Introduction

This document outlines best practices, naming conventions, and advanced topics for creating and managing database migrations in a Laravel application. Following these guidelines will help maintain a consistent and manageable codebase.

Naming Conventions

Migration Files

Prefix: Use a timestamp prefix generated by Laravel to ensure migrations are run in the correct order.

Descriptive Names: Use a descriptive name that explains the purpose of the migration.

Example:

2024\_05\_21\_123456\_create\_policies\_table.php

Tables and Columns

Tables: Use plural snake\_case for table names (e.g., policies, claims).

Columns: Use snake\_case for column names (e.g., policy\_number, claim\_date).

Primary Key: The primary key of each table should be named id.

Example:

Schema::create('policies', function (Blueprint $table) {

$table->id();

$table->string('policy\_number');

$table->string('holder\_name');

$table->timestamps();

});

Foreign Keys

Naming: Follow the convention table\_singular\_id for foreign key columns.

Example:

$table->foreignId('policy\_id')->constrained();

Indexes

Naming: Use the convention table\_column\_index for index names.

Example:

$table->index('created\_at', 'claims\_created\_at\_index');

Pivot Tables

Naming: Use singular names of the related models in alphabetical order, separated by an underscore.

Example:

Schema::create('claim\_policy', function (Blueprint $table) {

$table->id();

$table->foreignId('claim\_id')->constrained();

$table->foreignId('policy\_id')->constrained();

$table->timestamps();

});

Best Practices

1. Use Descriptive Names for Migrations

Purpose: Make it clear what the migration does.

Format: Use the format create\_table\_name\_table, add\_column\_to\_table\_name\_table, update\_table\_name\_structure, etc.

Examples:

php artisan make:migration create\_policies\_table

php artisan make:migration add\_status\_to\_claims\_table

php artisan make:migration update\_claims\_structure

2. Atomic Migrations

Ensure each migration is atomic, meaning it performs a single clear task.

Avoid combining multiple unrelated schema changes in a single migration.

3. Use Schema Builder Methods

Use Laravel's schema builder methods to define your tables and columns.

This ensures compatibility and leverages Laravel's abstraction for various database systems.

4. Foreign Keys and Indexes

Always define foreign keys to enforce data integrity.

Add indexes to columns that are frequently queried or used in joins.

Example:

Schema::table('claims', function (Blueprint $table) {

$table->foreignId('policy\_id')->constrained()->nullOnDelete();

$table->index('created\_at');

});

5. Default Values and Nullable Columns

Define default values where applicable to avoid NULL values unless they are necessary.

Use nullable columns thoughtfully, ensuring only columns that truly need to accept null values are nullable.

6. Use Descriptive Enums

Purpose: Make use of database enums for columns that have a fixed set of possible values.

Format: Use descriptive names for enum values and document them within your code for clarity.

Example: Define an enum column in a migration and use it in the model for type safety.

Migration:

Schema::table('claims', function (Blueprint $table) {

$table->enum('status', ['pending', 'approved', 'denied'])->default('pending');

});

Model:

use BenSampo\Enum\Traits\CastsEnums;

class Claim extends Model

{

use CastsEnums;

protected $enumCasts = [

'status' => ClaimStatus::class,

];

}

// Enum definition

namespace App\Enums;

use BenSampo\Enum\Enum;

final class ClaimStatus extends Enum

{

const Pending = 'pending';

const Approved = 'approved';

const Denied = 'denied';

}

7. Avoid Direct Schema Modifications

Avoid making schema changes directly in production.

Always use migrations to ensure changes are tracked and can be rolled back if necessary.

8. Handle Column Deletions with Care

If you need to remove a column, ensure that it doesn't break any application logic.

Consider the impact on existing data and any dependencies.

9. Testing Migrations

Test migrations locally before deploying them to other environments.

Use Laravel's migrate:fresh --seed command to ensure your migrations work from scratch.

10. Managing Large Migrations

Break down large migrations into smaller, more manageable pieces.

Consider using batches to handle large data transformations.

11. Handling Data Migrations

Use Laravel's DB facade to perform data transformations within migrations.

Ensure data integrity by validating data before performing transformations.

Example:

Schema::table('policies', function (Blueprint $table) {

$table->string('full\_name')->nullable();

});

DB::table('policies')->get()->each(function ($policy) {

DB::table('policies')

->where('id', $policy->id)

->update(['full\_name' => "{$policy->first\_name} {$policy->last\_name}"]);

});

Schema::table('policies', function (Blueprint $table) {

$table->dropColumn(['first\_name', 'last\_name']);

});

12. Using Migration Events

Utilize Laravel's migration events (migrating, migrated, rollingBack, rolledBack) to hook into the migration process.

This can be useful for logging, monitoring, or triggering additional processes.

Example:

Event::listen('Illuminate\Database\Events\MigrationsStarted', function ($event) {

// Handle the event

});

13. Handling Production Migrations

Backup your database before running migrations in production.

Run migrations in a maintenance window to minimize impact on users.

Monitor the migration process and be prepared to roll back if necessary.

14. Using UUIDs

Use UUIDs for non-primary key columns where unique identification is necessary or where you want to hide the ID from users.

Note that integers are faster for indexing and queries, so reserve UUIDs for where uniqueness across distributed systems or user privacy is crucial.

Example:

use Illuminate\Support\Str;

Schema::table('claims', function (Blueprint $table) {

$table->uuid('claim\_uuid')->unique();

});

// In your model

public static function boot()

{

parent::boot();

static::creating(function ($model) {

$model->claim\_uuid = (string) Str::uuid();

});

}