Transforming API Responses with Eloquent API Resources and Pagination

Laravel's Eloquent API Resources provide a powerful way to transform your models and collections into JSON responses with full control over the structure. Here's how to implement them with pagination.

1. Create an API Resource

Generate a resource for your model (e.g., `User`):

```
php artisan make:resource UserResource
```

This creates `app/Http/Resources/UserResource.php`:

```
namespace App\Http\Resources;

use Illuminate\Http\Request;
use Illuminate\Http\Resources\Json\JsonResource;

class UserResource extends JsonResource
{
    public function toArray(Request $request): array
    {
        return [
            'id' => $this->id,
            'name' => $this->name,
            'email' => $this->email,
            'created_at' => $this->created_at->format('Y-m-d H:i:s'),
            'updated_at' => $this->updated_at->format('Y-m-d H:i:s'),

            // Relationships
            'posts' => PostResource::collection($this-
>whenLoaded('posts')),
            'profile' => new ProfileResource($this-
>whenLoaded('profile')),
        ];
    }
}
```

2. Create a Resource Collection (Optional)

For more control over collection responses:

```
php artisan make:resource UserCollection --collection
```

Or use the existing resource with `::collection()` method (shown below).

3. Implement Paginated API Endpoint

In your controller:

```
namespace App\Http\Controllers;

use App\Http\Resources\UserResource;
use App\Models\User;
use Illuminate\Http\Request;

class UserController extends Controller
{
    public function index(Request $request)
    {
        $users = User::with(['posts', 'profile'])
            ->filter($request->all())
            ->paginate($request->per_page ?? 15);

        return UserResource::collection($users);
    }

    public function show(User $user)
    {
        $user->load(['posts', 'profile']);
        return new UserResource($user);
    }
}
```

4. Customizing Pagination Response

PROF

To customize the pagination structure, create a custom resource collection:

```
namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class UserCollection extends ResourceCollection
{
    public function toArray($request)
    {
        return [
            'data' => $this->collection,
            'pagination' => [
                'total' => $this->total(),
                'count' => $this->count(),
```

```
                'per_page' => $this->perPage(),
                'current_page' => $this->currentPage(),
                'total_pages' => $this->lastPage(),
                'links' => [
                    'next' => $this->nextPageUrl(),
                    'previous' => $this->previousPageUrl(),
                ],
            ],
        ];
    }

    public function with($request)
    {
        return [
            'meta' => [
                'version' => '1.0',
                'api_status' => 'stable',
            ],
        ];
    }
}
```

Then update your controller to use the collection:

```
public function index(Request $request)
{
    $users = User::with(['posts', 'profile'])
        ->filter($request->all())
        ->paginate($request->per_page ?? 15);

    return new UserCollection($users);
}
```

PROF

5. Conditional Attributes and Relationships

Add logic to include fields conditionally:

```
// In UserResource
public function toArray(Request $request): array
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->when($request->user()->isAdmin(), $this->email),
        'roles' => RoleResource::collection($this->whenLoaded('roles')),
        'profile' => new ProfileResource($this->whenLoaded('profile')),
        'created_at' => $this->when($request->show_timestamps, $this-
```

```
>created_at),
        ];
    }
```

6. Adding Metadata to Responses

Add metadata at the resource level:

```php
public function with($request)
{
    return [
        'meta' => [
            'version' => '1.0.0',
            'author' => 'Your API Team',
        ],
    ];
}
```

7. Example API Response

With pagination, the response will look like:

```
{
    "data": [
        {
            "id": 1,
            "name": "John Doe",
            "email": "john@example.com",
            "created_at": "2023-05-15 10:00:00",
            "posts": [
                {
                    "id": 1,
                    "title": "First Post"
                }
            ],
            "profile": {
                "bio": "Web developer"
            }
        }
    ],
    "links": {
        "first": "http://example.com/users?page=1",
        "last": "http://example.com/users?page=5",
        "prev": null,
        "next": "http://example.com/users?page=2"
    },
    "meta": {
        "current_page": 1,
```

```
        "from": 1,
        "last_page": 5,
        "path": "http://example.com/users",
        "per_page": 15,
        "to": 15,
        "total": 75,
        "version": "1.0",
        "api_status": "stable"
    }
}
```

8. Sorting and Filtering

Enhance your endpoint with sorting and filtering:

```php
public function index(Request $request)
{
    $query = User::query()->with(['posts', 'profile']);

    // Filtering
    if ($request->has('name')) {
        $query->where('name', 'like', '%' . $request->name . '%');
    }

    // Sorting
    $sortField = $request->sort_field ?? 'created_at';
    $sortOrder = $request->sort_order ?? 'desc';
    $query->orderBy($sortField, $sortOrder);

    // Pagination
    $perPage = $request->per_page ?? 15;
    $users = $query->paginate($perPage);

    return new UserCollection($users);
}
```

PROF

9. API Resource Best Practices

1. **Keep resources lean** - Only include necessary data
2. **Use relationships sparingly** - Load them explicitly with `with()`
3. **Consistent structure** - Maintain the same response format across endpoints
4. **Document your API** - Use tools like Swagger/OpenAPI
5. **Version your API** - Include version in response metadata

10. Testing Your API Resources

Create tests to ensure your resources work as expected:

```php
namespace Tests\Feature;

use App\Http\Resources\UserResource;
use App\Models\User;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Tests\TestCase;

class UserResourceTest extends TestCase
{
    use RefreshDatabase;

    public function test_user_resource_structure()
    {
        $user = User::factory()->create();

        $resource = new UserResource($user);

        $response = $resource->response()->getData(true);

        $this->assertArrayHasKey('data', $response);
        $this->assertEquals([
            'id',
            'name',
            'email',
            'created_at',
            'updated_at',
        ], array_keys($response['data']));
    }

    public function test_user_collection_pagination()
    {
        User::factory()->count(20)->create();

        $response = $this->getJson('/api/users?per_page=5');

        $response->assertJsonStructure([
            'data',
            'links',
            'meta' => [
                'current_page',
                'per_page',
                'total',
            ],
        ]);
    }
}
```

PROF

This implementation gives you full control over your API responses while maintaining clean, maintainable code. The pagination integration ensures your API is scalable and follows RESTful best practices.