How to Use Laravel Query Scopes for Cleaner Code

As your application grows, your database queries can quickly become repetitive and messy. You may find yourself writing the same conditions across multiple controllers, such as filtering active users, published posts, or verified accounts. This is where **Laravel 12 Query Scopes** come in. They allow you to encapsulate common query logic directly inside your Eloquent models, keeping your code clean, reusable, and easier to maintain.

In this tutorial, we'll walk through **creating and using query scopes in Laravel 12**. We'll cover what they are, how to build local and global scopes, and how to integrate them into your models, controllers, and even your UI for filtering data. By the end, you'll have a clear understanding of how to simplify your queries and keep your codebase organized.

1 – What Are Query Scopes in Laravel?

A **query scope** in Laravel is a way to define reusable query constraints inside your Eloquent models. Instead of repeating the same `where` conditions across your controllers and repositories, you can wrap them into a scope method and call it like a regular query method. This keeps your code cleaner and more maintainable.

Laravel offers two types of scopes:

- **Local scopes:** Defined inside models, reusable across queries with a short method call.
- **Global scopes:** Applied automatically to all queries for a model (e.g., only load active users).

2 – Creating a Local Scope

Local scopes are useful when you need to apply a common filter repeatedly. For example, if you want to fetch only `active` users from your `users` table, you can define a scope directly in the model:

```
// app/Models/User.php

namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    // Local scope method
    public function scopeActive($query)
    {
        return $query->where('is_active', true);
    }
}Code language: PHP (php)
```

The `scopeActive` method defines a filter where `is_active` is true. By convention, Laravel removes the “scope” prefix, so you can call it simply as `User::active()`.

```
// Using the scope in a controller
$activeUsers = User::active()->get();Code language: PHP (php)
```

Instead of writing `User::where('is_active', true)->get()` everywhere, you now have a clean reusable method.

3 – Parameterized Scopes

Scopes can also accept parameters, making them flexible for filtering dynamic data. For example, filtering posts by status:

```
// app/Models/Post.php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    public function scopeStatus($query, $status)
    {
        return $query->where('status', $status);
    }
}Code language: PHP (php)
```

Now you can query posts by any status in a clean way:

```
// Fetch only published posts
$published = Post::status('published')->get();

// Fetch only drafts
$drafts = Post::status('draft')->get();Code language: PHP (php)
```

By using parameterized scopes, you keep your queries consistent and avoid typos in column names across controllers.

4 – Global Scopes

Unlike local scopes, **global scopes** are automatically applied to all queries on a model. They're perfect for multi-tenant apps or cases where you always want to hide certain records.

```
// app/Models/Scopes/ActiveScope.php
namespace App\Models\Scopes;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Scope;

class ActiveScope implements Scope
{
    public function apply(Builder $builder, Model $model)
    {
        $builder->where('is_active', true);
    }
}Code language: PHP (php)
```

This scope forces all queries to include `where is_active = true` automatically.

```
// app/Models/User.php
namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;
use App\Models\Scopes\ActiveScope;

class User extends Authenticatable
{
    protected static function booted()
    {
        static::addGlobalScope(new ActiveScope);
    }
}Code language: PHP (php)
```

Now whenever you query `User::all()`, only active users are returned. If you need to

ignore the global scope, use `User::withoutGlobalScope(ActiveScope::class)->get()`.

5 – Using Scopes in the UI

Suppose you want to show users a filtered list of published posts on the dashboard. You can use your scope inside a controller and then display the results in a Blade view.

```
// app/Http/Controllers/PostController.php
namespace App\Http\Controllers;

use App\Models\Post;

class PostController extends Controller
{
    public function index()
    {
        $posts = Post::status('published')->latest()->paginate(10);
        return view('posts.index', compact('posts'));
    }
}Code language: PHP (php)
```

This controller uses the `status` scope we defined to only fetch published posts. It also uses `paginate(10)` so we can show 10 posts per page.

```
<!-- resources/views/posts/index.blade.php -->
@extends('layouts.app')

@section('content')
<div class="container">
  <h1 class="mb-4">Published Posts</h1>

  @foreach($posts as $post)
```

```
    <div class="card mb-3">
      <div class="card-body">
        <h5 class="card-title">{{ $post->title }}</h5>
        <p class="card-text">{{ Str::limit($post->content, 120) }}</p>
        <a href="{{ route('posts.show', $post) }}" class="btn btn-
theme r-04">
          <i class="bi bi-eye"></i> View Post
        </a>
      </div>
    </div>
  @endforeach

  {{ $posts->links() }}
</div>
@endsectionCode language: PHP (php)
```

This Blade file lists published posts using the `@foreach` loop. The scope keeps the controller clean, while the view focuses only on presentation.

Wrapping Up

Laravel Eloquent Query Scopes are a powerful way to organize and reuse your query logic. Instead of repeating conditions in multiple controllers, you can move them into your models as local or global scopes. Local scopes keep queries cleaner and easier to read, while global scopes automatically apply rules like filtering active users or excluding soft-deleted records. Adding scopes to your projects not only improves performance but also makes your codebase more maintainable and easier to extend.

What’s Next

- Filtering and Searching with Laravel Eloquent Query Builder
- Soft Deletes in Laravel: Restore, Force Delete, and Prune Data
- How to Use Eloquent API Resources for Clean APIs