Introduction

Service classes in Laravel are designed to handle the business logic of your application, promoting a clean and maintainable codebase. This document outlines how to architect effective service classes, showcasing their utility and providing guidelines for creating services with low coupling that can be easily used in controllers, jobs, commands, and more.

Benefits of Service Classes

Separation of Concerns: Keeps controllers, jobs, and commands lean by offloading business logic.

Reusability: Business logic can be reused across different parts of the application.

Testability: Isolated business logic makes unit testing simpler and more effective.

Maintainability: Changes in business logic are confined to service classes, reducing the impact on other components.

Naming Conventions

A clear and consistent naming convention helps maintain a clean codebase and makes it easier to understand the purpose of each service class. Follow these guidelines for naming your service classes:

Use a Descriptive Name: The class name should clearly describe the functionality it provides. This helps in understanding the purpose of the service at a glance.

Use PascalCase: Capitalize the first letter of each word in the class name. This standardization improves readability and consistency.

Suffix with "Service": Include the suffix "Service" to indicate that the class is a service class, distinguishing it from other types of classes in your application.

Examples:

PolicyService: For handling policy-related business logic.

ClaimService: For handling claim-related business logic.

UserService: For handling user-related business logic.

Edge Cases:

Compound Actions: When a service involves compound actions, combine the actions in the name.

Example: PolicyRenewalService for handling policy renewal processes.

Specific Contexts: When the service is for a specific context or module, include the context in the name.

Example: AdminPolicyService for handling policy-related logic specific to the admin panel.

Complex Services: For services handling multiple but related responsibilities, clearly outline the scope in the name.

Example: ClaimAdjustmentService for handling claim adjustments.

Composite Names: Use composite names for services that interact with multiple entities.

Example: PolicyClaimService for handling interactions between policies and claims.

Creating a Service Class

1. Basic Structure

Manually create a service class in the App\Services directory:

<?php

namespace App\Services;

class PolicyService

{

public function createPolicy(array $data)

{

// Business logic for creating a policy

}

}

Using Service Classes in Controllers

Service classes can be injected into your controller's constructor or directly into a controller method. Here are examples of both approaches:

Injecting Service Class into Controller Constructor

<?php

namespace App\Http\Controllers;

use App\Services\PolicyService;

use Illuminate\Http\Request;

use App\Http\Resources\PolicyResource;

class PolicyController extends Controller

{

protected $policyService;

public function \_\_construct(PolicyService $policyService)

{

$this->policyService = $policyService;

}

public function store(Request $request)

{

$policy = $this->policyService->createPolicy($request->all());

return new PolicyResource($policy);

}

}

Injecting Service Class into Controller Method

<?php

namespace App\Http\Controllers;

use App\Services\PolicyService;

use Illuminate\Http\Request;

use App\Http\Resources\PolicyResource;

class PolicyController extends Controller

{

public function handleRequest(Request $request, PolicyService $policyService)

{

$policy = $policyService->createPolicy($request->all());

return new PolicyResource($policy);

}

}

Both approaches allow you to utilize service classes to handle business logic while keeping your controllers clean and focused. The constructor injection is useful when the service is needed in multiple methods, while method injection is beneficial for single-use scenarios.

Using Service Classes in Jobs

Inject the service class into your job and call the appropriate method:

<?php

namespace App\Jobs;

use App\Services\PolicyService;

class ProcessPolicyJob extends Job

{

protected $data;

public function \_\_construct(array $data)

{

$this->data = $data;

}

public function handle(PolicyService $policyService)

{

$policyService->createPolicy($this->data);

}

}

Building Good Service Classes

1. Single Responsibility Principle

Ensure each service class has a single responsibility. This makes the code easier to understand, test, and maintain.

Best Practice Example:

Separate the creation of a policy and the processing of a claim into different services:

Policy Creation Service:

<?php

namespace App\Services;

use App\Models\Policy;

class PolicyService

{

public function createPolicy(array $data): Policy

{

// Logic for creating a policy

$policy = new Policy();

$policy->fill($data);

$policy->save();

return $policy;

}

}

Claim Processing Service:

<?php

namespace App\Services;

use App\Models\Claim;

class ClaimService

{

public function processClaim(array $data): Claim

{

// Logic for processing a claim

$claim = new Claim();

$claim->fill($data);

$claim->save();

return $claim;

}

}

Bad Practice Example:

Combining unrelated responsibilities into a single service class, making it harder to maintain and test.

<?php

namespace App\Services;

use App\Models\Policy;

use App\Models\Claim;

class PolicyAndClaimService

{

public function createPolicy(array $data): Policy

{

$policy = new Policy();

$policy->fill($data);

$policy->save();

return $policy;

}

public function processClaim(array $data): Claim

{

$claim = new Claim();

$claim->fill($data);

$claim->save();

return $claim;

}

}

2. Low Coupling

Minimize dependencies within service classes. Use dependency injection to pass only the necessary dependencies.

Best Practice Example:

Inject dependencies through the constructor.

<?php

namespace App\Services;

use App\Repositories\PolicyRepository;

class PolicyService

{

protected $policyRepository;

public function \_\_construct(PolicyRepository $policyRepository)

{

$this->policyRepository = $policyRepository;

}

public function createPolicy(array $data): Policy

{

// Business logic for creating a policy

$policy = new Policy();

$policy->fill($data);

$policy->save();

return $policy;

}

}

Bad Practice Example:

Tightly coupling services by creating dependencies directly within the service class.

<?php

namespace App\Services;

use App\Repositories\PolicyRepository;

class PolicyService

{

protected $policyRepository;

public function \_\_construct()

{

$this->policyRepository = new PolicyRepository();

}

public function createPolicy(array $data): Policy

{

// Business logic for creating a policy

$policy = new Policy();

$policy->fill($data);

$policy->save();

return $policy;

}

}

3. Reusability

Design service methods to be reusable across different parts of the application. Avoid using specific input sources like Auth::user() or Request directly inside service methods. Instead, pass the required data as parameters.

Best Practice Example:

Pass all required data as parameters.

<?php

namespace App\Services;

use App\Models\Policy;

class PolicyService

{

public function createPolicy(array $data): Policy

{

// Logic to create a policy using provided data

$policy = new Policy();

$policy->fill($data);

$policy->save();

return $policy;

}

}

Bad Practice Example:

Using specific input sources inside the service method, making it less reusable.

<?php

namespace App\Services;

use App\Models\Policy;

use Illuminate\Support\Facades\Auth;

class PolicyService

{

public function createPolicy(): Policy

{

// Logic to create a policy using the authenticated user

$policy = new Policy();

$policy->user\_id = Auth::id();

$policy->save();

return $policy;

}

}

4. Simplicity

Keep service methods simple and focused. If a method becomes too complex, consider breaking it into smaller methods or creating a new service class.

Best Practice Example:

Simple and focused methods.

<?php

namespace App\Services;

use App\Models\Policy;

class PolicyService

{

public function createPolicy(array $data): Policy

{

$policy = new Policy();

$policy->fill($data);

$policy->save();

return $policy;

}

public function updatePolicy(Policy $policy, array $data): Policy

{

$policy->update($data);

return $policy;

}

}

Bad Practice Example:

Complex methods that do too much.

<?php

namespace App\Services;

use App\Models\Policy;

use Illuminate\Support\Facades\Log;

class PolicyService

{

public function createOrUpdatePolicy(array $data): Policy

{

if (isset($data['id'])) {

$policy = Policy::find($data['id']);

if

($policy) {

$policy->update($data);

} else {

$policy = new Policy();

$policy->fill($data);

$policy->save();

}

} else {

$policy = new Policy();

$policy->fill($data);

$policy->save();

}

Log::info('Policy created or updated', ['policy' => $policy]);

return $policy;

}

}

By following these best practices and avoiding the common pitfalls shown in the bad practices, you can ensure that your service classes are well-structured, maintainable, and effective.

Service Interaction

To achieve complex business logic, you can compose services within other services:

<?php

namespace App\Services;

use App\Models\Policy;

use App\Models\Claim;

class InsuranceService

{

protected $policyService;

protected $claimService;

public function \_\_construct(PolicyService $policyService, ClaimService $claimService)

{

$this->policyService = $policyService;

$this->claimService = $claimService;

}

public function createPolicyWithClaim(array $policyData, array $claimData): array

{

// Create the policy

$policy = $this->policyService->createPolicy($policyData);

// Create the claim

$claim = $this->claimService->createClaim($claimData);

return ['policy' => $policy, 'claim' => $claim];

}

}

Avoid Returning Response Classes

Service classes should not return Response instances directly. Instead, they should return data, which can then be transformed in controllers, for example, by using resource classes.

Service Class:

<?php

namespace App\Services;

use App\Models\Policy;

class PolicyService

{

public function createPolicy(array $data): Policy

{

$policy = new Policy();

$policy->fill($data);

$policy->save();

return $policy;

}

public function updatePolicy(Policy $policy, array $data): Policy

{

$policy->update($data);

return $policy;

}

}

Controller:

<?php

namespace App\Http\Controllers;

use App\Services\PolicyService;

use Illuminate\Http\Request;

use App\Http\Resources\PolicyResource;

class PolicyController extends Controller

{

protected $policyService;

public function \_\_construct(PolicyService $policyService)

{

$this->policyService = $policyService;

}

public function store(Request $request)

{

$policy = $this->policyService->createPolicy($request->all());

return new PolicyResource($policy);

}

public function update(Request $request, $id)

{

$policy = Policy::findOrFail($id);

$updatedPolicy = $this->policyService->updatePolicy($policy, $request->all());

return new PolicyResource($updatedPolicy);

}

}