Eloquent: API Resources

# Introduction

When building an API, you may need a transformation layer that sits between your Eloquent models and the JSON responses that are actually returned to your application's users. Laravel's resource classes allow you to expressively and easily transform your models and model collections into JSON.

Generating Resources

To generate a resource class, you may use the `make:resource` Artisan command. By default, resources will be placed in the `app/Http/Resources` directory of your application. Resources extend the `Illuminate\Http\Resources\Json\JsonResource` class:

```
php artisan make:resource User
```

Resource Collections

In addition to generating resources that transform individual models, you may generate resources that are responsible for transforming collections of models. This allows your response to include links and other meta information that is relevant to an entire collection of a given resource.

To create a resource collection, you should use the `--collection` flag when creating the resource. Or, including the word `Collection` in the resource name will indicate to Laravel that it should create a collection resource. Collection resources extend the `Illuminate\Http\Resources\Json\ResourceCollection` class:

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

php artisan make:resource UserCollection
```

# Concept Overview

This is a high-level overview of resources and resource collections. You are highly encouraged to read the other sections of this documentation to gain a deeper understanding of the customization and power offered to you by resources.

Before diving into all of the options available to you when writing resources, let's first take a high-level look at how resources are used within Laravel. A resource class represents a single model that needs to be transformed into a JSON structure. For example, here is a simple `User` resource class:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

class User extends JsonResource
{
    /**
     * Transform the resource into an array.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return array
     */
    public function toArray($request)
    {
        return [
            'id' => $this->id,
            'name' => $this->name,
            'email' => $this->email,
            'created_at' => $this->created_at,
            'updated_at' => $this->updated_at,
        ];
    }
}
```

Every resource class defines a `toArray` method which returns the array of attributes that should be converted to JSON when sending the response. Notice that we can access model properties directly from the `$this` variable. This is because a resource class will automatically proxy property and method access down to the underlying model for convenient access. Once the resource is defined, it may be returned from a route or controller:

```
use App\Http\Resources\User as UserResource;
use App\User;

Route::get('/user', function () {
    return new UserResource(User::find(1));
});
```

Resource Collections

If you are returning a collection of resources or a paginated response, you may use the `collection` method when creating the resource instance in your route or controller:

```
use App\Http\Resources\User as UserResource;
use App\User;

Route::get('/user', function () {
    return UserResource::collection(User::all());
});
```

Note that this does not allow any addition of meta data that may need to be returned with the collection. If you would like to customize the resource collection response, you may create a dedicated resource to represent the collection:

```
php artisan make:resource UserCollection
```

Once the resource collection class has been generated, you may easily define any meta data that should be included with the response:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class UserCollection extends ResourceCollection
{
    /**
     * Transform the resource collection into an array.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return array
     */
    public function toArray($request)
    {
        return [
            'data' => $this->collection,
            'links' => [
                'self' => 'link-value',
            ],
        ];
    }
}
```

After defining your resource collection, it may be returned from a route or controller:

```
use App\Http\Resources\UserCollection;
use App\User;

Route::get('/users', function () {
    return new UserCollection(User::all());
});
```

Preserving Collection Keys

When returning a resource collection from a route, Laravel resets the collection's keys so that they are in simple numerical order. However, you may add a `preserveKeys` property to your resource class indicating if collection keys should be preserved:

```php
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

class User extends JsonResource
{
    /**
     * Indicates if the resource's collection keys should be preserved.
     *
     * @var bool
     */
    public $preserveKeys = true;
}
```

When the `preserveKeys` property is set to `true`, collection keys will be preserved:

```php
use App\Http\Resources\User as UserResource;
use App\User;

Route::get('/user', function () {
    return UserResource::collection(User::all()->keyBy->id);
});
```

Customizing The Underlying Resource Class

Typically, the `$this->collection` property of a resource collection is automatically populated with the result of mapping each item of the collection to its singular resource class. The singular resource class is assumed to be the collection's class name without the trailing `Collection` string.

For example, `UserCollection` will attempt to map the given user instances into the `User` resource. To customize this behavior, you may override the `$collects` property of your resource collection:

```php
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class UserCollection extends ResourceCollection
{
    /**
     * The resource that this resource collects.
     *
     * @var string
     */
    public $collects = 'App\Http\Resources\Member';
}
```

Writing Resources

If you have not read the concept overview, you are highly encouraged to do so before proceeding with this documentation.

In essence, resources are simple. They only need to transform a given model into an array. So, each resource contains a `toArray` method which translates your model's attributes into an API friendly array that can be returned to your users:

```php
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

class User extends JsonResource
{
    /**
     * Transform the resource into an array.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return array
     */
    public function toArray($request)
    {
        return [
            'id' => $this->id,
            'name' => $this->name,
            'email' => $this->email,
            'created_at' => $this->created_at,
            'updated_at' => $this->updated_at,
        ];
    }
}
```

Once a resource has been defined, it may be returned directly from a route or controller:

```
use App\Http\Resources\User as UserResource;
use App\User;

Route::get('/user', function () {
    return new UserResource(User::find(1));
});
```

Relationships

If you would like to include related resources in your response, you may add them to the array returned by your `toArray` method. In this example, we will use the `Post` resource's `collection` method to add the user's blog posts to the resource response:

```
/**
 * Transform the resource into an array.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return array
 */
public function toArray($request)
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->email,
        'posts' => PostResource::collection($this->posts),
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ];
}
```

If you would like to include relationships only when they have already been loaded, check out the documentation on conditional relationships.

Resource Collections

While resources translate a single model into an array, resource collections translate a collection of models into an array. It is not absolutely necessary to define a resource collection class for each one of your model types since all resources provide a `collection` method to generate an "ad-hoc" resource collection on the fly:

```
use App\Http\Resources\User as UserResource;
use App\User;

Route::get('/user', function () {
    return UserResource::collection(User::all());
});
```

However, if you need to customize the meta data returned with the collection, it will be necessary to define a resource collection:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class UserCollection extends ResourceCollection
{
    /**
     * Transform the resource collection into an array.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return array
     */
    public function toArray($request)
    {
        return [
            'data' => $this->collection,
            'links' => [
                'self' => 'link-value',
            ],
        ];
    }
}
```

Like singular resources, resource collections may be returned directly from routes or controllers:

```
use App\Http\Resources\UserCollection;
use App\User;

Route::get('/users', function () {
    return new UserCollection(User::all());
});
```

Data Wrapping

By default, your outer-most resource is wrapped in a `data` key when the resource response is converted to JSON. So, for example, a typical resource collection response looks like the following:

```
{
    "data": [
        {
            "id": 1,
            "name": "Eladio Schroeder Sr.",
            "email": "therese28@example.com",
        },
        {
            "id": 2,
            "name": "Liliana Mayert",
            "email": "evandervort@example.com",
        }
    ]
}
```

If you would like to disable the wrapping of the outer-most resource, you may use the `withoutWrapping` method on the base resource class. Typically, you should call this method from your `AppServiceProvider` or another service provider that is loaded on every request to your application:

```
<?php

namespace App\Providers;

use Illuminate\Http\Resources\Json\Resource;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function register()
    {
        //
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Resource::withoutWrapping();
    }
}
```

! The `withoutWrapping` method only affects the outer-most response and will not remove `data` keys that you manually add to your own resource collections.

Wrapping Nested Resources

You have total freedom to determine how your resource's relationships are wrapped. If you would like all resource collections to be wrapped in a `data` key, regardless of their nesting, you should define a resource collection class for each resource and return the collection within a `data` key.

You may be wondering if this will cause your outer-most resource to be wrapped in two `data` keys. Don't worry, Laravel will never let your resources be accidentally double-wrapped, so you don't have to be concerned about the nesting level of the resource collection you are transforming:

```php
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class CommentsCollection extends ResourceCollection
{
    /**
     * Transform the resource collection into an array.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return array
     */
    public function toArray($request)
    {
        return ['data' => $this->collection];
    }
}
```

Data Wrapping And Pagination

When returning paginated collections in a resource response, Laravel will wrap your resource data in a `data` key even if the `withoutWrapping` method has been called. This is because paginated responses always contain `meta` and `links` keys with information about the paginator's state:

```
{
    "data": [
        {
            "id": 1,
            "name": "Eladio Schroeder Sr.",
            "email": "therese28@example.com",
        },
        {
            "id": 2,
            "name": "Liliana Mayert",
            "email": "evandervort@example.com",
        }
    ],
    "links":{
        "first": "http://example.com/pagination?page=1",
        "last": "http://example.com/pagination?page=1",
        "prev": null,
        "next": null
    },
    "meta":{
        "current_page": 1,
        "from": 1,
        "last_page": 1,
        "path": "http://example.com/pagination",
        "per_page": 15,
        "to": 10,
        "total": 10
    }
}
```

Pagination

You may always pass a paginator instance to the `collection` method of a resource or to a custom resource collection:

```php
use App\Http\Resources\UserCollection;
use App\User;

Route::get('/users', function () {
    return new UserCollection(User::paginate());
});
```

Paginated responses always contain `meta` and `links` keys with information about the paginator's state:

```
{
    "data": [
        {
            "id": 1,
            "name": "Eladio Schroeder Sr.",
            "email": "therese28@example.com",
        },
        {
            "id": 2,
            "name": "Liliana Mayert",
            "email": "evandervort@example.com",
        }
    ],
    "links":{
        "first": "http://example.com/pagination?page=1",
        "last": "http://example.com/pagination?page=1",
        "prev": null,
        "next": null
    },
    "meta":{
        "current_page": 1,
        "from": 1,
        "last_page": 1,
        "path": "http://example.com/pagination",
        "per_page": 15,
        "to": 10,
        "total": 10
    }
}
```

Conditional Attributes

Sometimes you may wish to only include an attribute in a resource response if a given condition is met. For example, you may wish to only include a value if the current user is an "administrator". Laravel provides a variety of helper methods to assist you in this situation. The `when` method may be used to conditionally add an attribute to a resource response:

```
/**
 * Transform the resource into an array.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return array
 */
public function toArray($request)
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->email,
        'secret' => $this->when(Auth::user()->isAdmin(), 'secret-value'),
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ];
}
```

In this example, the `secret` key will only be returned in the final resource response if the authenticated user's `isAdmin` method returns `true`. If the method returns `false`, the `secret` key will be removed from the resource response entirely before it is sent back to the client. The `when` method allows you to expressively define your resources without resorting to conditional statements when building the array.

The `when` method also accepts a Closure as its second argument, allowing you to calculate the resulting value only if the given condition is `true`:

```
'secret' => $this->when(Auth::user()->isAdmin(), function () {
    return 'secret-value';
}),
```

Merging Conditional Attributes

Sometimes you may have several attributes that should only be included in the resource response based on the same condition. In this case, you may use the `mergeWhen` method to include the attributes in the response only when the given condition is `true`:

```
/**
 * Transform the resource into an array.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return array
 */
public function toArray($request)
{
        return [
            'id' => $this->id,
            'name' => $this->name,
            'email' => $this->email,
            $this->mergeWhen(Auth::user()->isAdmin(), [
                'first-secret' => 'value',
                'second-secret' => 'value',
            ]),
            'created_at' => $this->created_at,
            'updated_at' => $this->updated_at,
        ];
    }
```

Again, if the given condition is `false`, these attributes will be removed from the resource response entirely before it is sent to the client.

! The `mergeWhen` method should not be used within arrays that mix string and numeric keys. Furthermore, it should not be used within arrays with numeric keys that are not ordered sequentially.

Conditional Relationships

In addition to conditionally loading attributes, you may conditionally include relationships on your resource responses based on if the relationship has already been loaded on the model. This allows your controller to decide which relationships should be loaded on the model and your resource can easily include them only when they have actually been loaded.

Ultimately, this makes it easier to avoid "N+1" query problems within your resources. The `whenLoaded` method may be used to conditionally load a relationship. In order to avoid unnecessarily loading relationships, this method accepts the name of the relationship instead of the relationship itself:

```
/**
 * Transform the resource into an array.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return array
 */
public function toArray($request)
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->email,
        'posts' => PostResource::collection($this->whenLoaded('posts')),
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ];
}
```

In this example, if the relationship has not been loaded, the `posts` key will be removed from the resource response entirely before it is sent to the client.

Conditional Pivot Information

In addition to conditionally including relationship information in your resource responses, you may conditionally include data from the intermediate tables of many-to-many relationships using the `whenPivotLoaded` method. The `whenPivotLoaded` method accepts the name of the pivot table as its first argument. The second argument should be a Closure that defines the value to be returned if the pivot information is available on the model:

```
/**
 * Transform the resource into an array.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return array
 */
public function toArray($request)
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'expires_at' => $this->whenPivotLoaded('role_user', function () {
            return $this->pivot->expires_at;
        }),
    ];
}
```

If your intermediate table is using an accessor other than `pivot`, you may use the `whenPivotLoadedAs` method:

```
/**
 * Transform the resource into an array.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return array
 */
public function toArray($request)
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'expires_at' => $this->whenPivotLoadedAs('subscription', 'role_user', func
            return $this->subscription->expires_at;
        }),
    ];
}
```

Adding Meta Data

Some JSON API standards require the addition of meta data to your resource and resource collections responses. This often includes things like `links` to the resource or related resources, or meta data about the resource itself. If you need to return additional meta data about a resource, include it in your `toArray` method. For example, you might include `link` information when transforming a resource collection:

```
/**
 * Transform the resource into an array.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return array
 */
public function toArray($request)
{
    return [
        'data' => $this->collection,
        'links' => [
            'self' => 'link-value',
        ],
    ];
}
```

When returning additional meta data from your resources, you never have to worry about accidentally overriding the `links` or `meta` keys that are automatically added by Laravel when returning paginated responses. Any additional `links` you define will be merged with the links provided by the paginator.

Top Level Meta Data

Sometimes you may wish to only include certain meta data with a resource response if the resource is the outer-most resource being returned. Typically, this includes meta information about the response as a whole. To define this meta data, add a `with` method to your resource class. This method should return an array of meta data to be included with the resource response only when the resource is the outer-most resource being rendered:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class UserCollection extends ResourceCollection
{
    /**
     * Transform the resource collection into an array.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return array
     */
    public function toArray($request)
    {
        return parent::toArray($request);
    }

    /**
     * Get additional data that should be returned with the resource array.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return array
     */
    public function with($request)
    {
        return [
            'meta' => [
                'key' => 'value',
            ],
        ];
    }
}
```

Adding Meta Data When Constructing Resources

You may also add top-level data when constructing resource instances in your route or controller. The `additional` method, which is available on all resources, accepts an array of data that should be added to the resource response:

```
return (new UserCollection(User::all()->load('roles')))
                ->additional(['meta' => [
                    'key' => 'value',
                ]]);
```

Resource Responses

As you have already read, resources may be returned directly from routes and controllers:

```
use App\Http\Resources\User as UserResource;
use App\User;

Route::get('/user', function () {
    return new UserResource(User::find(1));
});
```

However, sometimes you may need to customize the outgoing HTTP response before it is sent to the client. There are two ways to accomplish this. First, you may chain the `response` method onto the resource. This method will return an `Illuminate\Http\Response` instance, allowing you full control of the response's headers:

```
use App\Http\Resources\User as UserResource;
use App\User;

Route::get('/user', function () {
    return (new UserResource(User::find(1)))
                ->response()
                ->header('X-Value', 'True');
});
```

Alternatively, you may define a `withResponse` method within the resource itself. This method will be called when the resource is returned as the outer-most resource in a response:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

class User extends JsonResource
{
    /**
     * Transform the resource into an array.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return array
     */
    public function toArray($request)
    {
        return [
            'id' => $this->id,
        ];
    }

    /**
     * Customize the outgoing response for the resource.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Illuminate\Http\Response  $response
     * @return void
     */
    public function withResponse($request, $response)
    {
        $response->header('X-Value', 'True');
    }
}
```

Become a Laravel Partner

Laravel Partners are elite shops providing top-notch Laravel development and consulting. Each of our partners can help you craft a beautiful, well-architected project.

Our Partners

Laravel

Highlights	Resources	Partners	Ecosystem
Release Notes	Laracasts	Vehikl	Vapor
Getting Started	Laravel News	Tighten Co.	Forge
Routing	Laracon	Kirschbaum	Envoyer
Blade Templates	Laracon EU	Byte 5	Horizon
Authentication	Laracon AU	64Robots	Lumen

Laravel is a web application framework with expressive, elegant syntax. We believe development must be an enjoyable and creative experience to be truly fulfilling. Laravel attempts to take the pain out of development by easing common tasks used in most web projects.

Authorization
Artisan Console
Database
Eloquent ORM
Testing

Jobs
Certification
Forums

Cubet
DevSquad
Ideil
Cyber-Duck
ABOUT YOU
Become A Partner

Nova
Echo
Valet
Mix
Spark
Cashier
Homestead
Dusk
Passport
Scout
Socialite