Mastering Validation Rules in Laravel 12

Validation is one of the most important parts of any Laravel application. It ensures that the data coming from forms, APIs, or user input is safe, correct, and ready to be stored in the database. Without validation, users could submit broken data, leave required fields empty, or even try to exploit your app with malicious input.

In this guide, we'll go deep into mastering validation rules in Laravel 12. We'll look at basic validation, custom error messages, form request classes, and some of the most useful built-in rules. Each example is explained in detail so new developers can follow along easily.

1 – Basic Validation with the `validate()` Method

```php
// app/Http/Controllers/UserController.php

public function store(Request $request)
{
    $validated = $request->validate([
        'name'  => 'required|string|max:255',
        'email' => 'required|email|unique:users,email',
        'age'   => 'nullable|integer|min:18',
    ]);

    // At this point $validated contains only valid, safe data
    User::create($validated);

    return redirect()->back()->with('status', 'User created
successfully!');
}
```
Code language: PHP (php)

Here we use `$request->validate()` inside a controller method. Laravel automatically checks the input against the rules:

- `required` means the field must be present and not empty.
- `string` ensures the name is plain text, not an array or number.
- `max:255` limits the length of the name field to 255 characters.
- `email` ensures the input looks like an email address.
- `unique:users,email` checks the email doesn't already exist in the `users` table.
- `nullable|integer|min:18` means age is optional, but if provided it must be an integer at least 18.

If validation fails, Laravel will automatically redirect back to the previous page with errors stored in the session. You can display them in Blade using `@error`.

2 - Displaying Validation Errors in Blade

```
// resources/views/users/create.blade.php

<form method="POST" action="{{ route('users.store') }}" class="card
card-body">
    @csrf

    <input type="text" name="name" placeholder="Name" value="{{
old('name') }}" class="form-control mb-2">
    @error('name')
        <div class="text-danger">{{ $message }}</div>
    @enderror

    <input type="email" name="email" placeholder="Email" value="{{
old('email') }}" class="form-control mb-2">
    @error('email')
```

```
        <div class="text-danger">{{ $message }}</div>
    @enderror

    <input type="number" name="age" placeholder="Age" value="{{
old('age') }}" class="form-control mb-2">
    @error('age')
        <div class="text-danger">{{ $message }}</div>
    @enderror

    <button class="btn btn-primary">Create User</button>
</form>Code language: PHP (php)
```

The `old()` helper keeps the previous value if the form fails validation. The `@error` directive shows the error message for each field if validation fails.

3 - Commonly Used Validation Rules

- `required` — Field must be present and not empty.
- `nullable` — Field can be empty, but if present, it must follow other rules.
- `email` — Must be a valid email format.
- `url` — Must be a valid URL.
- `min:n` / `max:n` — Minimum or maximum length/size.
- `numeric` / `integer` — Value must be a number.
- `date` — Must be a valid date (Y-m-d, etc.).
- `confirmed` — Requires a matching `_confirmation` field (commonly used for passwords).
- `unique:table,column` — Ensures no duplicates in the database.
- `exists:table,column` — Ensures the value exists in another table (useful for foreign keys).

Laravel provides dozens of validation rules out of the box. These rules are simple strings, but you can also pass arrays if you need more flexibility.

4 – Custom Error Messages

```
// app/Http/Controllers/UserController.php

public function store(Request $request)
{
    $messages = [
        'name.required' => 'Please enter your full name.',
        'email.required' => 'We need your email address.',
        'email.unique'  => 'That email is already taken, please choose
another.',
    ];

    $validated = $request->validate([
        'name'  => 'required|string|max:255',
        'email' => 'required|email|unique:users,email',
        'age'   => 'nullable|integer|min:18',
    ], $messages);

    User::create($validated);

    return redirect()->back()->with('status', 'User created
successfully!');
}Code language: PHP (php)
```

Here we pass a second argument to `validate()` with an array of messages. This overrides Laravel's default messages with custom, user-friendly text.

5 – Using Form Request Classes

```
php artisan make:request StoreUserRequestCode language: CSS (css)
```

This command creates a new class in `app/Http/Requests/StoreUserRequest.php`. Inside it, you define validation rules and messages:

```
namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class StoreUserRequest extends FormRequest
{
    public function authorize(): bool
    {
        return true; // allow all for now
    }

    public function rules(): array
    {
        return [
            'name'  => 'required|string|max:255',
            'email' => 'required|email|unique:users,email',
            'age'   => 'nullable|integer|min:18',
        ];
    }

    public function messages(): array
    {
        return [
            'name.required' => 'Your name is required.',
            'email.unique'  => 'This email is already registered.',
        ];
    }
}Code language: PHP (php)
```

Form requests separate validation logic from controllers. You can now type-hint this request class in your controller:

```
// app/Http/Controllers/UserController.php

public function store(StoreUserRequest $request)
{
    User::create($request->validated());
    return redirect()->back()->with('status', 'User created!');
}Code language: PHP (php)
```

This makes your controllers cleaner and your validation rules reusable across multiple places.

6 – Conditional Validation

```
// Example: phone is required only if contact_method = phone

$request->validate([
    'contact_method' => 'required|in:email,phone',
    'email' => 'required_if:contact_method,email|email',
    'phone' => 'required_if:contact_method,phone|digits:10',
]);Code language: PHP (php)
```

Here, `required_if` makes the `email` field required if `contact_method` is set to “email”, and the `phone` field required if “phone” is selected. This gives you dynamic validation depending on user choices.

7 – Custom Validation Rules

```
php artisan make:rule UppercaseCode language: CSS (css)
```

This generates a rule class. Inside you define your custom logic:

```
// app/Rules/Uppercase.php

namespace App\Rules;

use Closure;
use Illuminate\Contracts\Validation\ValidationRule;

class Uppercase implements ValidationRule
{
    public function validate(string $attribute, mixed $value, Closure
$fail): void
    {
        if (strtoupper($value) !== $value) {
            $fail('The :attribute must be uppercase.');
        }
    }
}Code language: PHP (php)
```

Now you can use it like this:

```
$request->validate([
    'code' => ['required', new \App\Rules\Uppercase],
]);Code language: PHP (php)
```

This ensures the `code` field is always uppercase. Custom rules let you handle business logic that built-in rules don't cover.

Wrapping Up

You now have a strong understanding of validation in Laravel 12: from basic rules and error messages to form requests and custom validators. Validation is a core part of keeping your app secure and reliable, and Laravel makes it both powerful and easy to use.

As you build bigger apps, you'll often combine multiple rules, create reusable form request classes, and even add custom validation rules specific to your project's business logic.

Next Steps

Once you're comfortable with the basics of Laravel validation, here are some advanced topics you can explore to take your skills further:

- **File Upload Validation:** Learn to validate file types, sizes, and dimensions (e.g., images, PDFs).
- **Regex Rules:** Use regular expressions for highly customized validation logic.
- **Sanitizing Input:** Combine validation with data cleaning (like trimming whitespace, escaping HTML).
- **Cross-Field Validation:** Write rules that compare multiple fields (e.g., start_date must be before end_date).
- **API Validation:** Apply validation rules in API controllers and return JSON error responses.
- **Localization:** Provide validation error messages in multiple languages for

international apps.

These next steps will help you build more robust, secure, and user-friendly applications where data is always validated properly before saving.