api-skeletons-laravel-hal Documentation

Release latest

Nov 09, 2021

CONTENTS

This library is modeled off of HAL in Apigility written by Zend and myself. Apigility in turn implements HAL according to the specification.

This library is written exclusively for Laravel. It implements the expected `_links` and `_embedded` sections as well as pagination links for supported collections.

CHAPTER

ONE

INSTALLATION

Installation of this module uses composer. For composer documentation, please refer to getcomposer.org.

```
composer require api-skeletons/laravel-hal
```

This library is a collection of classes and does not require a ServiceProvider to be configured.

CHAPTER

TWO

VERSIONS

For PHP 7.0 - 7.3 use version 1.0. For PHP 7.4+ and Laravel 8.0 use the latest version.

CHAPTER

THREE

WITH THANKS

This work was inspired by an article written by Giulio Troccoli-Allard: Using HAL in content APIs

3.1 Overview

3.1.1 Hydrators

Hydrators are not a common programming pattern in Laravel. Hydrators are responsible for moving data into (hydration) and out of (extraction) an object. For their use in this library hydrators are used for extraction only and not hydration.

3.1.2 Hydrator Manager

This library works around a Hydrator Manager which has a configuration of hydrators mapped to the classes they hydrate. The hydrators are individually composed classes thereby allowing any markup you desire, but it is strongly recommended you stick to the HAL specification.

The hydrator manager is used to extract a class using a mapped hydrator.

Note: It is possible to have two fully separate hydrator managers.

3.1.3 ApiSkeletons\Laravel\HAL\Resource

As you extract data from classes using a hydrator you will be creating HAL Resources. A Resource object is the common object used to compose a HAL response. Resource objects hold the object data (the state), links, and embedded data which can also be a Resource. Collections of resources can create an array as a response such as a paginated collection.

When you're ready to turn a resource into HAL JSON you will have a tree of resources.

3.1.4 Abstract Classes

Because it is possible to have multiple hydrator managers an instance of the hydrator manager is assigned to each hydrator and resource. For this reason it is necessary that classes involved in HAL in Laravel extend from certain abstract classes. This is a common pattern in Laravel services.

This is documentation for Hypertext Application Language for Laravel. If you find this useful please add your ★ star to the project.

3.2 Configuration

Create a new directory to hold your hydrators and hydrator manager. These docs recommend `~/app/HAL` and this path will be used throughout this documentation.

Create a new class called `HydratorManager`

```
namespace App\HAL;

use ApiSkeletons\Laravel\HAL\HydratorManager as HALHydratorManager;

final class HydratorManager extends HALHydratorManager
{
    protected array $classHydrators = [
    ];
}
```

Now create your first hydrator

```
namespace App\HAL\Hydrator;

use ApiSkeletons\Laravel\HAL\Hydrator;
use ApiSkeletons\Laravel\HAL\Resource;

final class UserHydrator extends Hydrator
{
    /**
     * The extract function will extract a model into a HAL Resource object
     */
    public function extract($class): Resource
    {
        $data = [];

        $fields = [
            'id',
            'name',
            'created_at',
            'updated_at',
        ];

        // Extract model fields into an array to be used as the resource
        foreach ($fields as $field) {
```

(continues on next page)

(continued from previous page)

```
            $data[$field] = $class->$field;
        }

        // Create a new resource and assign self link
        return $this->hydratorManager->resource($data)
            ->addLink('self', route('hal/user::fetch', $class->id));
    }
}
```

There's a bit going on here. First, we extend from the abstract Hydrator which defines the extract() function and the return value of a Resource.

A simple but effective pattern is used to map the fields we want to the model's properties and assign that to an array. This is the act of extraction. But because this is a HAL hydrator we need to return a Resource. So, using the hydrator manager property of the abstract hydrator we assign the array of data and add a self referential link.

Before we can use this hydrator we must assign it to a model to hydrate from within the hydrator manager

```
protected array $classHydrators = [
  \App\Models\User::class => \App\HAL\Hydrator\UserHydrator::class,
];
```

Having finished these steps we're now ready to return a HAL response for a User model from a controller

```
public function fetch(User $user, Request $request)
{
    $hydratorManger = new HydratorManager();
    return $hydratorManager->extract($user)->toArray();
}
```

The experienced developer will take one look at this and say, "Why aren't you using a facade?" And you're right. It is recommended you use a facade for your hydrator manager. The rest of this documentation will assume the use of a `HALHydratorManager` facade for the hydrator manager.

This is documentation for Hypertext Application Language for Laravel. If you find this useful please add your star to the project.

3.3 Links

HAL defines two specific structures: `_links` and `_embedded`. This document discusses using links, self referential links, related links, and complex links. For pagination links see collections.

Links are URLs to URL resources (not to be confused with a HAL resourcee class). You may create any number of uniquely named links for each HAL Resource. The only strongly-suggested link is the `self` link.

3.3.1 Self Link

Every HAL resource should include a `self` link. The self link reflects the URL to the current resource. This is not just the current requested url: in a collection of resources each resource `self` link is a link to that individual resource.

Every hydrator should assign a `self` link

```
return $this->hydratorManager->resource($data)
    ->addLink('self', route('routeName', $class->id));
```

3.3.2 Related Link

If you are extracting a class with embedded / related data but you do not want to include the embedded class with the HAL response you may choose to just include a link to the API data instead. For an example consider a User > Address 1:1 relationship. When hydrating the User it is not necessary to return the Address every time so just include a link to the data

```
return $this->hydratorManager->resource($data)
    ->addLink('self', route('routeName', $class->id))
    ->addLink('address', route('addressRoute', $class->address->id));
```

3.3.3 Complex Link

By default, when you add a new link, the link is added to the href property of the link. However the HAL specification allows for multiple properties and even arrays of objects. For this reason you may pass an array as a second parameter to `addLink`. The array will be rendered exaclty as it was assigned

```
->addLink('ea:find', ['href' => '/orders{?id}', 'templated' => true]);
```

The special name `curies`, see curie syntax allows for arrays of link data:

```
->addLink('curies', [['name' => 'ea', 'href' => 'http://example.com/docs/rels/{rel}',
↪'templated' => true]]);
```

This is documentation for Hypertext Application Language for Laravel. If you find this useful please add your ★ star to the project.

3.4 Embedded Resources

The HAL specification lays out an `_embedded` section optional for each resource. Instead of using embedded data in your HAL resource state, that kind of data belongs in the `_embedded` section.

THE WRONG WAY:

```
{
  "name": "example",
  "roles": [
    "guest",
    "user",
```

(continues on next page)

(continued from previous page)

```
    "admin"
  ],
  "address": {
    "zipcode": "12345"
  }
}
```

THE HAL WAY:

```
{
  "_links": {
    "self": {
      "href": "https://myapi/name/1"
    }
  }
  "name": "example",
  "_embedded": {
    "roles": [
        {
          "_links": {
            "self": {
              "href": "https://myapi/role/1"
            }
          },
          "id": 1,
          "roleId": "guest"
        },
        {
          "_links": {
            "self": {
              "href": "https://myapi/role/2"
            }
          },
          "id": 2,
          "roleId": "user"
        },
        {
          "_links": {
            "self": {
              "href": "https://myapi/role/3"
            }
          },
          "id": 3,
          "roleId": "admin"
        },
    ],
    "address": {
      "_links": {
        "self": {
          "href": "https://myapi/address/5"
        }
      },
```

(continues on next page)

(continued from previous page)

```
      "id": 5,
      "zipcode": "12345"
    }
  }
}
```

Now you know. The author hopes this lesson has not come too late for you. Your front end consumers of your API will love you for it. They are your client so treat them like a good client.

3.4.1 Embedding Resources

You may embed as many resources as you see fit. Embedded resources should have a relationship with the parent data. Below a related class with its own hydrator is assigned as a resource to the currently extracting class

```
return $this->hydratorManager->resource($data)
    ->addLink('self', route('routeName', $data['id']))
    ->addEmbeddedResource('example', $this->hydratorManager->extract($class->example));
```

This is documentation for Hypertext Application Language for Laravel. If you find this useful please add your ★ star to the project.

3.5 Collecitons and Paginated HAL

3.5.1 Collections

You may assign a collection of resources to the `_embedded` section of any resource. And you can directly assign a collection of mapped [or unmapped!] models to a resource as a collection

```
HALHydratorManager::resource()
  ->addEmbeddedResources('users', HALHydratorManager::extract($model->users))
```

Or

```
HALHydratorManager::resource()
  ->addEmbeddedResources('users', HALHydratorManager::extract($model->users,␣
↪CustomHydrator::class))
```

Or

```
$userCollection = collect();

foreach ($model->users as $user) {
    $userCollection->push(
        HALHydratorManager::extract($user)->toArray()
    );
}

return HALHydratorManager::resource()
    ->addEmbeddedResources('users', $userCollection);
```

Or

```
$userCollection = collect();

foreach ($model->users as $user) {
    $userCollection->push(
        HALHydratorManager::extract($user, CustomHydrator::class)->toArray()
    );
}

return HALHydratorManager::resource()
    ->addEmbeddedResources('users', $userCollection)
    ;
```

3.5.2 Pagination

HAL supports pagination using the `_links` section, `_embedded` section, and includes pagination info in the state. Automatic HAL pagination is supported for the `use Illuminate\Pagination\LengthAwarePaginator` class. This paginator is created from a controller

```
public function fetchAll(Request $request)
{
    $data = DataModel::filtered()->sorted()->paginate(50);

    return HALHydratorManager::paginate('data', $data)->toArray();
}
```

The above example uses the searchable and sortable libraries to turn an api endpoint into a rich database queryable and sortable resources. This technique is strongly encouraged.

3.6 Specifying a Custom Hydrator For Extracting

There are instances where a model may not exist for data such as *pivot* data. There are very good reasons for including pivot data with a HAL response but it has two big drawbacks:

1. It may not have a primary key or a URL it can use as a self referential link.
2. It probably does not have a model which can be associated with a hydrator in the hydrator manager.

For these types of scenarios there is still a way to hydrate the data to HAL by using custom hydrators

```
$hal = HALHydratorManager::extract($pivot, PivotHydrator::class)->toArray();
```

The PivotHydrator does not need to be added to the hydrator manager but it wouldn't hurt. In the above code the $pivot data may be a *stdClass* or a class not mapped in the hydrator manager, but you can still create a resource from it by specifying the hydrator to use.

It is for cases like these that the `self` link is not a required datapoint.

3.7 Manually Creating Resources

Most of the time resources are created within a hydrator. But imagine you want to compose a response which is a combination of unrelated data by including it in the `_embedded` section. Build this inside your controller

```
return HALHydratorManager::resource()
    ->addEmbeddedResource('model1', HALHydratorManager::extract($model1))
    ->addEmbeddedResources('model2', HALHydratorManager::extract($model2))
    ->addLink('self', $request->url())
    ->toArray();
```

This is documentation for Hypertext Application Language for Laravel. If you find this useful please add your star to the project.