Introduction

This document outlines the structure and best practices for defining models in the Laravel backend code. The goal is to ensure that models are well-structured, maintainable, and adhere to the SOLID principles, particularly the Single Responsibility Principle and the Open/Closed Principle.

General Guidelines

Single Responsibility Principle: Each model should only be responsible for interacting with its corresponding database table. Avoid adding business logic directly into the model.

Open/Closed Principle: Models should be open for extension but closed for modification. Use inheritance and traits to add functionality without modifying existing code.

Keep Models Simple: Limit the content of your models to the following:

Fillable attributes

Attribute accessors and mutators

Casts

Appends

Relationships

Common behavior through Traits

Enum classes for enum fields

Simple Query scopes

Naming Conventions

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

Use Singular Form: Model names should be in singular form, as each instance represents a single entity.

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

Reflect the Table Name: The model name should closely reflect the corresponding table name.

Examples:

Policy: For the policies table.

Claim: For the claims table.

UserPolicy: For a table representing the relationship between users and policies.

PolicyClaimDetail: For a table storing detailed information about policy claims.

Edge Cases:

Compound Words: Use PascalCase for models representing tables with compound words.

Example: InsurancePolicy for the insurance\_policies table.

Abbreviations: Maintain readability and consistency, using full words where possible.

Example: UserNotification instead of UsrNotif for the user\_notifications table.

Irregular Plurals: Ensure the model name is in the singular form even if the table name uses an irregular plural.

Example: Person for the people table.

Structure of a Model

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

<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;

use Illuminate\Database\Eloquent\Model;

use App\Enums\ClaimStatusEnum;

use App\Traits\CommonBehaviorTrait;

class Claim extends Model

{

use HasFactory, CommonBehaviorTrait;

/\*\*

\* The attributes that are mass assignable.

\*

\* @var array

\*/

protected $fillable = ['policy\_id', 'claim\_amount', 'status', 'description'];

/\*\*

\* The attributes that should be cast to native types.

\*

\* @var array

\*/

protected $casts = [

'claim\_amount' => 'decimal:2',

'status' => ClaimStatusEnum::class,

];

/\*\*

\* The accessors to append to the model's array form.

\*

\* @var array

\*/

protected $appends = ['formatted\_claim\_amount'];

/\*\*

\* Get the formatted claim amount.

\*

\* @return string

\*/

public function getFormattedClaimAmountAttribute()

{

return '$' . number\_format($this->claim\_amount, 2);

}

/\*\*

\* Define a relationship to the Policy model.

\*

\* @return \Illuminate\Database\Eloquent\Relations\BelongsTo

\*/

public function policy()

{

return $this->belongsTo(Policy::class);

}

// Other relationships...

}

Detailed Explanation

Fillable Attributes

Use the $fillable property to define which attributes are mass assignable. This protects against mass assignment vulnerabilities.

protected $fillable = ['policy\_id', 'claim\_amount', 'status', 'description'];

Attribute Casting

Use the $casts property to cast attributes to common data types. This ensures data consistency and simplifies data manipulation. For enum fields, use enum classes as a cast.

protected $casts = [

'claim\_amount' => 'decimal:2',

'status' => ClaimStatusEnum::class,

];

Appended Attributes

Use the $appends property to add custom attributes to the model's array form. These attributes should be computed properties that do not exist in the database.

protected $appends = ['formatted\_claim\_amount'];

public function getFormattedClaimAmountAttribute()

{

return '$' . number\_format($this->claim\_amount, 2);

}

Relationships

Define relationships using Eloquent's relationship methods. Each relationship should be clearly named and follow Laravel's conventions.

public function policy()

{

return $this->belongsTo(Policy::class);

}

Common Behavior Using Traits

For shared behavior or relationships between models, use Traits. This promotes code reuse and keeps your models clean.

Example Trait:

<?php

namespace App\Traits;

trait CommonBehaviorTrait

{

public function scopeActive($query)

{

return $query->where('status', 'active');

}

public function commonRelationship()

{

return $this->belongsTo(CommonModel::class);

}

}

Usage in a model:

use App\Traits\CommonBehaviorTrait;

class Claim extends Model

{

use CommonBehaviorTrait;

// ...

}

Enum Classes for Enum Fields

Define enum classes for enum fields to enforce strict typing and improve code readability. This approach is useful because it allows you to extend the column behavior without updating the model itself, which achieves the Open/Closed Principle.

Example Enum:

<?php

namespace App\Enums;

enum ClaimStatusEnum: string

{

case PENDING = 'pending';

case APPROVED = 'approved';

case REJECTED = 'rejected';

}

Usage in a model:

protected $casts = [

'status' => ClaimStatusEnum::class,

];

What to Avoid

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

Avoid Business Logic: Do not place business logic in the model. Models should only handle data interactions with the database. Use service classes or other appropriate layers for business logic.

Example: Instead of placing logic in the model, create a service class.

// In a service class

class ClaimService

{

public function processClaim(Claim $claim)

{

// Business logic here

}

}

Avoid Complex Queries in Models: Avoid placing complex queries directly in the model. Use Laravel scopes, query scopes, or repositories to encapsulate query logic.

Example:

public function scopeActive($query)

{

return $query->where('status', 'active');

}

Avoid Direct Access to Request Data: Do not directly access request data in the model. This can lead to tightly coupled code and violates the single responsibility principle. Handle request data in controllers or service classes.

Incorrect:

// Directly accessing request data in a model

public static function createFromRequest($request)

{

return self::create($request->all());

}

Correct:

// Handling request data in a controller or service class

$claim = Claim::create($request->only(['policy\_id', 'claim\_amount', 'status']));

Avoid Using Authentication (Auth) and Request in Models: Do not use the Auth facade or access request data directly in models. This practice tightly couples the model to the framework and specific request/response handling, which violates the single responsibility principle and makes testing difficult.

Incorrect:

// Using Auth in a model

public function scopeOwnedByAuthenticatedUser($query)

{

return $query->where('user\_id', Auth::id());

}

Correct:

// Handling authentication in a controller or service class

$ownedClaims = Claim::where('user\_id', Auth::id())->get();

Avoid Adding Non-Data Related Methods: Models should only contain methods related to data manipulation, such as accessors, mutators, and relationships. Avoid adding methods that handle non-data related logic, such as notifications, translations, and logging.

Incorrect:

// Non-data related method in a model

public function sendNotification()

{

Notification::send($this->user, new ClaimNotification($this));

}

// Translation logic in a model

public function translateTitle($locale)

{

return $this->translations[$locale]['title'] ?? $this->title;

}

// Logging in a model

public function logCreation()

{

Log::info('Claim created', ['id' => $this->id]);

}

Correct:

// Non-data related logic handled in a service class

class ClaimService

{

public function sendNotification(Claim $claim)

{

Notification::send($claim->user, new ClaimNotification($claim));

}

public function translateTitle(Claim $claim, $locale)

{

return $claim->translations[$locale]['title'] ?? $claim->title;

}

public function logCreation(Claim $claim)

{

Log::info('Claim created', ['id' => $claim->id]);

}

}