Introduction

This document outlines the structure and best practices for defining controller classes in the Laravel backend code. The goal is to ensure that controllers are well-structured, maintainable, and adhere to best practices by separating concerns such as validation, authorization, business logic, and data transformation.

General Guidelines

CRUD Methods: Controllers should primarily handle CRUD operations and routing. Method names should follow the camelCase convention.

Validation: Use Form Request classes to handle validation logic. Form Request class names should follow the PascalCase convention.

Authorization: Use Policy classes to handle authorization logic. Policy class names should follow the PascalCase convention.

Business Logic: Delegate business logic to Service classes. Service class names should follow the PascalCase convention.

Data Transformation: Use Resource classes to transform the data before sending it to the client. Resource class names should follow the PascalCase convention.

Naming Conventions

Adhering to a consistent naming convention is crucial for maintaining a clean and understandable codebase.

Controller Class Names: Controller class names should be in PascalCase and should follow the naming pattern [Model]Controller. For example, a controller handling operations for the Claim model should be named ClaimController.

Method Names: Methods within the controller should follow the camelCase convention and be named to clearly describe their action. Common CRUD methods include index (for listing resources), store (for creating a new resource), show (for displaying a single resource), update (for updating a resource), and destroy (for deleting a resource).

Handling Multiple Entities: When dealing with multiple entities, the controller name should clearly reflect the relationship or the combined purpose. Use descriptive and concise names to indicate the involved entities and their interaction. For instance:

If the controller manages the relationship between User and Profile, name it UserProfileController.

For controllers handling complex operations involving multiple models, use a name that describes the overall functionality, such as OrderInvoiceController for operations involving both Order and Invoice.

Single Responsibility Principle

By following these guidelines, controllers will adhere to the Single Responsibility Principle. Their single responsibility will be to act as intermediaries or proxies between the client (HTTP requests) and the business logic, which is handled by Service classes. This keeps controllers focused on routing and delegation, making them easier to maintain and test.

Structure of a Controller

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

<?php

namespace App\Http\Controllers;

use App\Http\Requests\ClaimRequest;

use App\Http\Resources\ClaimResource;

use App\Models\Claim;

use App\Services\ClaimService;

use Illuminate\Http\Request;

class ClaimController extends Controller

{

protected $claimService;

public function \_\_construct(ClaimService $claimService)

{

$this->claimService = $claimService;

}

/\*\*

\* Display a listing of the resource.

\*

\* @return \Illuminate\Http\Response

\*/

public function index(Request $request)

{

$this->authorize('viewAny', Claim::class);

$claims = $this->claimService->getAllClaims($request);

return ClaimResource::collection($claims);

}

/\*\*

\* Store a newly created resource in storage.

\*

\* @param \App\Http\Requests\ClaimRequest $request

\* @return \Illuminate\Http\Response

\*/

public function store(ClaimRequest $request)

{

$this->authorize('create', Claim::class);

$claim = $this->claimService->createClaim($request->validated());

return new ClaimResource($claim);

}

/\*\*

\* Display the specified resource.

\*

\* @param \App\Models\Claim $claim

\* @return \Illuminate\Http\Response

\*/

public function show(Claim $claim)

{

$this->authorize('view', $claim);

return new ClaimResource($claim);

}

/\*\*

\* Update the specified resource in storage.

\*

\* @param \App\Http\Requests\ClaimRequest $request

\* @param \App\Models\Claim $claim

\* @return \Illuminate\Http\Response

\*/

public function update(ClaimRequest $request, Claim $claim)

{

$this->authorize('update', $claim);

$claim = $this->claimService->updateClaim($claim, $request->validated());

return new ClaimResource($claim);

}

/\*\*

\* Remove the specified resource from storage.

\*

\* @param \App\Models\Claim $claim

\* @return \Illuminate\Http\Response

\*/

public function destroy(Claim $claim)

{

$this->authorize('delete', $claim);

$this->claimService->deleteClaim($claim);

return response()->noContent();

}

// Other actions...

}

Detailed Explanation

CRUD Methods

Define the basic CRUD methods (index, store, show, update, destroy) in your controller. These methods should handle routing and delegation.

Validation

Use Form Request classes to handle validation logic. This keeps your controllers clean and ensures validation logic is reusable.

Example Form Request:

<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class ClaimRequest extends FormRequest

{

public function authorize()

{

return true; // or handle authorization here

}

public function rules()

{

return [

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

'claim\_amount' => 'required|numeric|min:0',

'status' => 'required|in:pending,approved,rejected',

// other rules...

];

}

}

Authorization

Use Policy classes to handle authorization logic. This ensures that authorization is handled consistently and keeps your controllers clean.

Example Policy:

<?php

namespace App\Policies;

use App\Models\User;

use App\Models\Claim;

use Illuminate\Auth\Access\HandlesAuthorization;

class ClaimPolicy

{

use HandlesAuthorization;

public function viewAny(User $user)

{

// Authorization logic...

}

public function view(User $user, Claim $claim)

{

// Authorization logic...

}

public function create(User $user)

{

// Authorization logic...

}

public function update(User $user, Claim $claim)

{

// Authorization logic...

}

public function delete(User $user, Claim $claim)

{

// Authorization logic...

}

}

Business Logic

Delegate business logic to Service classes. This keeps your controllers clean and ensures that business logic is reusable and testable.

Example Service:

<?php

namespace App\Services;

use App\Models\Claim;

class ClaimService

{

public function getAllClaims($request)

{

// Business logic to get all claims

}

public function createClaim(array $data)

{

// Business logic to create a claim

}

public function updateClaim(Claim $claim, array $data)

{

// Business logic to update a claim

}

public function deleteClaim(Claim $claim)

{

// Business logic to delete a claim

}

}

Data Transformation

Use Resource classes to transform the data before sending it to the client. This ensures a consistent API response format and makes it easy to modify the response format if needed.

Example Resource:

<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

class ClaimResource extends JsonResource

{

/\*\*

\* Transform the resource into an array.

\*

\* @param \Illuminate\Http\Request $request

\* @return array

\*/

public function toArray($request)

{

return [

'id' => $this->id,

'policy\_id' => $this->policy\_id,

'claim\_amount' => $this->claim\_amount,

'status' => $this->status,

// other attributes...

];

}

}

What to Avoid

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

Avoid Business Logic in Controllers: Controllers should primarily handle routing and delegation. Use Service classes for business logic.

Incorrect:

public function store(Request $request)

{

$validated = $request->validate([

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

'claim\_amount' => 'required|numeric|min:0',

'status' => 'required|in:pending,approved,rejected',

]);

$claim = Claim::create($validated);

return new ClaimResource($claim);

}

Correct:

public function store(ClaimRequest $request)

{

$claim = $this->claimService->createClaim($request->validated());

return new ClaimResource($claim);

}

Avoid Direct Access to Request Data: Use Form Request classes to handle validation and authorization.

Incorrect:

public function update(Request $request, Claim $claim)

{

$claim->update($request->all());

return new ClaimResource($claim);

}

Correct:

public function update(ClaimRequest $request, Claim $claim)

{

$claim = $this->claimService->updateClaim($claim, $request->validated());

return new ClaimResource($claim);

}

Avoid Complex Queries in Controllers: Use Service classes or repositories to encapsulate query logic.

Incorrect:

public function index(Request $request)

{

$claims = Claim::where('status', 'active')->get();

return ClaimResource::collection($claims);

}

Correct:

public function index(Request $request)

{

$claims = $this->claimService->getAllClaims($request);

return ClaimResource::collection($claims);

}