc các field tương tự.

accepted_if:anotherfield,value,...

Field được giám định phải có giá trị là "`yes`", "`on`", `1` hoặc `true` nếu một field khác đang được giám định bằng một giá trị đã chỉ định. Điều này rất hữu ích để xác thực việc chấp nhận "Điều khoản dịch vụ" hoặc các trường tương tự.

active_url

Field được giám định phải có record A hoặc AAAA hợp lệ theo hàm `dns_get_record` PHP. hostname của URL đã cung cấp được trích xuất bằng cách sử dụng hàm PHP `parse_url` trước khi được truyền đến `dns_get_record`.

after:date

Field được giám định phải là một giá trị sau một ngày nhất định. Ngày tháng sẽ được truyền vào hàm `strtotime` PHP để được chuyển đổi thành đối tượng `DateTime` hợp lệ:

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

Thay vì truyền một chuỗi date string (ví dụ: `tomorrow`) để được đánh giá bởi `strtotime`, bạn có thể chỉ định một field khác để so sánh với ngày:

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

after_or_equal:date

Field được giám định phải là một giá trị sau hoặc đang trong ngày đã chỉ định. Để biết thêm thông tin, hãy xem quy tắc giám định `after:date`.

alpha

Field được giám định phải hoàn toàn là các chữ cái.

alpha_dash

Field được giám định có thể có các ký tự chữ và số, cũng như dấu gạch ngang và dấu gạch dưới.

alpha_num

Field được giám định phải hoàn toàn là các ký tự chữ và số.

array

Field được giám định phải là một mảng PHP.

Khi các giá trị bổ sung được cung cấp cho quy tắc `array`, mỗi giá trị trong mảng đầu vào phải có trong danh sách đã cung cấp cho quy tắc. Trong ví dụ sau, danh sách có các giá trị là `username,locale`, như vậy `admin` trong mảng đầu vào không hợp lệ vì nó không có trong danh sách này khi cung cấp cho quy tắc `array`:

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

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

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

Nói chung, bạn phải luôn chỉ định các *array key* được phép hiện diện trong dữ liệu đầu vào của bạn. Nếu không, các phương thức `validate` và `validated` của `Validator` sẽ trả về tất cả dữ liệu đã được giám định, bao gồm mảng và tất cả các *key* của nó, ngay cả khi các khóa đó không được giám định bởi các quy tắc giám định mảng lồng nhau khác.

Nếu muốn, bạn có thể cho validator của Laravel không bao giờ đưa vào các *array key* chưa được giám định trong dữ liệu "validated" mà nó trả về, ngay cả khi bạn sử dụng quy tắc `array` mà không chỉ định danh sách các key được phép. Để thực hiện điều này, bạn có thể gọi phương thức `excludeUnvalidatedArrayKeys` của chương trình giám định trong phương thức `boot` của `AppServiceProvider` của ứng dụng của bạn. Sau khi làm như vậy, chương trình giám định sẽ đưa vào các *array key* trong dữ liệu "validated", nó chỉ trả về khi các *key* đã được giám định bởi các quy tắc `array` lồng nhau:

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

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

```
}
```

bail

Dừng chạy các quy tắc giám định cho field sau lần giám định đầu tiên không thành công.

Mặc dù quy tắc `bail` sẽ chỉ dừng giám định một trường cụ thể khi nó gặp lỗi giám định, nhưng phương thức `stopOnFirstFailure` sẽ thông báo cho chương trình giám định rằng nó sẽ ngừng giám định tất cả các *attribute* sau khi có một lỗi giám định xảy ra:

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

before:date

Field được giám định phải là một ngày trước ngày đã cho. Ngày tháng sẽ được truyền vào hàm `strtotime` của PHP để được chuyển đổi thành một đối tượng `DateTime` hợp lệ. Tương tự, giống như quy tắc `after`, tên của một field khác nào đó đang được giám định có thể được cho vào dưới dạng giá trị ngày tháng.

before_or_equal:date

Field được giám định phải là một giá trị trước hoặc đang trong ngày đã cho. Ngày tháng sẽ được truyền vào hàm `strtotime` của PHP để được chuyển đổi thành một đối tượng `DateTime` hợp lệ. Tương tự, giống như quy tắc `after`, tên của một field khác nào đó đang được giám định có thể được cho vào dưới dạng giá trị của ngày tháng.

between:min,max

Field được giám định phải có kích thước giữa giá trị tối thiểu (min) và tối đa (max) đã cho. String, numeric, array và tập tin sẽ được đánh giá theo cùng một hình thức với quy tắc `size`.

boolean

Field được giám định phải có thể được lưu truyền dưới dạng boolean. Đầu vào được chấp nhận là `true, false`, `1`, `0`, "`1`" và "`0`".

confirmed

Field được giám định phải có một field so sánh khớp là `{field}_confirmation`. Ví dụ: nếu field được giám định là `password`, thì field `password_confirmation` so sánh khớp cũng phải có trong dữ liệu đầu vào.

current_password

Field được giám định phải khớp với mật khẩu của người dùng đã xác thực. Bạn có thể chỉ định một chương trình bảo vệ xác thực bằng cách sử dụng tham số đầu tiên của quy tắc:

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

date

Field được giám định phải là một ngày hợp lệ, và không liên quan tới hàm `strtotime` PHP.

date_equals:date

Field được giám định phải đang trong ngày đã cho. Ngày tháng sẽ được truyền vào hàm `strtotime` của PHP để được chuyển đổi thành một đối tượng `DateTime` hợp lệ.

date_format:format

Field được giám định phải phù hợp với định dạng đã cho. Bạn nên sử dụng một trong hai quy tắc, `date` hoặc `date_format` khi giám định một field. Quy tắc giám định này hỗ trợ tất cả các định dạng được hỗ trợ bởi class `DateTime` của PHP.

declined

Field được giám định phải là "`no`", "`off`", `0` hoặc `false`.

declined_if:anotherfield,value,...

Field đang được giám định phải là "`no`", "`off`", `0` hoặc `false` nếu một field khác liên quan đang được giám định bằng một giá trị được chỉ định.

different:field

Field được giám định phải có giá trị khác với giá trị field đã cho.

digits:value

Field được giám định phải là field số và phải có độ lớn chính xác theo giá trị đã quy định.

digits_between:min,max

Field được giám định phải là số và phải có độ lớn nằm giữa giá trị tối thiểu và tối đa đã quy định.

dimensions

Tập tin được giám định phải là một hình ảnh đáp ứng các quy định về kích thước bởi các tham số của quy tắc:

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

Các ràng buộc có sẵn là: `min_width`, `max_width`, `min_height`, `max_height`, `width`, `height`, `ratio`.

Đồng bộ tỷ lệ phải được biểu diễn dưới dạng chiều rộng chia cho chiều cao. Điều này có thể được chỉ định với một phân số như `3/2` hoặc một `float` như `1.5`:

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

Vì quy tắc này yêu cầu một số đối số, nên bạn có thể sử dụng phương thức `Rule::dimensions` để xây dựng quy tắc một cách trôi chảy:

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

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

```
    ],
]);
```

distinct

Khi giám định mảng, field được giám định không được có bất kỳ giá trị trùng lặp nào:

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

distinct phân biệt biến rất lỏng lẻo. Để phân biệt chặt chẽ hơn, bạn có thể thêm thông số `strict` vào quy tắc giám định của mình:

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

Bạn có thể thêm `ignore_case` vào các đối số của quy tắc giám định để làm cho quy tắc bỏ qua sự phân biệt về chữ viết hoa:

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

email

Field được giám định phải được định dạng dưới dạng địa chỉ email. Quy tắc giám định này sử dụng thư viện *egulias/email-validator* để xác thực địa chỉ email. Theo mặc định, chương trình giám định `RFCValidation` sẽ được áp dụng, nhưng bạn cũng có thể áp dụng các kiểu giám định khác:

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

Ví dụ trên sẽ áp dụng chương trình giám định `RFCValidation` và `DNSCheckValidation`. Dưới đây là danh sách đầy đủ các kiểu giám định mà bạn có thể áp dụng:

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

- `filter: FilterEmailValidation`

Chương trình giám định `filter`, sử dụng hàm `filter_var` của PHP, đi kèm với Laravel và là chức năng giám định email mặc định của Laravel trước phiên bản Laravel 5.8.

> **Chú ý:** Các chương trình giám định `dns` và `spoof` yêu cầu PHP extension `intl`.

end_with: foo, bar, ...

Field được giám định phải kết thúc bằng một trong các giá trị đã quy định.

enum

Quy tắc `Enum` là quy tắc dựa trên class sẽ thẩm định xem field được giám định có chứa giá trị enum hợp lệ hay không. Quy tắc `Enum` chấp nhận tên của `enum` làm đối số cho constructor của nó:

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

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

> **Chú ý:** Enums chỉ có sẵn trên PHP 8.1+.

exclude

Field được giám định sẽ bị loại trừ khỏi dữ liệu request được trả về bởi các phương thức `validate` và `validated`.

exclude_if:anotherfield,value

Field được giám định sẽ bị loại trừ khỏi dữ liệu request được trả về bởi các phương thức `validate` và `validated` nếu field liên quan khác của field bằng giá trị đã quy định.

exclude_unless:anotherfield,value

Field được giám định sẽ bị loại trừ khỏi dữ liệu request do các phương thức `validate` và `validated` trả về ngoại trừ khi field liên quan khác bằng giá trị đã quy định. Nếu giá trị là `null` (ví dụ: `exclude_unless:name,null`), field được giám định sẽ bị loại trừ ngoại trừ khi field so sánh là `null` hoặc field so sánh bị thiếu (missing) trong dữ liệu request.

exclude_without:anotherfield

Field được giám định sẽ bị loại trừ khỏi dữ liệu request được trả về bởi các phương thức `validate` và `validated` nếu field liên qua khác không xuất hiện.

exists:table,column

Field được giám định phải tồn tại trong một bảng cơ sở dữ liệu nhất định.

Cách sử dụng cơ bản của quy tắc exist

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

Nếu tùy chọn `column` không được chỉ định, tên field sẽ được sử dụng thay thế. Vì vậy, trong trường hợp này, quy tắc sẽ xác nhận rằng *table* `states` chứa *record* có giá trị ở cột `state` khớp với giá trị mà field `state` đang nắm giữ.

Chỉ định tên cột cụ thể

Bạn có thể chỉ định rõ ràng tên *column* sẽ được sử dụng bởi quy tắc giám định bằng cách đặt nó sau tên *table*:

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

Đôi khi, bạn có thể cần chỉ định một kết nối *database* cụ thể sẽ được sử dụng cho quy tắc `exists`. Bạn có thể thực hiện điều này bằng cách thêm tên kết nối vào tên *table*:

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

Thay vì chỉ định trực tiếp tên *table*, bạn có thể chỉ định model Eloquent sẽ được sử dụng để xác định tên *table*:

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

Nếu bạn muốn điều chỉnh truy vấn được thực thi bởi quy tắc giám định, bạn có thể sử dụng class `Rule` để làm cho quy tắc giám định trôi chảy hơn. Trong ví dụ này, chúng tôi cũng sẽ quy định các quy tắc giám định dưới dạng một mảng thay vì sử dụng ký tự `|` để phân giới chúng:

```
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);
    }),
  ],
]);
```

file

Field được giám định phải là một tập tin được upload thành công.

filled

Field được giám định không được để trống khi nó tồn tại.

gt:field

Field được giám định phải lớn hơn field đã cho. Hai field phải cùng loại. String, numeric, array và tập tin được đánh giá bằng cách sử dụng các quy ước giống như quy tắc `size`.

gte:field

Field được giám định phải lớn hơn hoặc bằng field đã cho. Hai field phải cùng loại. String, numeric, array và tập tin được đánh giá bằng cách sử dụng các quy ước giống như quy tắc `size`.

image

Tập tin được giám định phải là hình ảnh (*jpg*, *jpeg*, *png*, *bmp*, *gif*, *svg* hoặc *webp*).

in:foo,bar,...

Field được giám định phải được bao gồm trong danh sách các giá trị đã cho. Vì quy tắc này thường yêu cầu bạn nhập một mảng, nên phương thức `Rule::in` có thể được sử dụng để xây dựng quy tắc một cách trôi chảy:

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

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

Khi quy tắc `in` được kết hợp với quy tắc `array`, mỗi giá trị trong mảng đầu vào phải có trong danh sách các giá trị được cung cấp cho quy tắc `in`. Trong ví dụ sau, mã sân bay `LAS` trong mảng đầu vào không hợp lệ vì nó không có trong danh sách các sân bay được cung cấp cho quy tắc `in`:

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

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

Validator::make($input, [
  'airports' => [
    'required',
    'array',
    Rule::in(['NYC', 'LIT']),
```

```
    ],
]);
```

in_array:anotherfield.*

Field được giám định phải tồn tại trong các giá trị của field khác.

integer

Field được giám định phải là một số nguyên.

> **Lưu ý:** Quy tắc giám định này không xác minh rằng đầu vào thuộc loại biến "integer", chỉ là đầu vào thuộc loại được chấp nhận bởi quy tắc `FILTER_VALIDATE_INT` của PHP. Nếu bạn cần xác thực đầu vào là một số, vui lòng sử dụng quy tắc này kết hợp với quy tắc `numeric`.

ip

Field được giám định phải là địa chỉ IP.

ipv4

Field được giám định phải là địa chỉ IPv4.

ipv6

Field được giám định phải là địa chỉ IPv6.

mac_address

Field được giám định phải là địa chỉ MAC.

json

Field được giám định phải là một chuỗi JSON hợp lệ.

lt:field

Field được giám định phải nhỏ hơn field đã cho. Hai field phải cùng loại. String, numeric, array và tập tin được đánh giá bằng cách sử dụng các quy ước giống như quy tắc `size`.

lte:field

Field được giám định phải nhỏ hơn hoặc bằng field đã cho. Hai field phải cùng loại. String, numeric, array và tập tin được đánh giá bằng cách sử dụng các quy ước giống như quy tắc `size`.

max:value

Field được giám định phải nhỏ hơn hoặc bằng giá trị `value`. String, numeric, array và tập tin được đánh giá theo cùng một quy ước với quy tắc `size`.

mimetypes:text/plain,...

Tập tin đang được giám định phải khớp với một trong các loại MIME đã quy định:

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

Để xác định kiểu MIME của tập tin được upload, nội dung của tập tin sẽ được đọc và framework sẽ cố gắng đoán kiểu MIME, có thể khác với kiểu MIME được cung cấp của client.

mimes:foo,bar,...

Tập tin đang được giám định phải có kiểu MIME tương ứng với một trong các extension đã được liệt kê.

Cách sử dụng cơ bản của quy tắc MIME

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

Mặc dù bạn chỉ cần chỉ định các extension, quy tắc này thực sự giám định kiểu MIME của tập tin bằng cách đọc nội dung của tập tin và đoán kiểu MIME của nó. Bạn có thể tìm thấy danh sách đầy đủ các loại MIME và các tiện ích mở rộng tương ứng của chúng tại địa chỉ sau:

https://svn.apache.org/repos/asf/httpd/httpd/trunk/docs/conf/mime.types

min:value

Field được giám định phải có giá trị nhỏ nhất. String, numeric, array và tập tin được đánh giá theo cùng một quy ước với quy tắc `size`.

multiple_of:value

Field được giám định phải là một bội số của giá trị.

> **Chú ý:** Extension PHP `bcmath` là bắt buộc để sử dụng quy tắc `multiple_of`.

not_in:foo,bar,...

Field được giám định không được bao gồm trong danh sách các giá trị đã cho. Phương thức `Rule::notIn` có thể được sử dụng để xây dựng quy tắc làm cho nó giám định một cách trôi chảy:

```
use Illuminate\Validation\Rule;

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

not_regex:pattern

Field được giám định không được khớp với biểu thức mẫu đã cho. Bên trong, quy tắc này sử dụng hàm `preg_match` trong PHP. Biểu thức mẫu được chỉ định phải tuân theo cùng một định dạng mà `preg_match` yêu cầu và do đó cũng bao gồm các dấu phân cách hợp lệ. Ví dụ: `'email' => 'not_regex: /^.+$/ i'`.

> **Chú ý:** Khi sử dụng các biểu thức mẫu `regex/not_regex`, có thể cần phải chỉ định các quy tắc giám định của bạn bằng cách sử dụng một mảng thay vì sử dụng dấu phân cách `|`, đặc biệt nếu biểu thức mẫu chứa dấu phân cách `|`.

nullable

Field được giám định có thể là trống.

numeric

Field được giám định phải là field số.

password

Field được giám định phải khớp với mật khẩu của người dùng đã xác thực.

> **Chú ý:** Quy tắc này đã được đổi tên thành `current_password` với ý định loại bỏ nó trong Laravel 9. Thay vào đó, hãy sử dụng quy tắc `current_password` để thay thế.

present

Field được giám định phải có trong dữ liệu đầu vào nhưng có thể để trống.

prohibited

Field được giám định phải trống hoặc không có.

prohibited_if:anotherfield,value,...

Field được giám định phải trống hoặc không có nếu field liên quan khác bằng bất kỳ giá trị nào.

prohibited_unless:anotherfield,value,...

Field được giám định phải trống hoặc không có trừ khi trường khác bằng bất kỳ giá trị nào.

prohibits:anotherfield,...

Nếu field được giám định có mặt, thì không có field nào trong `anotherfield` có thể có mặt, ngay cả khi giá trị của chúng trống.

regex:pattern

Field được giám định phải khớp với biểu thức mẫu đã cho. Bên trong, quy tắc này sử dụng hàm `preg_match` của PHP. Biểu thức mẫu được chỉ định phải tuân theo cùng một định dạng mà `preg_match` yêu cầu và do đó cũng bao gồm các dấu phân cách hợp lệ. Ví dụ: `'email' => 'regex: /^.+@.+$/ i'`.

> **Chú ý:** Khi sử dụng các mẫu `regex/not_regex`, có thể cần chỉ định các quy tắc trong một mảng thay vì sử dụng dấu phân cách `|`, đặc biệt nếu biểu thức mẫu chứa dấu phân cách `|`.

required

Field được giám định phải có trong dữ liệu đầu vào và không được để trống. Một field được coi là "empty" nếu một trong các điều kiện sau là `true`:

- Giá trị là null.
- Giá trị là một chuỗi rỗng.
- Giá trị là một mảng trống hoặc đối tượng có thể đếm được trống.
- Giá trị là một tập tin được upload không có đường dẫn.

required_if:anotherfield,value,...

Field được giám định phải có mặt và không được để trống nếu field liên quan khác bằng bất kỳ giá trị nào.

Nếu bạn muốn tạo một điều kiện phức tạp hơn cho quy tắc `require_if`, bạn có thể sử dụng phương thức `Rule::requireIf`. Phương thức này chấp nhận một boolean hoặc hàm nặc danh. Khi truyền vào một hàm nặc danh, thì hàm phải trả về `true` hoặc `false` để cho biết liệu field đang được giám định có bị bắt buộc hay không:

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

Field được giám định bị bắt buộc và không được để trống trừ khi field liên quan khác bằng bất kỳ giá trị nào. Điều này cũng có nghĩa là field liên quan khác phải có trong dữ liệu yêu cầu trừ khi giá trị là rỗng. Nếu giá trị là `null` (Ví dụ: `required_unless:name,null`), field được giám định sẽ được yêu cầu trừ khi field so sánh là `null` hoặc field so sánh bị thiếu trong dữ liệu request.

required_with:foo,bar,...

Field được giám định bị bắt buộc và không trống chỉ khi bất kỳ field nào trong số các field được liệt kê đang tồn tại và không trống.

required_with_all:foo,bar,...

Field được giám định bị bắt buộc và không trống chỉ khi tất cả các field được liệt kê đều có mặt và không trống.

required_without:foo,bar,...

Field được giám định bị bắt buộc và không được để trống chỉ khi bất kỳ trường nào được chỉ định khác trống hoặc không có.

required_without_all:foo,bar,...

Field được giám định bị bắt buộc và không được trống khi tất cả các field được liệt kê khác trống hoặc không có.

same:field

Field đã cho phải khớp với field đang được giám định.

size:value

Field được thẩm định phải có kích thước phù hợp với giá trị đã cho.

- Đối với dữ liệu chuỗi, giá trị tương ứng với số ký tự.
- Đối với dữ liệu số, giá trị tương ứng với một giá trị số nguyên quy định (thuộc tính cũng phải có quy tắc số hoặc số nguyên).
- Đối với một mảng, kích thước tương ứng với số lượng phần tử.
- Đối với tập tin, kích thước tương ứng với kích thước tập tin tính bằng kilobyte.

Hãy xem một số ví dụ:

```
// Validate that a string is exactly 12 characters long...
'title' => 'size:12';

// Validate that a provided integer equals 10...
'seats' => 'integer|size:10';

// Validate that an array has exactly 5 elements...
'tags' => 'array|size:5';

// Validate that an uploaded file is exactly 512 kilobytes...
'image' => 'file|size:512';
```

starts_with:foo,bar,...

Field được giám định phải bắt đầu bằng một trong các giá trị đã cho.

string

Field được giám định phải là một chuỗi. Nếu bạn muốn cho phép field cũng là `null`, thì bạn nên gán quy tắc `nullable` cho field.

timezone

Field được giám định phải là mã định danh múi giờ hợp lệ theo hàm `timezone_identifiers_list` của PHP.

unique:table,column

Field được giám định không được tồn tại trong bảng cơ sở dữ liệu đã cho.

Điều chỉnh tên table/column

Thay vì chỉ định trực tiếp tên bảng, bạn có thể chỉ định mô hình Eloquent sẽ được sử dụng để xác định tên bảng:

```
'email' => 'unique:App\Models\User,email_address'
```

Tùy chọn `column` có thể được sử dụng để chỉ định cột cơ sở dữ liệu tương ứng của field. Nếu tùy chọn `column` không được chỉ định, tên của field được giám định sẽ được sử dụng thay thế.

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

Điều chỉnh một kết nối cơ sở dữ liệu

Đôi khi, bạn có thể cần đặt kết nối điều chỉnh cho các truy vấn cơ sở dữ liệu do chương trình thẩm định thực hiện. Để thực hiện điều này, bạn có thể thêm tên kết nối vào tên bảng:

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

Ép buộc quy tắc unique bỏ qua một ID đã cho:

Đôi khi, bạn có thể muốn bỏ qua một ID nào đó trong quá trình giám định unique. Ví dụ: hãy xem xét màn hình "update profile" bao gồm tên, địa chỉ email và vị trí của người dùng. Bạn có thể sẽ muốn xác minh rằng địa chỉ email là duy nhất. Tuy nhiên, nếu người dùng chỉ thay đổi field name chứ không phải field email. Bạn không muốn xảy ra lỗi giám định vì người dùng đã là chủ sở hữu của địa chỉ email được đề cập.

Để khiến cho chương trình xác thực bỏ qua ID của người dùng, chúng tôi sẽ sử dụng class Rule để xác định quy tắc một cách thành thạo. Trong ví dụ này, chúng ta cũng sẽ chỉ định các quy tắc thẩm định dưới dạng một mảng thay vì sử dụng ký tự `|` để phân định các quy tắc:

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

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

> **Chú ý:** Bạn không bao giờ được truyền bất kỳ đầu vào do người dùng kiểm soát nào vào phương thức ignore. Thay vào đó, bạn chỉ nên truyền một ID duy nhất do hệ thống tạo ra, chẳng hạn như ID hoặc UUID tăng tự động từ một đối tượng model Eloquent. Nếu không, ứng dụng của bạn sẽ dễ bị tấn công SQL injection.

Thay vì truyền giá trị của *model key* cho phương thức `ignore`, bạn cũng có thể truyền toàn bộ đối tượng model. Laravel sẽ tự động trích xuất key từ mô hình:

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

Nếu bảng của bạn sử dụng tên cột nào đó làm *primary key* khác với `id`, bạn có thể chỉ định tên của cột khi gọi phương thức `ignore`:

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

Mặc định, quy tắc `unique` sẽ kiểm tra tính duy nhất của cột khớp với tên của *attribute* đang được giám định. Tuy nhiên, bạn có thể chuyển một tên cột khác làm đối số thứ hai cho phương thức `unique`:

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

Thêm quy ước bổ sung ở đâu:

Bạn có thể chỉ định các điều kiện truy vấn bổ sung bằng cách điều chỉnh truy vấn bằng phương thức `where`. Ví dụ: hãy thêm một điều kiện truy vấn phạm vi truy vấn để chỉ tìm

kiếm các *record* có giá trị cột `account_id` là `1`:

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

url

Field được giám định phải là một URL hợp lệ.

uuid

Field được giám định phải là mã định danh duy nhất RFC 4122 (phiên bản 1, 3, 4 hoặc 5) hợp lệ (UUID).

Bổ sung quy tắc có điều kiện

Bỏ qua giám định khi trường có giá trị nào đó

Đôi khi bạn có thể muốn không giám định một field nào đó nếu một field khác có giá trị nào đó. Bạn có thể thực hiện điều này bằng cách sử dụng quy tắc thẩm định `include_if`. Trong ví dụ này, các field `current_date` và `doctor_name` sẽ không được giám định nếu trường `has_appointment` có giá trị là `false`:

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

$validator = Validator::make($data, [
  'has_appointment' => 'required|boolean',
  'appointment_date' => 'exclude_if:has_appointment,false|required|date',
  'doctor_name' => 'exclude_if:has_appointment,false|required|string',
]);
```

Ngoài ra, bạn có thể sử dụng quy tắc `include_unless` để không giám định một field nào đó trừ khi một field liên quan khác có một giá trị nào đó:

```
$validator = Validator::make($data, [
```

```
  'has_appointment' => 'required|boolean',
  'appointment_date' => 'exclude_unless:has_appointment,true|required|date',
  'doctor_name' => 'exclude_unless:has_appointment,true|required|string',
]);
```

Giám định khi có mặt

Trong một số trường hợp, bạn có thể muốn chạy kiểm tra giám định đối với một field chỉ khi field đó có trong dữ liệu đang được giám định. Để nhanh chóng thực hiện điều này, hãy thêm quy tắc `sometimes` vào danh sách quy tắc của bạn:

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

Trong ví dụ trên, field email sẽ chỉ được giám định nếu nó có trong mảng `$data`.

> Nếu bạn đang cố gắng giám định một field luôn có nhưng có thể trống, hãy xem ghi chú này trên các field tùy chọn.

Giám định có điều kiện phức tạp

Đôi khi bạn có thể muốn thêm các quy tắc giám định dựa trên logic điều kiện phức tạp hơn. Ví dụ: bạn có thể chỉ muốn yêu cầu một field nhất định nếu một field khác có giá trị lớn hơn 100. Hoặc, bạn có thể cần hai field để có một giá trị nhất định chỉ khi có field khác. Việc thêm các quy tắc giám định này không phải là một vấn đề khó khăn. Đầu tiên, hãy tạo một đối tượng `Validator` với các quy tắc bất biến không bao giờ thay đổi của bạn:

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

$validator = Validator::make($request->all(), [
  'email' => 'required|email',
  'games' => 'required|numeric',
]);
```

Hãy giả sử ứng dụng web của chúng tôi là dành cho những người sưu tập trò chơi. Nếu

một người sưu tập trò chơi đăng ký với ứng dụng của chúng tôi và họ sở hữu hơn 100 trò chơi, chúng tôi muốn họ giải thích lý do tại sao họ sở hữu nhiều trò chơi như vậy. Ví dụ: có thể họ điều hành một cửa hàng bán lại trò chơi hoặc có thể họ chỉ thích sưu tập trò chơi. Để thêm yêu cầu này một cách có điều kiện, chúng ta có thể sử dụng phương thức `sometimes` trên đối tượng `Validator`.

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

Đối số đầu tiên được truyền cho phương thức `sometimes` là tên của field mà chúng ta đang giám định có điều kiện. Đối số thứ hai là danh sách các quy tắc mà chúng ta muốn thêm vào. Nếu một hàm được truyền dưới dạng đối số thứ ba mà trả về `true`, thì các quy tắc sẽ được thêm vào. Phương pháp này giúp bạn dễ dàng xây dựng các cuộc giám định có điều kiện phức tạp. Bạn thậm chí có thể thêm việc giám định có điều kiện cho một số field cùng một lúc:

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

> Tham số `$input` được truyền vào hàm `function` của bạn sẽ là một đối tượng của `Illuminate\Support\Fluent` và có thể được sử dụng để truy cập thông tin đầu vào và các tập tin của bạn mà đang được thẩm định.

Giám định mảng có điều kiện phức tạp

Đôi khi bạn có thể muốn giám định một field dựa trên một field khác trong cùng một mảng lồng nhau có chỉ mục (index) mà bạn không hề biết. Trong những trường hợp này, bạn có thể cho phép hàm của mình nhận được đối số thứ hai, đối số này sẽ là phần tử riêng lẻ hiện tại trong mảng đang được giám định:

```
$input = [
  'channels' => [
    [
      'type' => 'email',
```

```
      'address' => 'abigail@example.com',
    ],
    [
      'type' => 'url',
      'address' => 'https://example.com',
    ],
  ],
];

$validator->sometimes('channels.*.address', 'email', function ($input, $item) {
  return $item->type === 'email';
});

$validator->sometimes('channels.*.address', 'url', function ($input, $item) {
  return $item->type !== 'email';
});
```

Giống như tham số `$input` được truyền cho hàm, tham số `$item` là một đối tượng của `Illuminate\Support\Fluent` khi dữ liệu của *attribute* là một mảng; nếu không, nó là một chuỗi.

Giám định các mảng

Như đã thảo luận trong tài liệu quy tắc giám định `array`, quy tắc `array` chấp nhận danh sách các *array key* được phép. Nếu có bất kỳ *key* bổ sung nào trong mảng, quá trình giám định sẽ không thành công:

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

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

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

Nói chung, bạn phải luôn chỉ định các *array key* được phép hiện diện trong mảng của bạn. Nếu không, các phương thức `validate` và `validated` của `Validator` sẽ trả về tất cả dữ liệu đã được giám định, bao gồm mảng và tất cả các key của nó, ngay cả khi các key đó không được giám định bởi các quy tắc giám định mảng lồng nhau khác.

Loại bỏ các array key không được giám định

Nếu muốn, bạn có thể cho chương trình giám định của Laravel không bao giờ liệt kê các array key chưa được giám định trong dữ liệu "validated" mà nó trả về, ngay cả khi bạn sử dụng quy tắc `array` mà không chỉ định danh sách các key được phép. Để thực hiện điều này, bạn có thể gọi phương thức `excludeUnvalidatedArrayKeys` của chương trình giám định trong phương thức `boot` của `AppServiceProvider` của ứng dụng của bạn. Sau khi làm như vậy, chương trình giám định sẽ liệt kê các *array key* trong dữ liệu "validated", và nó chỉ trả về khi các *key* đã được giám định cụ thể bởi các quy tắc mảng lồng nhau:

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

/**
 * Register any application services.
 *
 * @return void
 */
public function boot()
{
  Validator::excludeUnvalidatedArrayKeys();
}
```

Giám định đầu vào có mảng lồng nhau

Việc giám định các form field có mảng lồng nhau không phải là điều khó khăn. Bạn có thể sử dụng ký hiệu dấu chấm (`.`) để giám định các *attribute* trong một mảng. Ví dụ: nếu HTTP request đến chứa field `photo[profile]`, thì bạn có thể giám định nó như sau:

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

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

Bạn cũng có thể giám định từng phần tử của một mảng. Ví dụ: để thẩm định rằng mỗi email trong một field nhập là mảng nào đó chỉ duy nhất, thì bạn có thể thực hiện như sau:

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

Tương tự như vậy, bạn có thể sử dụng ký tự * khi chỉ định thông báo giám định tự tạo trong tập tin ngôn ngữ của mình, nó sẽ giúp bạn dễ dàng sử dụng một thông báo giám định duy nhất cho các field mảng:

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

Giám định mật khẩu

Để đảm bảo rằng mật khẩu có mức độ phức tạp phù hợp, bạn có thể sử dụng đối tượng quy tắc `Password` của Laravel:

```php
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rules\Password;

$validator = Validator::make($request->all(), [
  'password' => ['required', 'confirmed', Password::min(8)],
]);
```

Đối tượng quy tắc `Password` cho phép bạn dễ dàng tùy chỉnh các yêu cầu về độ phức tạp của mật khẩu cho ứng dụng của mình, chẳng hạn như chỉ định rằng mật khẩu yêu cầu ít nhất một chữ cái, số, ký hiệu hoặc các ký tự có cách viết hoa-thường hỗn hợp:

```php
// Require at least 8 characters...
Password::min(8)

// Require at least one letter...
Password::min(8)->letters()

// Require at least one uppercase and one lowercase letter...
Password::min(8)->mixedCase()

// Require at least one number...
Password::min(8)->numbers()

// Require at least one symbol...
Password::min(8)->symbols()
```

Ngoài ra, bạn có thể đảm bảo rằng mật khẩu không bị xâm phạm trong vụ rò rỉ dữ liệu mật khẩu công khai bằng cách sử dụng phương thức `uncompromised`:

```php
Password::min(8)->uncompromised()
```

Trong nội bộ, đối tượng quy tắc `Password` sử dụng mô hình k-Anonymous để xác định xem mật khẩu có bị rò rỉ qua dịch vụ hasibeenpwned.com mà không ảnh hưởng đến quyền riêng tư hoặc bảo mật của người dùng hay không.

Mặc định, nếu mật khẩu xuất hiện ít nhất một lần trong một lần rò rỉ dữ liệu, mật khẩu đó

sẽ bị coi là bị xâm phạm. Bạn có thể điều chỉnh ngưỡng này bằng cách sử dụng đối số đầu tiên của phương thức `uncompromised`:

```
// Ensure the password appears less than 3 times in the same data leak...
Password::min(8)->uncompromised(3);
```

Tất nhiên, bạn có thể gọi lần lượt tất cả các phương thức trong các ví dụ trên:

```
Password::min(8)
  ->letters()
  ->mixedCase()
  ->numbers()
  ->symbols()
  ->uncompromised()
```

Khai báo các quy tắc mật khẩu mặc định

Bạn có thể thấy thuận tiện khi khai báo các quy tắc giám định mặc định cho mật khẩu ở một vị trí duy nhất trong ứng dụng của bạn. Bạn có thể dễ dàng thực hiện điều này bằng cách sử dụng phương thức `Password::defaults`, chấp nhận một hàm nặc danh. Hàm này được cung cấp cho phương thức `defaults` sẽ trả về cấu hình mặc định của quy tắc `Password`. Thông thường, quy tắc `defaults` phải được gọi trong phương thức `boot` của một trong các *service provider* của ứng dụng của bạn:

```
use Illuminate\Validation\Rules\Password;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
  Password::defaults(function () {
    $rule = Password::min(8);
```

```
    return $this->app->isProduction()
      ? $rule->mixedCase()->uncompromised()
      : $rule;
  });
}
```

Sau đó, khi bạn muốn áp dụng các quy tắc `defaults` cho một mật khẩu cụ thể đang được giám định, thì bạn có thể gọi phương thức `defaults` mà không có đối số:

```
'password' => ['required', Password::defaults()],
```

Đôi khi, bạn có thể muốn đính kèm các quy tắc giám định bổ sung vào các quy tắc giám định mật khẩu mặc định của mình. Bạn có thể sử dụng phương thức `rules` để thực hiện điều này:

```
use App\Rules\ZxcvbnRule;

Password::defaults(function () {
  $rule = Password::min(8)->rules([new ZxcvbnRule]);

  // ...
});
```

Tự tạo các quy tắc giám định

Sử dụng các đối tượng Rule

Laravel cung cấp nhiều quy tắc giám định hữu ích; tuy nhiên, bạn có thể muốn chỉ định một số với mục đích riêng bạn. Có một cách đăng ký quy tắc giám định tự tạo đó là sử dụng các đối tượng rule. Để tạo một đối tượng rule mới, bạn có thể sử dụng lệnh Artisan `make:rule`. Hãy sử dụng lệnh này để tạo quy tắc xác minh một chuỗi là chữ hoa. Laravel sẽ đặt quy tắc mới trong thư mục *app/Rules*. Nếu thư mục này không tồn tại, Laravel sẽ tạo nó khi bạn thực hiện lệnh Artisan để tạo quy tắc của bạn:

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

Khi quy tắc đã được tạo, chúng tôi đã sẵn sàng tạo chức năng của nó. Một đối tượng quy tắc chứa hai phương thức: `passes` và `message`. Phương thức `passes` nhận giá trị và tên *attribute* và phải trả về `true` hoặc `false` tùy thuộc vào việc giá trị *attribute* có hợp lệ hay không. Phương thức `message` sẽ trả về thông báo lỗi giám định sẽ được sử dụng khi giám định không thành công:

```php
<?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.';
  }
}
```

Bạn có thể gọi hàm toàn cục `trans` từ phương thức `message` của mình nếu bạn muốn trả lại thông báo lỗi từ các tập tin thông dịch của mình:

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

Khi quy tắc đã được xác định, bạn có thể đính kèm quy tắc đó vào chương trình giám định (validator) bằng cách truyền một đối tượng rule với các quy tắc giám định khác của bạn:

```
use App\Rules\Uppercase;

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

Truy cập vào dữ liệu bổ sung

Nếu class quy tắc giám định tự bạn tạo cần truy cập tất cả dữ liệu khác đang được giám định, thì class quy tắc của bạn có thể thực thi interface `Illuminate\Contracts\Validation\DataAwareRule`. Interface này yêu cầu class của bạn xác định một phương thức `setData`. Phương thức này sẽ tự động được gọi bởi Laravel (trước khi tiến hành giám định) với tất cả dữ liệu đang được giám định:

```
<?php
namespace App\Rules;
use Illuminate\Contracts\Validation\Rule;
use Illuminate\Contracts\Validation\DataAwareRule;
class Uppercase implements Rule, DataAwareRule
{
```

```
  /**
   * All of the data under validation.
   *
   * @var array
   */
  protected $data = [];

  // ...

  /**
   * Set the data under validation.
   *
   * @param  array  $data
   * @return $this
   */
  public function setData($data)
  {
    $this->data = $data;

    return $this;
  }
}
```

Hoặc, nếu quy tắc giám định của bạn yêu cầu quyền truy cập vào đối tượng `validator` mà đang thực hiện giám định, thì bạn có thể thực thi `ValidatorAwareRule` interface:

```
<?php
namespace App\Rules;
use Illuminate\Contracts\Validation\Rule;
use Illuminate\Contracts\Validation\ValidatorAwareRule;
class Uppercase implements Rule, ValidatorAwareRule
{
  /**
   * The validator instance.
   *
   * @var \Illuminate\Validation\Validator
   */
```

```
  protected $validator;

  // ...

  /**
   * Set the current validator.
   *
   * @param  \Illuminate\Validation\Validator  $validator
   * @return $this
   */
  public function setValidator($validator)
  {
    $this->validator = $validator;

    return $this;
  }
}
```

Sử dụng hàm nặc danh

Nếu bạn chỉ cần chức năng của quy tắc tự tạo có một lần trong toàn bộ ứng dụng của mình, thì bạn có thể sử dụng hàm nặc danh thay vì đối tượng quy tắc. Hàm này sẽ nhận được tên của *attribute,* giá trị của *attribute* và một callback trong biến `$fail` sẽ được gọi nếu quá trình giám định không thành công:

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

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

```
]);
```

Các quy tắc ngầm

Mặc định, khi một *attribute* đang được giám định không có mặt hoặc chứa một chuỗi trống, các quy tắc giám định thông thường, bao gồm cả các quy tắc tự tạo, sẽ không được chạy. Ví dụ: quy tắc `unique` sẽ không được chạy với một chuỗi trống:

```
use Illuminate\Support\Facades\Validator;
$rules = ['name' => 'unique:users,name'];
$input = ['name' => ''];
Validator::make($input, $rules)->passes(); // true
```

Để quy tắc tùy chỉnh chạy ngay cả khi *attribute* có giá trị trống, thì quy tắc phải ngầm định rằng attribute đó là bắt buộc (required). Để tạo quy tắc "implicit", hãy thực thi interface `Illuminate\Contracts\Validation\ImplicitRule`. Interface này chạy như một "maker interface" cho chương trình giám định; do đó, nó không chứa bất kỳ phương thức bổ sung nào bạn cần triển khai ngoài các phương thức được yêu cầu bởi interface Rule điển hình.

Để tạo một đối tượng quy tắc ngầm định mới, bạn có thể sử dụng lệnh Artisan `make:rule` với tùy chọn `--implicit`:

```
php artisan make:rule Uppercase --implicit
```

> **Chú ý:** Quy tắc "implicit" chỉ ngầm hiểu rằng *attribute* là bắt buộc (*required*). Việc nó có thực sự làm mất hiệu lực của một *attribute* bị thiếu hoặc trống hay không là tùy thuộc vào bạn.