Creating Custom Validation Rules in Laravel

Laravel ships with a wide variety of validation rules, but sometimes your application requires domain-specific validation that doesn’t exist out-of-the-box. Custom validation rules allow you to encapsulate business logic in a clean, reusable way. In this guide, you’ll learn multiple approaches: using closures, creating dedicated Rule classes, and leveraging dependency injection. We’ll also integrate custom rules into Blade forms, controllers, and feature tests to make sure everything is robust in production.

Approach 1: Inline Custom Rules with Closures

For simple, one-off validations you can define a closure directly inside a `FormRequest` or controller. The closure receives the attribute, value, and a fail callback.

```
// app/Http/Requests/RegisterUserRequest.php
namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class RegisterUserRequest extends FormRequest
{
    public function authorize(): bool { return true; }

    public function rules(): array
    {
        return [
            'username' => [
                'required',
                function ($attribute, $value, $fail) {
                    if (str_contains(strtolower($value), 'admin')) {
                        $fail('The '.$attribute.' may not contain
"admin".');
                    }
                }
            ],
        ];
    }
}
```
Code language: PHP (php)

This prevents reserved words from being used as usernames. While closures are quick, they can’t be reused across multiple forms, so prefer dedicated rules for shared logic.

Approach 2: Custom Rule Classes

Use Artisan to scaffold a new rule class:

```
php artisan make:rule StrongPassword
```
Code language: Bash (bash)

This generates `app/Rules/StrongPassword.php`:

```
namespace App\Rules;

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

class StrongPassword implements ValidationRule
{
    public function validate(string $attribute, mixed $value, Closure
$fail): void
    {
        if (strlen($value) < 8
            || !preg_match('/[A-Z]/', $value)
            || !preg_match('/[0-9]/', $value)) {
            $fail('The '.$attribute.' must be at least 8 characters
and contain an uppercase letter and number.');
        }
    }
}Code language: PHP (php)
```

Now apply it in a request:

```
// app/Http/Requests/RegisterUserRequest.php
use App\Rules\StrongPassword;

public function rules(): array
{
    return [
        'password' => ['required', new StrongPassword],
    ];
}Code language: PHP (php)
```

Custom rule classes encapsulate logic in reusable components. They can be unit tested directly, keeping validation logic isolated and clean.

Approach 3: Dependency Injection in Rules

Rules can access services via constructor injection. This is powerful for validating against external APIs or domain services.

```
namespace App\Rules;

use App\Services\EmailBlacklistService;
use Closure;
use Illuminate\Contracts\Validation\ValidationRule;

class NotBlacklistedEmail implements ValidationRule
{
    public function __construct(protected EmailBlacklistService
$service) {}

    public function validate(string $attribute, mixed $value, Closure
$fail): void
    {
        if ($this->service->isBlacklisted($value)) {
            $fail('The email address is not allowed.');
        }
    }
}
```
Code language: PHP (php)

Bind the service in the container, then Laravel will inject it when instantiating the rule. This pattern is ideal for business-critical checks like fraud prevention.

Blade Form Integration

```
<form method="POST" action="{{ route('register') }}">
    @csrf
    <label>Username</label>
    <input name="username" value="{{ old('username') }}" />
    @error('username') <div class="error">{{ $message }}</div>
@enderror

    <label>Password</label>
    <input type="password" name="password" />
    @error('password') <div class="error">{{ $message }}</div>
@enderror

    <button type="submit">Register</button>
</form>
```
Code language: PHP (php)

Custom validation rules behave exactly like built-in ones. Errors are displayed in Blade using `@error` blocks, and old values are preserved across submissions.

Testing Custom Rules

Custom rules can be tested in isolation or via full feature tests. Here's a unit test for the `StrongPassword` rule:

```php
// tests/Unit/StrongPasswordTest.php
use App\Rules\StrongPassword;
use Illuminate\Support\Facades\Validator;

test('validates strong password', function () {
    $rule = new StrongPassword();

    $v = Validator::make(['password' => 'Weak'], ['password' =>
[$rule]]);
    expect($v->fails())->toBeTrue();

    $v = Validator::make(['password' => 'StrongPass1'], ['password' =>
[$rule]]);
    expect($v->passes())->toBeTrue();
});Code language: PHP (php)
```

This test ensures the rule works with both failing and passing values. For request-level validation, write `Feature` tests that submit forms and assert validation errors appear.

Best Practices for Custom Validation

- Use **closures** for simple, one-off rules.
- Use **dedicated rule classes** for reusable or complex rules.
- Inject services into rules for external checks (databases, APIs, business rules).
- Always **unit test** your custom rules.
- Provide clear, user-friendly error messages.
- Avoid overloading rules; keep them focused on one responsibility.

Wrapping Up

Custom validation rules let you enforce domain-specific constraints in a clean and reusable way. We explored closures, dedicated rule classes, and injected rules. With proper Blade integration, user-friendly messages, and thorough testing, your forms will be both flexible and reliable.

What’s Next

Continue learning about form handling and validation in Laravel:

- Mastering Validation Rules in Laravel 12
- Building a Multi-Step Form Wizard in Laravel
- Handling File Uploads and Image Storage in Laravel