Creating Custom Form Request Validation with Dynamic Rules and Messages

Laravel's Form Request validation provides a powerful way to organize your validation logic. Here's how to create dynamic validation with customizable rules and error messages.

1. Generate a Form Request

First, create a custom Form Request class:

```
php artisan make:request DynamicFormRequest
```

2. Implement Dynamic Rules

Modify the generated file to include dynamic rules:

```php
namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;
use Illuminate\Validation\Rule;

class DynamicFormRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     */
    public function authorize(): bool
    {
        return true; // Or add your authorization logic
    }

    /**
     * Get the validation rules that apply to the request.
     */
    public function rules(): array
    {
        $baseRules = [
            'email' => ['required', 'email', 'max:255'],
            'name' => ['required', 'string', 'max:100'],
        ];

        // Add conditional rules
        if ($this->has('update_profile')) {
            $baseRules['avatar'] = ['nullable', 'image', 'max:2048'];
            $baseRules['bio'] = ['nullable', 'string', 'max:500'];
        }
```

```
        // Dynamic role-based rules
        if ($this->user()->isAdmin()) {
            $baseRules['role'] = ['required', Rule::in(['admin',
'editor', 'user'])];
        }

        // Merge with any additional rules from the request
        return array_merge($baseRules, $this->additionalRules());
    }

    /**
     * Generate additional rules based on request data
     */
    protected function additionalRules(): array
    {
        $additional = [];

        // Example: Add validation for dynamic fields
        if ($this->has('custom_fields')) {
            foreach ($this->input('custom_fields') as $field => $value)
{
                $additional["custom_fields.{$field}"] = $this-
>getFieldRule($field);
            }
        }

        return $additional;
    }

    /**
     * Get validation rule for a specific dynamic field
     */
    protected function getFieldRule(string $field): array
    {
        // You might get these from a database or config
        $fieldRules = [
            'age' => ['numeric', 'min:18', 'max:120'],
            'website' => ['nullable', 'url'],
            'phone' => ['required', 'regex:/^\+?[0-9]{10,15}$/'],
        ];

        return $fieldRules[$field] ?? ['string', 'max:255'];
    }
}
```

PROF

3. Customize Error Messages

Add dynamic error messages to the same class:

```php
/**
 * Get custom error messages for validator errors.
 */
public function messages(): array
{
    return [
        // General messages
        'required' => 'The :attribute field is required.',
        'email' => 'The :attribute must be a valid email address.',

        // Field-specific messages
        'name.required' => 'Please provide your full name.',
        'name.max' => 'Your name cannot exceed 100 characters.',

        // Dynamic field messages
        'custom_fields.*.numeric' => 'The :attribute must be a number.',
        'custom_fields.age.min' => 'You must be at least 18 years old.',

        // Conditional messages
        'role.required' => $this->user()->isAdmin()
            ? 'Please select a user role.'
            : 'You cannot set roles.',
    ];
}
```

4. Add Custom Attributes

Improve the display names of fields in error messages:

```php
/**
 * Get custom attributes for validator errors.
 */
public function attributes(): array
{
    $attributes = [
        'email' => 'email address',
        'custom_fields.age' => 'age',
    ];

    // Add dynamic field attributes
    if ($this->has('custom_fields')) {
        foreach (array_keys($this->input('custom_fields')) as $field) {
            $attributes["custom_fields.{$field}"] = str_replace('_', '
', $field);
        }
    }

    return $attributes;
}
```

5. Using the Form Request in a Controller

Here's how to use your dynamic form request:

```
namespace App\Http\Controllers;

use App\Http\Requests\DynamicFormRequest;

class UserController extends Controller
{
    public function updateProfile(DynamicFormRequest $request)
    {
        // The request is already validated at this point

        $validated = $request->validated();

        // Process the validated data
        auth()->user()->update($validated);

        return response()->json([
            'message' => 'Profile updated successfully',
            'data' => $validated,
        ]);
    }
}
```

6. Advanced: Dynamic Rules Based on Database

For rules that depend on database values:

PROF

```
protected function additionalRules(): array
{
    $additional = [];

    // Get validation rules from database configuration
    $validationConfig = \App\Models\ValidationConfig::where('model',
'User')->get();

    foreach ($validationConfig as $config) {
        $additional[$config->field_name] = explode('|', $config-
>validation_rules);
    }

    return $additional;
}
```

7. Handling Array Validation

For complex array data validation:

```
public function rules(): array
{
    return [
        'products' => ['required', 'array'],
        'products.*.id' => ['required', 'exists:products,id'],
        'products.*.quantity' => ['required', 'integer', 'min:1'],
        'products.*.options' => ['sometimes', 'array'],
        'products.*.options.*' => ['string', 'max:50'],
    ];
}

public function messages(): array
{
    return [
        'products.*.id.exists' => 'One or more products are invalid.',
        'products.*.quantity.min' => 'Quantity must be at least 1 for
all items.',
    ];
}
```

8. Testing Your Dynamic Validation

Create tests to verify your validation logic:

```
namespace Tests\Feature\Http\Requests;

use App\Http\Requests\DynamicFormRequest;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Support\Facades\Validator;
use Tests\TestCase;

class DynamicFormRequestTest extends TestCase
{
    use RefreshDatabase;

    public function test_validation_rules_are_dynamic()
    {
        $request = new DynamicFormRequest();

        // Simulate request with custom fields
        $request->merge([
            'email' => 'test@example.com',
            'name' => 'John Doe',
            'custom_fields' => [
                'age' => 25,
```

PROF

```
                'website' => 'https://example.com'
            ]
        ]);

        $validator = Validator::make(
            $request->all(),
            $request->rules(),
            $request->messages(),
            $request->attributes()
        );

        $this->assertFalse($validator->fails());
    }

    public function test_admin_role_validation()
    {
        $admin = \App\Models\User::factory()->admin()->create();
        $this->actingAs($admin);

        $request = new DynamicFormRequest();
        $request->merge([
            'email' => 'admin@example.com',
            'name' => 'Admin User',
            'role' => 'editor'
        ]);

        $rules = $request->rules();

        $this->assertArrayHasKey('role', $rules);
        $this->assertContains('required', $rules['role']);
    }
}
```

9. Additional Tips

PROF

1. **Rule Objects**: Create custom rule objects for complex validation logic
2. **After Hooks**: Use `withValidator` to add after-validation logic
3. **Prepare for Validation**: Use `prepareForValidation` to modify data before validation

Example with `withValidator`:

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

This implementation gives you complete control over your validation logic while keeping it organized and maintainable. The dynamic nature allows you to adapt to different scenarios without creating multiple request classes.