Introduction

This document outlines the structure and best practices for defining Request classes in the Laravel backend code. The goal is to ensure that Request classes are well-structured, maintainable, and adhere to best practices by clearly separating validation, authorization, and data manipulation concerns.

General Guidelines

Validation Rules: Define validation rules using the rules method.

Bail Rule: Use the bail rule when needed to stop running validation rules on an attribute after the first validation failure.

Custom Rules: Create custom validation rules for complex validation logic.

Data Preparation: Adjust request data before validation using the prepareForValidation method.

Layered Validation: Add second or third layers of validation using the withValidator method.

Post-Validation Processing: Update or append data to the request after validation using the passedValidation method.

Authorization: Check authorization by verifying policies or gates in the authorize method.

Naming Conventions

Clear and consistent naming conventions help maintain a clean codebase and make it easier to understand the purpose of each Request class. Follow these guidelines for naming your Request classes:

Use a Descriptive Name: The class name should clearly describe the action being requested.

Use PascalCase: Capitalize the first letter of each word in the class name.

Include the Entity and Action: Include both the entity and the action in the class name to make it clear what the request is for.

Examples:

StorePolicyRequest: For handling the creation of a policy.

UpdatePolicyRequest: For handling the updating of a policy.

DeletePolicyRequest: For handling the deletion of a policy.

ApproveClaimRequest: For handling the approval of a claim.

RejectClaimRequest: For handling the rejection of a claim.

Edge Cases:

Bulk Actions: Use plural form for actions that affect multiple entities.

Example: BulkDeletePoliciesRequest, BulkApproveClaimsRequest

Complex Actions: Combine multiple actions or steps in the name if the request involves a complex process.

Example: SubmitClaimWithDocumentsRequest, RenewPolicyWithPaymentRequest

Nested Resources: Include both resources in the name if the request is for a nested resource.

Example: UpdatePolicyClaimRequest, StorePolicyCommentRequest

Structure of a Request Class

Here is a template and example of a well-structured Request class in Laravel:

<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

use Illuminate\Validation\Rule;

class ClaimRequest extends FormRequest

{

/\*\*

\* Determine if the user is authorized to make this request.

\*

\* @return bool

\*/

public function authorize()

{

return $this->user()->can('create', Claim::class);

}

/\*\*

\* Prepare the data for validation.

\*/

protected function prepareForValidation()

{

$this->merge([

'claim\_amount' => number\_format($this->claim\_amount, 2),

]);

}

/\*\*

\* Get the validation rules that apply to the request.

\*

\* @return array

\*/

public function rules()

{

return [

'policy\_id' => ['required', 'exists:policies,id'],

'claim\_amount' => ['required', 'numeric', 'min:0', 'bail'],

'status' => ['required', 'in:pending,approved,rejected'],

'description' => ['nullable', 'string'],

];

}

/\*\*

\* Configure the validator instance.

\*

\* @param \Illuminate\Validation\Validator $validator

\* @return void

\*/

public function withValidator($validator)

{

if ($validator->fails()) {

return;

}

$validator->after(function ($validator) {

if ($this->claim\_amount && $this->claim\_amount < 0) {

$validator->errors()->add('claim\_amount', trans('validation.custom.positive'));

}

});

}

/\*\*

\* Handle a passed validation attempt.

\*

\* @return void

\*/

protected function passedValidation()

{

$this->merge([

'processed\_at' => now(),

]);

}

}

Detailed Explanation

Authorization

Use the authorize method to check if the user is authorized to make this request. This is typically done by checking a policy or gate.

public function authorize()

{

return $this->user()->can('create', Claim::class);

}

Data Preparation

Use the prepareForValidation method to adjust or manipulate request data before the validation rules are applied.

protected function prepareForValidation()

{

$this->merge([

'claim\_amount' => number\_format($this->claim\_amount, 2),

]);

}

Validation Rules

Define the validation rules that apply to the request in the rules method. Use the bail rule when needed to stop running validation rules on an attribute after the first validation failure.

public function rules()

{

return [

'policy\_id' => ['required', 'exists:policies,id'],

'claim\_amount' => ['required', 'numeric', 'min:0', 'bail'],

'status' => ['required', 'in:pending,approved,rejected'],

'description' => ['nullable', 'string'],

];

}

Custom Validation Rules

For complex validation logic, create custom validation rules by extending Laravel's validation system.

Example Custom Rule:

<?php

namespace App\Rules;

use Illuminate\Contracts\Validation\Rule;

class PositiveAmount implements Rule

{

public function passes($attribute, $value)

{

return $value > 0;

}

public function message()

{

return trans('validation.custom.positive\_amount');

}

}

Using the Custom Rule in a Request:

'claim\_amount' => ['required', 'numeric', 'min:0', 'bail', new PositiveAmount()],

Layered Validation

Use the withValidator method to add additional validation logic that should run after the initial validation rules have been applied. This is useful when another layer of validation needs to be executed after the first one or it relies on some data from the previous validation.

public function withValidator($validator)

{

if ($validator->fails()) {

return;

}

$validator->after(function ($validator) {

if ($this->claim\_amount && $this->claim\_amount < 0) {

$validator->errors()->add('claim\_amount', trans('validation.custom.positive\_amount'));

}

});

// Example for checking Rule::exists() or Rule::unique() for an array of data

$validator->after(function ($validator) {

$emails = $this->input('emails');

$existingEmails = \App\Models\User::whereIn('email', $emails)->pluck('email')->toArray();

foreach ($emails as $i => $email) {

$validator->errors()->add("emails.$i", trans('validation.unique', ['attribute' => 'email']));

}

});

}

Post-Validation Processing

Use the passedValidation method to manipulate the request data after it has been validated but before it is used in the controller or service.

protected function passedValidation()

{

$this->merge([

'processed\_at' => now(),

]);

}

What to Avoid

To maintain clean, maintainable, and well-structured Request classes, follow these guidelines on what to avoid:

Avoid Business Logic in Requests: Request classes should primarily handle validation and authorization. Use Service classes for business logic.

Incorrect:

public function authorize()

{

return $this->user()->role == 'admin';

}

Correct:

public function authorize()

{

return $this->user()->can('create', Claim::class);

}

Avoid Hardcoding Validation Messages in Code: Localize validation messages and attribute names in the validation.php language file, rather than hardcoding them in the code. This improves maintainability and supports localization.

Incorrect:

public function withValidator($validator)

{

$validator->after(function ($validator) {

if ($this->claim\_amount && $this->claim\_amount < 0) {

$validator->errors()->add('claim\_amount', 'Claim amount must be positive.');

}

});

}

Correct:

// In the validation.php language file

'custom' => [

'claim\_amount' => [

'positive\_amount' => 'Claim amount must be positive.',

],

],

// In the Request class

public function withValidator($validator)

{

$validator->after(function ($validator) {

if ($this->claim\_amount && $this->claim\_amount < 0) {

$validator->errors()->add('claim\_amount', \_\_('validation.custom.claim\_amount.positive\_amount'));

}

});

}