How to Use Eloquent API Resources for Clean APIs

How to Use Eloquent API Resources for Clean APIs

Eloquent API Resources give you a clean, explicit layer to shape JSON responses. Instead of returning raw models (which may leak internal fields), resources let you control fields, nest relations, add meta and links, and keep your API consistent. In this guide, you'll build resources for posts and users, paginate collections, conditionally include attributes and relations, and add meta information—then consume the API from a simple UI.

1 – Create Models (Sample Domain)

We'll use a simple domain: `Post` belongs to `User`. If you already have these, skip ahead.

```
php artisan make:model Post -mfCode language: Bash (bash)
```

This creates the model, a migration, and a factory. You'll use them to scaffold data for testing the API responses.

```
// database/migrations/xxxx_xx_xx_create_posts_table.php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration {
    public function up(): void {
        Schema::create('posts', function (Blueprint $table) {
            $table->id();
$table->foreignId('user_id')->constrained()->cascadeOnDelete();
            $table->string('title');
            $table->text('body');
```

```
            $table->json('meta')->nullable(); // tags, reading_time,
etc.
            $table->timestamps();
        });
    }
    public function down(): void {
        Schema::dropIfExists('posts');
    }
};Code language: PHP (php)
```

The migration includes a nullable JSON `meta` field to demonstrate how resources can shape nested JSON cleanly.

2 – Generate API Resources

Create resources for `Post` and `User`. These classes transform models to JSON.

```
php artisan make:resource PostResource
php artisan make:resource UserResourceCode language: Bash (bash)
```

Resources live in `app/Http/Resources`. Each implements a `toArray($request)` method that returns the exact JSON structure you want to expose.

```
// app/Http/Resources/UserResource.php
namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

class UserResource extends JsonResource
{
    public function toArray($request): array
    {
        return [
```

```
            'id'    => $this->id,
            'name'  => $this->name,
            'email' => $this->email,
        ];
    }
}Code language: PHP (php)
```

This keeps the user payload minimal and explicit. If you don't want to expose emails publicly, omit or conditionally include it later.

```
// app/Http/Resources/PostResource.php
namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

class PostResource extends JsonResource
{
    public function toArray($request): array
    {
        return [
            'id'      => $this->id,
            'title'   => $this->title,
            'excerpt' => str($this->body)->limit(140),
            'body'    => $this->when($request->routeIs('posts.show'),
$this->body),
            'meta'    => [
                'tags'         => data_get($this->meta, 'tags', []),
                'reading_time' => data_get($this->meta,
'reading_time'),
            ],
            'author'  => new UserResource($this->whenLoaded('user')),
            'created_at' => $this->created_at?->toIso8601String(),
            'updated_at' => $this->updated_at?->toIso8601String(),
            // Example link
            'links' => [
                'self' => route('posts.show', $this->id),
            ],
        ];
```

```
    }
}Code language: PHP (php)
```

This resource exposes a short `excerpt` for lists and includes the full `body` only on the show route using `when()`. The `author` is wrapped in `UserResource` and included only if the relation is eager-loaded (`whenLoaded('user')`).

3 – Controller: Return Resources and Collections

Use resources in controller methods. For collections, either use `PostResource::collection($posts)` or make a separate `PostCollection`. Here we'll keep it simple.

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

use App\Http\Resources\PostResource;
use App\Models\Post;
use Illuminate\Http\Request;

class PostController extends Controller
{
    public function index(Request $request)
    {
        $posts = Post::with('user')
            ->latest()
            ->paginate(10);

        return PostResource::collection($posts)
            ->additional([
                'meta' => [
                    'copyright' => '© Your Company',
                ],
```

```
            ]);
    }

    public function show(Post $post)
    {
        $post->load('user');

        return new PostResource($post);
    }

    public function store(Request $request)
    {
        $data = $request->validate([
            'title' => ['required','string','max:150'],
            'body'  => ['required','string'],
            'meta'  => ['nullable','array'],
        ]);

        $post = Post::create($data + ['user_id' =>
$request->user()->id]);

        return (new PostResource($post->load('user')))
            ->response()
            ->setStatusCode(201);
    }
}Code language: PHP (php)
```

`index()` returns a paginated collection wrapped by `PostResource`, and `additional()` appends custom top-level `meta`. `show()` returns a single resource. `store()` validates input, creates the post, and returns the serialized resource with status 201.

4 – API Routes and Pagination Shape

Define API routes and see how pagination metadata/links are included automatically when you pass a LengthAwarePaginator into `::collection()`.

```
// routes/api.php
use App\Http\Controllers\PostController;

Route::get('/posts', [PostController::class,
'index'])->name('posts.index');
Route::get('/posts/{post}', [PostController::class,
'show'])->name('posts.show');
Route::middleware('auth:sanctum')->post('/posts',
[PostController::class, 'store'])->name('posts.store');Code language: PHP
(php)
```

Unprotected `GET` routes serve public data. The `POST` route requires authentication (e.g., Sanctum), keeping write operations secured.

If you don't want the default wrapping key (`data`) around your resource payloads, you can disable it globally:

```
// app/Providers/AppServiceProvider.php (boot)
use Illuminate\Http\Resources\Json\JsonResource;

public function boot(): void
{
    JsonResource::withoutWrapping();
}Code language: PHP (php)
```

With wrapping disabled, the collection returns an array at the top level alongside `links` and `meta` for pagination. Choose the style your clients expect.

5 – Conditional Attributes and Merged Blocks

Use helpers like `when()`, `mergeWhen()`, and `whenLoaded()` to keep responses compact and context-aware.

```
// app/Http/Resources/PostResource.php (snippets)
return [
    // ...
    'is_owner' => $this->when(
        optional($request->user())?->id === $this->user_id,
        true
    ),

    $this->mergeWhen($request->routeIs('posts.show'), [
        'full_body' => $this->body,
        'meta'      => $this->meta,
    ]),

    'author' => new UserResource($this->whenLoaded('user')),
];Code language: PHP (php)
```

`when()` includes fields only under certain conditions (e.g., current user is the owner). `mergeWhen()` adds a block of fields for detail views without bloating list responses. `whenLoaded()` safely includes relations if they were eager-loaded.

6 – Eager Loading and N+1 Safety

Resources don't fetch relations; they only serialize what you loaded. Always eager-load relations in controllers to avoid N+1 queries.

```
// In PostController@index
$posts = Post::with('user')->latest()->paginate(10);
```

```
return PostResource::collection($posts);Code language: PHP (php)
```

Because `user` is loaded on the query, `UserResource` in `PostResource` can serialize the `author` efficiently.

7 – Adding Top-Level Meta and Custom Status Codes

Return HTTP status codes using the `response()` helper, and add top-level `meta` or `links` when needed.

```
// Example in store()
return (new PostResource($post->load('user')))
    ->additional(['meta' => ['created' => true]])
    ->response()
    ->setStatusCode(201);Code language: PHP (php)
```

This returns the serialized post with a `201 Created` status and a `meta.created` flag, which is handy for clients.

8 – UI: Consuming the API from a Blade Page

Here's a minimal UI that fetches `/api/posts` and renders a list. You can adapt this to your marketing site or dashboard.

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

```
@section('content')
<div class="container">
  <h1 class="mb-4">Latest Posts</h1>
  <div id="post-list">Loading...</div>
</div>

@push('scripts')
<script>
  fetch('/api/posts')
    .then(r => r.json())
    .then(json => {
      const data = json.data ?? json; // supports wrapped or unwrapped
      const list = document.getElementById('post-list');
      list.innerHTML = '';
      data.forEach(p => {
        const card = document.createElement('div');
        card.className = 'card mb-3';
        card.innerHTML = `
          <div class="card-body">
            <h5 class="card-title">${p.title}</h5>
            <p class="card-text">${p.excerpt}</p>
            <a href="/posts/${p.id}" class="btn btn-theme
r-04">Read</a>
          </div>`;
        list.appendChild(card);
      });
    });
</script>
@endpush
@endsectionCode language: PHP (php)
```

The UI calls your API, supports either wrapped (`{ data: [...] }`) or unwrapped collection payloads, and renders cards with title and excerpt from the resource.

Wrapping Up

Eloquent API Resources give you a declarative way to shape JSON: pick fields, conditionally include details, nest relations, add meta, and keep pagination consistent. Keep controllers focused on queries and authorization, and let resources handle presentation of data for clients. This makes for predictable, maintainable APIs that evolve gracefully.

What's Next

- Soft Deletes: Restore, Force Delete, and Prune Data
- Handling Large Data Sets with Chunking & Cursors
- Filtering and Searching with Eloquent Query Builder