[Fractal Output complex, flexible, AJAX/RESTful data structures](http://fractal.thephpleague.com/)

# Introduction

## What is Fractal?

Note:HAL =Hypertext Application Language

Fractal provides a presentation and transformation layer for complex data output, the like found in RESTful APIs, and works really well with JSON. Think of this as a view layer for your JSON/YAML/etc.

When building an API it is common for people to just grab stuff from the database and pass it to json\_encode(). This might be passable for “trivial” APIs but if they are in use by the public, or used by mobile applications then this will quickly lead to inconsistent output.

[Fractal on Packagist](https://packagist.org/packages/league/fractal)

## Goals

* Create a “barrier” between source data and output, so schema changes do not affect users
* Systematic type-casting of data, to avoid foreach()ing through and (bool)ing everything
* Include (a.k.a embedding, nesting or side-loading) relationships for complex data structures
* Work with standards like HAL and JSON-API but also allow custom serialization
* Support the pagination of data results, for small and large data sets alike
* Generally ease the subtle complexities of outputting data in a non-trivial API

## Questions?

Fractal was created by Phil Sturgeon. Find him on Twitter at [@philsturgeon](https://twitter.com/philsturgeon). Fractal is maintained by Andrew Willis and Jason Lewis. They can be found on Twitter at [@ilovefluffy](https://twitter.com/ilovefluffy) and [@jasonclewis](https://twitter.com/jasonclewis).

# Simple Example

For the sake of simplicity, this example has been put together as though it was one file. In reality you would spread the manager initiation, data collection and JSON conversion into separate parts of your application.

<?php

use League\Fractal\Manager;

use League\Fractal\Resource\Collection;

a

// Create a top level instance somewhere

$fractal = new Manager();

// Get data from some sort of source

// Most PHP extensions for SQL engines return everything as a string, historically

// for performance reasons. We will fix this later, but this array represents that.

$books = [

[

'id' => '1',

'title' => 'Hogfather',

'yr' => '1998',

'author\_name' => 'Philip K Dick',

'author\_email' => 'philip@example.org',

],

[

'id' => '2',

'title' => 'Game Of Kill Everyone',

'yr' => '2014',

'author\_name' => 'George R. R. Satan',

'author\_email' => 'george@example.org',

]

];

// Pass this array (collection) into a resource, which will also have a "Transformer"

// This "Transformer" can be a callback or a new instance of a Transformer object

// We type hint for array, because each item in the $books var is an array

$resource = new Collection($books, function(array $book) {

return [

'id' => (int) $book['id'],

'title' => $book['title'],

'year' => (int) $book['yr'],

'author' => [

'name' => $book['author\_name'],

'email' => $book['author\_email'],

],

'links' => [

[

'rel' => 'self',

'uri' => '/books/'.$book['id'],

]

]

];

});

// Turn that into a structured array (handy for XML views or auto-YAML converting)

$array = $fractal->createData($resource)->toArray();

// Turn all of that into a JSON string

echo $fractal->createData($resource)->toJson();

// Outputs: {"data":[{"id":1,"title":"Hogfather","year":1998,"author":{"name":"Philip K Dick","email":"philip@example.org"}},{"id":2,"title":"Game Of Kill Everyone","year":2014,"author":{"name":"George R. R. Satan","email":"george@example.org"}}]}

It is worth noting that callbacks are a fairly shoddy replacement for using real [Transformers](http://fractal.thephpleague.com/transformers). They allow you to reuse transformers and keep your controllers lightweight.

# Installation

## System Requirements

You need **PHP >= 5.4.0** to use league/fractal but the latest stable version of PHP is recommended.

## Composer

Fractal is available on [Packagist](https://packagist.org/packages/league/fractal) and can be installed using [Composer](https://getcomposer.org/):

$ composer require league/fractal

Most modern frameworks will include the Composer autoloader by default, but ensure the following file is included:

<?php

// Include the Composer autoloader

require 'vendor/autoload.php';

## Going Solo

You can also use Fractal without using Composer by registering an autoloader function:

spl\_autoload\_register(function ($class) {

$prefix = 'League\\Fractal\\';

$base\_dir = \_\_DIR\_\_ . '/src/';

$len = strlen($prefix);

if (strncmp($prefix, $class, $len) !== 0) {

// no, move to the next registered autoloader

return;

}

$relative\_class = substr($class, $len);

$file = $base\_dir . str\_replace('\\', '/', $relative\_class) . '.php';

if (file\_exists($file)) {

require $file;

}

});

Or, use any other [PSR-4](http://www.php-fig.org/psr/psr-4/) compatible autoloader.

# Resources

Resources are objects that represent data, and have knowledge of a “Transformer”, which is an object or callback that will know how to output the data.

Two types of resource exist:

* **League\Fractal\Resource\Item** - A singular resource, probably one entry in a data store
* **League\Fractal\Resource\Collection** - A collection of resources

The Item and Collection constructors will take any kind of data you wish to send it as the first argument, and then a “transformer” as the second argument.

These examples use callback transformers instead of creating classes, purely for demonstrative purposes.

### Item Example

use Acme\Model\Book;

use League\Fractal;

$book = Book::find($id);

$resource = new Fractal\Resource\Item($book, function(Book $book) {

return [

'id' => (int) $book->id,

'title' => $book->title,

'year' => (int) $book->yr,

'links' => [

[

'rel' => 'self',

'uri' => '/books/'.$book->id,

]

]

];

});

### Collection Example

use Acme\Model\Book;

use League\Fractal;

$books = Book::all();

$resource = new Fractal\Resource\Collection($books, function(Book $book) {

return [

'id' => (int) $book->id,

'title' => $book->title,

'year' => (int) $book->yr,

'links' => [

[

'rel' => 'self',

'uri' => '/books/'.$book->id,

]

]

];

});

In this example $books is an array of Acme\Model\Book instances, or a collection class that implemented [ArrayIterator](http://php.net/ArrayIterator).

# Serializers

A Serializer structures your Transformed data in certain ways. There are many output structures for APIs, two popular ones being [HAL](http://stateless.co/hal_specification.html) and [JSON-API](http://jsonapi.org/). Twitter and Facebook output data differently to each other, and Google does it differently too. Most of the differences between these serializers are how data is namespaced.

Serializer classes let you switch between various output formats with minimal effect on your Transformers.

A very basic usage of Fractal will look like this, as has been seen in other sections:

use Acme\Model\Book;

use Acme\Transformer\BookTransformer;

use League\Fractal\Manager;

use League\Fractal\Resource\Item;

use League\Fractal\Serializer\DataArraySerializer;

$manager = new Manager();

$manager->setSerializer(new DataArraySerializer());

// Some sort of ORM call

$book = Book::find(1);

// Make a resource out of the data and

$resource = new Item($book, new BookTransformer(), 'book');

// Run all transformers

$manager->createData($resource)->toArray();

// Outputs:

// [

// 'data' => [

// 'id' => 'Foo',

// 'title' => 'Foo',

// 'year' => 1991,

// ],

// ];

What is new here is the $manager->setSerializer(new DataArraySerializer()); part. DataArraySerializer is the name of the default serializer in Fractal, but there are more.

## DataArraySerializer

This serializer is not to everyone’s tastes, because it adds a 'data' namespace to the output:

// Item

[

'data' => [

'foo' => 'bar'

],

];

// Collection

[

'data' => [

[

'foo' => 'bar'

]

],

];

This is handy because it allows space for meta data (like pagination, or totals) in both Items and Collections.

// Item with Meta

[

'data' => [

'foo' => 'bar'

],

'meta' => [

...

]

];

// Collection with Meta

[

'data' => [

[

'foo' => 'bar'

]

],

'meta' => [

...

]

];

This fits in nicely for meta and included resources, using the 'data' namespace. This means meta data can be added for those included resources too.

// Item with included resource using meta

[

'data' => [

'foo' => 'bar'

'comments' => [

'data' => [

...

],

'meta' => [

...

]

]

],

];

## ArraySerializer

Sometimes people want to remove that 'data' namespace for items, and that can be done using the ArraySerializer. This is mostly the same, other than that namespace for items. Collections keep the 'data' namespace to avoid confusing JSON when meta data is added.

use League\Fractal\Serializer\ArraySerializer;

$manager->setSerializer(new ArraySerializer());

// Item

[

'foo' => 'bar'

];

// Collection

[

'data' => [

'foo' => 'bar'

]

];

Meta data is is fine for items, but gets a little confusing for collections:

// Item with Meta

[

'foo' => 'bar'

'meta' => [

...

]

];

// Collection with Meta

[

[

'foo' => 'bar'

]

'meta' => [

...

]

];

Adding a named key to what is otherwise just a list confuses JSON:

{“0”:{“foo”:”bar”},”meta”:{}}

That "0" is there because you cannot mix index keys and non-indexed keys without JSON deciding to make it a structure (object) instead of a list (array).

This is why ArraySerialzier is not recommended, but if you are not using meta data then… carry on.

## JsonApiSerializer

This is a representation of the [JSON-API](http://jsonapi.org/) standard (v1.0). It implements the most common features such as

* Primary Data
* Resource Objects
* Resource Identifier Objects
* Compound Documents
* Meta Information
* Links
* Relationships
* Inclusion of Related Resources

Features that are not yet included

* Sparse Fieldsets
* Sorting
* Pagination
* Filtering

As Fractal is a library to output data structures, the serializer can only transform the content of your HTTP response. Therefore, the following has to be implemented by you

* Content Negotiation
* HTTP Response Codes
* Error Objects

For more information please refer to the official [JSON API specification](http://jsonapi.org/format).

JSON API requires a Resource Key for your resources, as well as an id on every object.

use League\Fractal\Serializer\JsonApiSerializer;

$manager->setSerializer(new JsonApiSerializer());

// Important, notice the Resource Key in the third parameter:

$resource = new Item($book, new JsonApiBookTransformer(), 'books');

$resource = new Collection($books, new JsonApiBookTransformer(), 'books');

The resource key is used to give it a named namespace:

// Item

[

'data' => [

'type' => 'books',

'id' => 1,

'attributes' => [

'foo' => 'bar'

],

],

];

// Collection

[

'data' => [

[

'type' => 'books',

'id' => 1,

'attributes' => [

'foo' => 'bar'

],

]

],

];

Just like DataArraySerializer, this works nicely for meta data:

// Item with Meta

[

'data' => [

'type' => 'books',

'id' => 1,

'attributes' => [

'foo' => 'bar'

]

],

'meta' => [

...

]

];

// Collection with Meta

[

'data' => [

[

'type' => 'books',

'id' => 1,

'attributes' => [

'foo' => 'bar'

]

]

],

'meta' => [

...

]

];

Adding a resource to an item response would look like this:

// Item with a related resource

[

'data' => [

'type' => 'books',

'id' => 1,

'attributes' => [

'foo' => 'bar'

],

'relationships' => [

'author' => [

'data' => [

'type' => 'people',

'id' => '1',

]

]

]

],

'included' => [

[

'type' => 'people',

'id' => 1,

'attributes' => [

'name' => 'Dave'

]

]

]

];

If you want to enable links support, just set a baseUrl on your serializer

use League\Fractal\Serializer\JsonApiSerializer;

$baseUrl = 'http://example.com';

$manager->setSerializer(new JsonApiSerializer($baseUrl));

The same resource as above will look like this

// Item with a related resource and links support

[

'data' => [

'type' => 'books',

'id' => 1,

'attributes' => [

'foo' => 'bar'

],

'links' => [

'self' => 'http://example.com/books/1'

],

'relationships' => [

'author' => [

'links' => [

'self' => 'http://example.com/books/1/relationships/author',

'related' => 'http://example.com/books/1/author'

],

'data' => [

'type' => 'people',

'id' => '1',

]

]

]

],

'included' => [

[

'type' => 'people',

'id' => 1,

'attributes' => [

'name' => 'Dave'

],

'links' => [

'self' => 'http://example.com/people/1'

]

]

]

];

## Custom Serializers

You can make your own Serializers by implementing [SerializerAbstract](https://github.com/thephpleague/fractal/blob/master/src/Serializer/SerializerAbstract.php).

use Acme\Serializer\CustomSerializer;

$manager->setSerializer(new CustomSerializer());

The structure of serializers will change at some point, to allow items and collections to be handled differently and to improve side-loading logic. Keep an eye on the change log, but do not be afraid to make one.

# Transformers

In the [Resources](http://fractal.thephpleague.com/resources/) section the examples show off callbacks for transformers, but these are of limited use:

<?php

use Acme\Model\Book;

use League\Fractal;

$books = Book::all();

$resource = new Fractal\Resource\Collection($books, function(Book $book) {

return [

'id' => (int) $book->id,

'title' => $book->title,

'year' => $book->yr,

'author' => [

'name' => $book->author\_name,

'email' => $book->author\_email,

],

'links' => [

[

'rel' => 'self',

'uri' => '/books/'.$book->id,

]

]

];

});

These can be handy in some situations, but most data will need to be transformed multiple times and in multiple locations, so creating classes to do this work can save code duplication.

## Classes for Transformers

To reuse transformers (recommended) classes can be defined, instantiated and passed in place of the callback.

These classes must extend League\Fractal\TransformerAbstract and contain at the very least a method with the name transform().

The method declaration can take mixed input, just like the callbacks:

<?php

namespace Acme\Transformer;

use Acme\Model\Book;

use League\Fractal;

class BookTransformer extends Fractal\TransformerAbstract

{

public function transform(Book $book)

{

return [

'id' => (int) $book->id,

'title' => $book->title,

'year' => (int) $book->yr,

'links' => [

[

'rel' => 'self',

'uri' => '/books/'.$book->id,

]

],

];

}

}

Once the Transformer class is defined, it can be passed as an instance in the resource constructor.

<?php

use Acme\Transformer\BookTransformer;

use League\Fractal;

$resource = new Fractal\Resource\Item($book, new BookTransformer);

$resource = new Fractal\Resource\Collection($books, new BookTransformer);

## Including Data

Your transformer at this point is mainly just giving you a method to handle array conversion from your data source (or whatever your model is returning) to a simple array. Including data in an intelligent way can be tricky as data can have all sorts of relationships. Many developers try to find a perfect balance between not making too many HTTP requests and not downloading more data than they need to, so flexibility is also important.

Sticking with the book example, the BookTransformer, we might want to normalize our database and take the two author\_\* fields out and put them in their own table. This include can be optional to reduce the size of the JSON response and is defined like so:

<?php namespace App\Transformer;

use Acme\Model\Book;

use League\Fractal\TransformerAbstract;

class BookTransformer extends TransformerAbstract

{

/\*\*

\* List of resources possible to include

\*

\* @var array

\*/

protected $availableIncludes = [

'author'

];

/\*\*

\* Turn this item object into a generic array

\*

\* @return array

\*/

public function transform(Book $book)

{

return [

'id' => (int) $book->id,

'title' => $book->title,

'year' => (int) $book->yr,

'links' => [

[

'rel' => 'self',

'uri' => '/books/'.$book->id,

]

],

];

}

/\*\*

\* Include Author

\*

\* @return League\Fractal\ItemResource

\*/

public function includeAuthor(Book $book)

{

$author = $book->author;

return $this->item($author, new AuthorTransformer);

}

}

These includes will be available but can never be requested unless the Manager::parseIncludes() method is called:

<?php

use League\Fractal;

$fractal = new Fractal\Manager();

if (isset($\_GET['include'])) {

$fractal->parseIncludes($\_GET['include']);

}

With this set, include can do some great stuff. If a client application were to call the URL /books?include=author then they would see author data in the response.

These includes can be nested with dot notation too, to include resources within other resources.

**E.g:** /books?include=author,publishers.somethingelse

Note: publishers will also be included with somethingelse nested under it. This is shorthand for publishers,publishers.somethingelse.

This can be done to a limit of 10 levels. To increase or decrease the level of embedding here, use the Manager::setRecursionLimit(5) method with any number you like, to strip it to that many levels. Maybe 4 or 5 would be a smart number, depending on the API.

### Default Includes

Just like with optional includes, default includes are defined in a property on the transformer:

<?php namespace App\Transformer;

use Acme\Model\Book;

use League\Fractal\TransformerAbstract;

class BookTransformer extends TransformerAbstract

{

/\*\*

\* List of resources to automatically include

\*

\* @var array

\*/

protected $defaultIncludes = [

'author'

];

// ....

/\*\*

\* Include Author

\*

\* @param Book $book

\* @return \League\Fractal\Resource\Item

\*/

public function includeAuthor(Book $book)

{

$author = $book->author;

return $this->item($author, new AuthorTransformer);

}

}

This will look identical in output as if the user requested ?include=author.

### Include Parameters

When including other resources, syntax can be used to provide extra parameters to the include methods. These parameters are constructed in the URL, ?include=comments:limit(5|1):order(created\_at|desc).

This syntax will be parsed and made available through a League\Fractal\ParamBag object, passed into the include method as the second argument.

<?php

use League\Fractal\ParamBag;

// ... transformer stuff ...

private $validParams = ['limit', 'order'];

/\*\*

\* Include Comments

\*

\* @param Book $book

\* @param \League\Fractal\ParamBag

\* @return \League\Fractal\Resource\Item

\*/

public function includeComments(Book $book, ParamBag $params)

{

// Optional params validation

$usedParams = array\_keys(iterator\_to\_array($params));

if ($invalidParams = array\_diff($usedParams, $this->validParams)) {

throw new \Exception(sprintf('Invalid param(s): "%s". Valid param(s): "%s"', implode(',', $usedParams), implode(',', $this->validParams)));

}

// Processing

list($limit, $offset) = $params->get('limit');

list($orderCol, $orderBy) = $params->get('order');

$comments = $book->comments

->take($limit)

->skip($offset)

->orderBy($orderCol, $orderBy)

->get();

return $this->collection($comments, new CommentTransformer);

}

Parameters have a name, then multiple values which are always returned as an array, even if there is only one. They are accessed by the get() method, but array access is also an option, so $params->get('limit') and $params['limit'] do the same thing.

### Eager-Loading vrs Lazy-Loading

This above examples happen to be using the lazy-loading functionality of an ORM for $book->author. Lazy-Loading can be notoriously slow, as each time one item is transformered, it would have to go off and find other data leading to a huge number of SQL requests.

Eager-Loading could easily be used by inspecting the value of $\_GET['include'], and using that to produce a list of relationships to eager-load with an ORM.

# Pagination

When working with a large data set it obviously makes sense to offer pagination options to the endpoint, otherwise that data can get very slow. To avoid writing your own pagination output into every endpoint, Fractal provides you with two solutions:

* Paginator
* Cursor

## Using Paginators

Paginators offer more information about your result-set including total, and have next/previous links which will only show if there is more data available. This intelligence comes at the cost of having to count the number of entries in a database on each call.

For some data sets this might not be an issue, but for some it certainly will. If pure speed is an issue, consider using Cursors instead.

Paginator objects are created, and must implement League\Fractal\Pagination\PaginatorInterface and its specified methods. The instantiated object must then be passed to the League\Fractal\Resource\Collection::setPaginator() method.

Fractal currently ships with the following adapters:

* Laravel’s illuminate/pagination package as League\Fractal\Pagination\IlluminatePaginatorAdapter
* The pagerfanta/pagerfanta package as League\Fractal\Pagination\PagerfantaPaginatorAdapter
* Zend Framework’s zendframework/zend-paginator package as League\Fractal\Pagination\ZendFrameworkPaginatorAdapter

### Laravel Pagination

As an example, you can use Laravel’s Eloquent or Query Builder method paginate() to achieve the following:

use League\Fractal\Resource\Collection;

use League\Fractal\Pagination\IlluminatePaginatorAdapter;

use Acme\Model\Book;

use Acme\Transformer\BookTransformer;

$paginator = Book::paginate();

$books = $paginator->getCollection();

$resource = new Collection($books, new BookTransformer);

$resource->setPaginator(new IlluminatePaginatorAdapter($paginator));

#### Including existing query string values in pagination links

In the example above, previous and next pages will be provided simply with ?page=# ignoring all other existing query strings. To include all query string values automatically in these links we can replace the last line above with:

use Acme\Model\Book;

$year = Input::get('year');

$paginator = Book::where('year', '=', $year)->paginate(20);

$queryParams = array\_diff\_key($\_GET, array\_flip(['page']));

$paginator->appends($queryParams);

$paginatorAdapter = new IlluminatePaginatorAdapter($paginator);

$resource->setPaginator($paginatorAdapter);

## Using Cursors

When we have large sets of data and running a SELECT COUNT(\*) FROM whatever isn’t really an option, we need a proper way of fetching results. One of the approaches is to use cursors that will indicate to your backend where to start fetching results. You can set a new cursor on your collections using the League\Fractal\Resource\Collection::setCursor() method.

The cursor must implement League\Fractal\Pagination\CursorInterface and its specified methods.

Fractal currently ships with a very basic adapter: League\Fractal\Pagination\Cursor. It’s really easy to use:

use Acme\Model\Book;

use Acme\Transformer\BookTransformer;

use League\Fractal\Pagination\Cursor;

use League\Fractal\Resource\Collection;

if ($currentCursorStr = Input::get('cursor', false)) {

$books = Book::where('id', '>', $currentCursorStr)->take(5)->get();

} else {

$books = Book::take(5)->get();

}

$prevCursorStr = Input::get('prevCursor', 6); // If prevCursor is not present, then indicate to fetch the first five results instead.

$newCursorStr = $books->last()->id;

$cursor = new Cursor($currentCursorStr, $prevCursorStr, $newCursorStr, $books->count());

$resource = new Collection($books, new BookTransformer);

$resource->setCursor($cursor);

These examples are for Laravel’s illuminate\database package, but you can do it however you like. The cursor also happens to be constructed from the id field, but it could just as easily be an offset number. Whatever is picked to represent a cursor, maybe consider using base64\_encode() and base64\_decode() on the values to make sure API users do not try and do anything too clever with them. They just need to pass the cursor to the new URL, not do any maths.

# Glosari

Learn more about the general concepts of Fractal.

Cursor

A *cursor* is an unintelligent form of Pagination, which does not require a total count of how much data is in the database. This makes it impossible to know if the "next" page exists, meaning an API client would need to keep making HTTP Requests until no data could be found (404).

Include

Data usually has relationships to other data. Users have posts, posts have comments, comments belong to posts, etc. When represented in RESTful APIs this data is usually "included" (a.k.a embedded or nested) into the resource. A transformer will contain includePosts() methods, which will expect a resource to be returned, so it can be placed inside the parent resource.

Manager

Fractal has a class named *Manager*, which is responsible for maintaining a record of what embedded data has been requested, and converting the nested data into arrays, JSON, YAML, etc. recursively.

Pagination

*Pagination* is the process of dividing content into pages, which in relation to Fractal is done in two alternative ways: *Cursors* and *Paginators*.

Paginator

A *paginator* is an intelligent form of Pagination, which will require a total count of how much data is in the database. This adds a "paginator" item to the response meta data, which will contain next/previous links when applicable.

Resource

A *resource* is an object which acts as a wrapper for generic data. A *resource* will have a *transformer* attached, for when it is eventually transformed ready to be serialized and output.

Serializer

A *Serializer* structures your *Transformed* data in certain ways. There are many output structures for APIs, two popular ones being [HAL](http://stateless.co/hal_specification.html) and [JSON-API](http://jsonapi.org/). Twitter and Facebook output data differently to each other, and Google does it differently too. *Serializers* let you switch between various output formats with minimal effect on your *Transformers*.

Transformer

*Transformers* are classes, or anonymous functions, which are responsible for taking one instance of the resource data and converting it to a basic array. This process is done to obfuscate your data store, avoiding [Object-relational impedance mismatch](https://en.wikipedia.org/wiki/Object-relational_impedance_mismatch) and allowing you to even glue various elements together from different data stores if you wish. The data is taken from these complex data store(s) and made into a format that is more manageable, and ready to be *Serialized*.