Laravel Collections Unraveled

Jeff Madsen

Laravel Collections Unraveled

Jeff Madsen

This book is for sale at http://leanpub.com/laravelcollectionsunraveled

This version was published on 2015-11-17

Contents

Why this “ePamphlet”?

“Laravel Collections are *fantastic*!” I constantly read this tweeted - and couldn’t agree more. More than five dozen useful functions that you might otherwise have to write out yourself - five dozen *tested* functions, important to say - all chain-able, documented and ready to make your life easier.

But why write an eBook about it? (I jokingly call it an “ePamphlet” because of its length, but I suppose eBook is the proper term).

Remember the early days of learning your first programming language? You spent so much time coming up with an algorithm to solve your problem - and then the instructor would put his solution up on the overhead (are you old enough to remember “overheads”?) complete with Built-In-Language-Function-That-Solves-This-Problem()? How you cried, “Unfair! How am I supposed to know that function even *exists*?”

Well, you know it exists by studying the source code. That’s what we are going to do. In the process, we’re going to perhaps improve our basic PHP skills a bit by getting more familiar with ArrayObjects, CachingIterators, IteratorAggregates and some other handy tools. We’re going to not just read through the list of available methods, but come up with some practical examples of when they might be useful to us.

All that seemed too long to write a blog post about, and I find that serial tutorials never really sink in with me personally. Hopefully this will prove useful enough that you’ll keep it handy as a reference item while working and internalizing the contents. As an eBook I can continually improve the examples and let you know when better content is available, or when a new version of Laravel introduces more functionality.

So…let’s jump in!

What are collections?

Just what *are* Laravel Collections, after all? They're *sort of* like Arrays, in that you can add, remove and iterate over the contents. The documentation calls them "convenient wrappers". They have all these extra functions - and you can chain them!

In fact, looking at the `Illuminate\\Support\\Collection` class itself gives us the answer - "class Collection implements ArrayAccess". An ArrayAccess[1] is an interface used primarily to define an ArrayObject[2]. An ArrayObject is the actual class you build and work with. The best definition and examples of these two terms I could find comes from the inestimable Lorna Jane Mitchell's article ArrayAccess vs ArrayObject[3]. Please give it a quick read - it is short, does a great job, and I'm going to reference her second example later in this chapter. So - if you haven't figured it out yet - a Collection is a custom ArrayObject, implementing ArrayAccess and some other interfaces in much the same way.

Let's look at those and have a quick review of what they do so we can better understand the functions themselves when we get to them. If you haven't already, open class Collection `Illuminate\\Support\\Collection`. You'll see it implements: ArrayAccess, Countable, IteratorAggregate, JsonSerializable, Arrayable, and Jsonable.

ArrayAccess - this is what makes a class behave like an array. It forces methods `offsetExists()`, `offsetGet()`, `offsetSet()`, `offsetUnset()`. You'll see those four methods near the bottom of the class file. As a matter of fact, you'll see the exact same thing in the PrettyBasket class from Lorna Jane Mitchell's article - which is why I recommended you read it.

Countable[4] - that other well-known function, count(), comes from here. In PHP, arrays don't actually implement this interface - that's why you don't see `$myArray->count()`. Our ArrayObject, however, has included it.

IteratorAggregate[5] - has the function `getIterator()` and is what lets us iterate over private properties of our objects. For example, in the manual's myData class example, if you removed the IteratorAggregate implementation you would still be able to foreach over the properties. However, if you made them *private* you would not. Implementing this interface gives us that access.

Finally, we have JsonSerializable and Laravel's own Arrayable and Jsonable. These interface's allow us to switch back and forth between json and array style output.

So what do we have at this point? Our own custom ArrayObject class that allows array-like access to items (corresponding to elements in an actual array), can add and remove items, can count how

[1] http://php.net/manual/en/class.arrayaccess.php
[2] http://php.net/manual/en/class.arrayobject.php
[3] http://www.lornajane.net/posts/2011/arrayaccess-vs-arrayobject
[4] http://php.net/manual/en/class.countable.php
[5] http://php.net/manual/en/class.iteratoraggregate.php

many items it holds, can traverse those items, and can return those items in array or json format. Notice, also, that these items are stored and operated on internally as an array.

One thing we do NOT have, however - is a resultset. People often comment on my Collections tips, asking “why not just use a WHERE clause” or other sql solutions. That might be a better solution under certain circumstances, but a Collection is not just a fancy wrapper for a database resultset. You may not even have access to the actual sql, such as when parsing an expensive API response, or you may wish to use these functions for your own internal data processing. So these methods should be looked at in a wider context.

Now we are ready to start looking at the class functions.

Array Mimicking

Let's get into the meat of the subject with the actual functions of the Collection object. I'm going to break them in several arbitrary groups that I think will help us remember what they do …

These first few you might almost think of as "magic function". They are part of the actual construction of the object and not something most people even think about. Do you remember our ArrayAccess we are implementing as a base for this object? This is what makes it all meet the requirements of that interface, and allow it to all work like an array.

It is unlikely you'll ever use these functions directly, unless you wish to inherit from the Collection class for some reason.

```
__construct($items = [])
```

```
offsetExists($key)
```

```
offsetGet($key)
```

```
offsetSet($key, $value)
```

```
offsetUnset($key)
```

These functions work just like their native php counterparts, but as class functions. This means

```
$users->offsetExists(3)
```

gives the same result as

```
array_key_exists(3, $users)
```

Function offsetSet($key, $value) can work two ways:

```
$collection->offsetSet(3, 200);
```

will *replace* the key value at position 3;

```
$collection->offsetSet(null, 200);
```

will act just like

```
$collection[] = 200;
```

and add a new element on the end.

One last "deep dive" into this area - the `__construct`. Notice that the `__construct` likes to start with an array. If you look in the source code, however, you'll see that if it *doesn't* get an array, it will call `getArrayableItems($items)`. This will cast Arrayables, Json objects, or even *other Collections* as an array. Furthermore, anything that *can* be cast as an array *will be* - but not always with the result you were thinking:

```
$collection = collect(11,12,13,14,15);
```

results in:

```
 Collection {#160 #items: array:1 [ 0 => 11 ] }
```

Array Mechanics

Used to format, transform or otherwise manipulate the structure of the $items array. I'm putting caching in here, as well.

`toArray()` - will transform from a Collection back to a straight array.

`toJson($options = 0)` - transforms the Collection to a json encoded string.

`jsonSerialize()` - a wrapper around toArray(). Making it serializeable. That's it.

We won't linger for long over these functions, but I thought it was a good opportunty to point out something you'll notice a lot in the source and might wonder about. Have a look at the actual function `toJson($options = 0)`:

```
 public function toJson($options = 0) { return json_encode($this->toArray(), $options);
}
```

It's true that this function actual handles two things, first converting to an array and then applying the php fumction json_encode(). A few of the functions don't even do that much - they are simple wrappers around php core functions. Why do that?

Chaining! Laravel is not just about RAD, it is about working in a clean, easy to read and work with manner…"eloquent", if you will. You'll notice a very heavy influence from Ruby on Rails when you work with it for a while. Php loves to wrap functions inside of function inside of functions. Laravel would prefer this:

```
$collection= collect([11,12,13,14,15])->shuffle()->toJson();
```

You can read it, left to right, like a book. Nice, don't you agree?

Looking at our interator functions, we have

`getIterator()` - Converts the Collection to an ArrayIterator

`getCachingIterator($flags = CachingIterator::CALL_TOSTRING)` - Converts the Collection to a CachingIterator built around an ArrayIterator

Let's take a look the following little code snippet to understand this better, and see an interesting use.

```
$collection= collect([11,12,13,14,15])->getCachingIterator();
```

This gives as a CachingIterator, which internally holds an ArrayIterator with the items 11,12,13,14,15. Our CachingIterator is *empty*, but our ArrayIterator is set to the first element. Therefore,

```
dump($collection->current()); //yields null

dump($collection->getInnerIterator()->current()); //yields 11, our first element
```

You see that because the outer CachingIterator does not cache anything until it has been "called" by the internal iterator, it is always one step behind the ArrayIterator. This lets us "look ahead" in our ArrayIterator to see what is coming next.

Advance one item and see for yourself:

```
 $collection->next();

dump($collection->current()); // yields 11

dump($collection->getInnerIterator()->current()); // yields 12
```

Neat!

That takes us deep into the internals of the Collection class. Let's move toward more familiar ground now with a look at the "Wrapper" Methods.

Array Wrappers

I've nicknamed this group the "wrappers" because by and large, they imitate the similarly named functions that exist in the PHP language itself. All of these work internally with the `$items` array that is created when you instantiate the Collection, meaning that they can be strung together in a more readable fashion as `collect([])->someMethod()->someOtherMethod()`.

Another thing to note with all of the functions that work with finding or extracting certain keys is, they all tend to funnel through an object called `Illuminate\\Support\\Arr`, using its `get()` method. If you look at this method, you'll see that this is what lets us use dot notation to access our multi-dimensional array keys in ways you are already familiar:

```
$api_key= \Config::get('services.mandrill.secret');
```

This is one of the reasons I enjoy working with Collections; after digging through it for a while and following the different Traits and Interfaces it uses, you start to see how so much of the Core ties together. If you spend enough time you'll see this thread run through the Request object, Eloquent and virtually everywhere else. What seems at first to be a giant codebase slowly condenses down to something much more easy to get your head around.

Let's jump into the methods. Because these are "wrapper" functions, there is a long list of them that I really don't have many Laravel-specific comments to make. The goal here is simply to let you recognize that the functions exists on this class; later we will show some useful examples with many of them. However, there are a few that work a little bit differently than their PHP counterparts that are interesting to take a look at. For example:

```
implode($value, $glue = null)
```

This can work in one of two ways. With normal array contents, the first param is the glue and everything functions as you would expect. But look what happens when you feed it an array of arrays or objects:

```
1  $collection = collect([
2
3      ['field1'=> 11],
4      ['field2'=> 12],
5      ['field1'=> 13],
6      ['field2'=> 14],
7      ['field1'=> 15]
8
9  ])->implode('field1');
10
```

```
dump($collection);

// Result: "111315"
```

The items are first *filtered* and then glued! You can do the same with your database results:

```
$users = App\User:: all();

$collection = collect($users)->implode('first_name', '-');

// Results: "Jeff-Joe-Gerry"
```

This simple tool means you can make one call to your data, then cleanly and easily rework it into different presentations. That may not be a big deal if you have full control over your datasource, but could be a Godsend if you need to make an expensive call to a restrictive api source.

Likewise, these two functions are ones you might prefer to do in the query when you can, but when you can't, their callback ability can be wonderful. Look at these three ways to use `sum()`:

```
echo collect([1,3,5])->sum(); // Result: 9

echo collect([

    ['field1'=> 11],
    ['field2'=> 12],
    ['field1'=> 13],
    ['field2'=> 14],
    ['field1'=> 15]

])->sum('field1');  // Result: 39

echo collect([

    ['field1'=> 11],
    ['field2'=> 12],
    ['field1'=> 13],
    ['field2'=> 14],
    ['field1'=> 15]

])->sum(function($field) {

    if (isset($field['field2'])) return $field['field2'];

});  // Result: 26
```

`max()`, `min()` and `avg()` all work the same way. Let me tell you how much I wish I had these back when I was creating sales and marketing reports! Try this yourself with the `search()` function, as well.

The rest of this list holds no big surprises that I've noticed - they are generally just the array_* version of the native function, written in a clean, chainable fashion to use the internal $items. I'll include them for completeness sake; if you notice anything unusual, let me know and I'll add it in.

```
flip() - array_flip()

keys() - array_keys()

pop() - array_pop()

pull($key, $default = null) - Arr::pull()

prepend($value) - array_unshift()

reduce(callable $callback, $initial = null) - array_reduce()

reverse() - array_reverse()

shift() - array_shift()

diff($items) - array_diff()

slice($offset, $length = null, $preserveKeys = false) - array_slice()

chunk($size, $preserveKeys = false) - array_chunk()

splice($offset, $length = null, $replacement = []) - array_splice()

map(callable $callback) - array_map()

merge($items) - array_merge()

intersect($items) - array_intersect()

values() - array_values()

count() - count()

push($value) - uses $this->offsetSet(), but key is set to null so always adds to end
of the Collection

put($key, $value) - uses $this->offsetSet() , but key is passed so you can update or
add to a particular place in the collection
```

Ways to sort

This is short chapter, but I think it is good to pull these into their own group and investigate them a little bit. Most of this information is already in the documentation and well-explained, but it's a good opportunity to play with a few more examples to see what creative solutions we can come up with.

`shuffle()`

Shuffle just randomizes your items. It will NOT maintain your keys, but unlike the PHP function, it returns a Collection rather than a boolean, which never made any sense anyway.

```
sort(callable $callback = null)
```

```
sortBy($callback, $options = SORT_REGULAR, $descending = false)
```

```
sortByDesc($callback, $options = SORT_REGULAR)
```

To summarize, the differences we have among the three functions is that `sort()` works with a simple array object to sort on the first-level values, either in the default alphanumerical manner or according to the callback you pass in. A very small note about something that is incorrect in the current documentation - the internal sort uses `uasort()`, not `usort`, and so the keys *are maintained*. This is why in the example they show the results with `$collection->values()->all()`, which is all you need to do if you want a new array of the vlaues only.

If you need to work with a *subset* of the $items, i.e., a field on the resultset, then you want to go with `sortBy()` and tell it which field to pluck and sort on. Optionally, instead of the field if you pass it a callable (that's to say, the name of a function to use as a callback, or an anonymous function itself) it will work with your own sorting algorithm.

`sortByDesc()` is merely `sortBy()` with the third param set to "true", for convenience when you don't wish to use the second $options argument.

There's only so much you can demonstrate with sorting all by itself, so let me leave you with one interesting sort and the promise that we'll visit these again combined with other functions.

Marketing is infamous for wanting custom sorts and displays. Imagine this array was full of item data and they wanted it to show up in a particular order on the page, say Bookcase, Desk, Chair…

```
$collection = collect([

    ['name' => 'Desk'],
    ['name' => 'Chair'],
    ['name' => 'Bookcase'],

]);
```

We could use the callback to sort it exactly as they wanted:

```
$sorted = $collection->sortBy(function ($product, $key)
{

    return array_search($product['name'], [1=>'Bookcase', 2=>'Desk', 3=>'Chair']\
);

});

dump($sorted->values()->all());
```

Hopefully that tip was worth the price of admission. On to filters...

Ways to filter

We saw last chapter how we could sort our data, which is handy, but where the real fun begins is when we start filtering the collection items. Most of these methods have certain similarities, the most outstanding being the ability to use a callback function to further modify its behavior. Let's group and compare them as we've been doing.

The first group we'll look at is what I call the "validation" filters. Not really filters in the traditional sense, but rather confirm the existence of certain data.

`isEmpty()`

isEmpty() is a little bit interesting not so much for the function, which is just a wrapper around `empty($items)`, but for how it is and can be used. First, compare the following (using a User id that doesn't exist):

```
$user1 = App\User:: find(100);

dump($user1->isEmpty());
```

with

```
$user2 = App\User:: find([100]);

dump($user2->isEmpty());
```

The first one should error out on you - the result of $user1 is not a collection. In the second one I used a little trick to force the result to be a collection so we could see it work properly. This is one of the small "gotcha's" developers often come across when they first decide to start using the isEmpty() function. Eloquent will, by default, *not* return a single row result type as a collection (as of v5.1 - this is being discussed and always returning a collection is being considered, so watch this space).

Why not just always use empty() and be safe? No reason, really. It's not "better" and doesn't have any obscure, under the hood differences. As a function of an object, you can occasionally do interesting things like a hand-rolled `contains()` function:

```
 $users = App\User::all();

dump($users->where('first_name', 'Jeff')->isEmpty());
```

But this is really just flexibility; there's no particular advantage I can think of for doing so in ordinary work.

`has($key)`

`has()` uses the internal `offsetExists` to determine if the key is there; it does *not* check it for emptiness. In other words, if the key `type` is set to null, or 0, it will still pass this method as true.

Be careful! This is actually very different from the Request::has() function, which *does* check for emptiness and can even check for multiple keys' existence at once.

```
contains($key, $value = null)
```

`contains` looks for values, or key-value pairs. Again, though, it is only returning a boolean about whether or not the value is found, not how many or the records themselves. "Is there a user with first name of Fred?" This will tell us. You may also get more precise in your searches by creating a callback for it to use.

That group is useful for checking for the existence of keys or values, but what about actually pulling them out and using them? Have no fear - this is one of the areas that Laravel Collections really shine!

```
get($key, $default = null)
```

`get()` is the matching pair to `has()`, and also allows for a default value if not found (like you've seen on the Request object).

If you need a full set of data, try:

```
pluck($value, $key = null)
```

This is the equivalent method for `contains()`, in that you tell it the field of the multi-array you want and it will return *all* of those items. This works for a collection of models, of course, so:

```
 $users = App\User::all();

dump($users->pluck('last_name'));
```

Will give you a directory of last names. Signify the key you'd like with the second parameter:

```
$users = App\User::all();

dump($users->pluck('last_name', 'id'));
```

Because one of the most common tasks we have for working with result sets is to create dropdowns, we also have a specialized form of `pluck()` called `lists($value, $key = null)`. There is absolutely no difference - `lists()` merely calls `pluck()` internally. If anything, the reason this second function was created is just to spotlight it for you so you make a habit of using this syntax rather than using a `foreach` loop to specially build your dropdown data.

While `pluck()` is nice for grabbing *all* the last names, sometimes you want the complete records but filtered on a certain name. In this case, try `where()` or `filter`.

‘ $users = AppUser::all();

dump($users->where(‘first_name’, ‘Jeff’)); ‘

is the same as:

```
 dump( $users->filter(function($item){ return ($item['first_name'] == 'Jeff'); }) );
```

As you can see, `filter` offers us much more flexibility, whereas `where` gives us simplicity to remember and use. Because these are Collections, we can chain them together, such as:

```
$users = App\User::all(); dump($users->where('first_name', 'Jeff')->where('active',1));
```

Be careful! This *looks* very much like the Eloquent `where()` function - until you start trying things like `where('last_name', '!=', 'Madsen')` or `whereActive(1)`. You quickly realize this is an array_filter() function and you'll need to write `filter` callbacks for that.

Speaking of `filter` callbacks - you may already know that `array_filter($array);` with no callback will simply filter out empty elements, but the Collection class' `filter` will do the same:

```
dump( collect([null, 2, 0, 6])->filter() ); // Results: array(2, 6)
```

Be careful, though - notice that both the null and the possibly valid `0` elements are removed.

Let's end with a few lesser known functions that might be "just the thing" some day.

`reject($callback)` is simply a `filter()` that returns a set of excluded values. Handy tool for finding outliers for marketing reports, or suspicious user agents.

`unique($key = null)` does what it says for simple arrays, but careful with more complex objects. On a multi-dimensional array, or example, it will essentially "group by" each unique value, taking the *first* value of the other rows.

`keyBy($keyBy)` - allows you to pull out one set of data from your items and make that the key.

`random($amount = 1)` - grab a random element or collection of elements (if more than one is asked for)

`first(callable $callback = null, $default = null)` - grab the head() of a simple array, or the of the filtered result if you use a callback.

`last(callable $callback = null, $default = null)` - grab the tail() of a simple array, or the of the filtered result if you use a callback.

Ways to loop, transform and misc. methods

Let's close things out with a few odds and ends that might be useful to know about.

Grouping:

This is one of my favorites. Remember how you would query all the user types, then loop over them and gather each set of users so you could make nice reports with the proper heading? Forget all that:

```
dump( collect([
    ['name' => 'Jeff', 'type' => 'programmer'],
    ['name' => 'Gerry', 'type' => 'designer'],
    ['name' => 'Joe', 'type' => 'programmer'],
])->groupBy('type') );
```

Looping:

`each(callable $callback)` - this works very much like the javascript version of its name. Instead of creating a loop with

```
foreach ($collection as $item){
    // do something
}
```

we can call it in a more mapping fashion, including chaining:

```
collect([1, 12, 32, 14, 5])->sort()->each( function($item, $key){
    echo $item . '-';
});
```

Another interesting option is `every()`, which allows you to make a subset of the collection:

```
dump( collect([1,2,3,4,5])->every(2,1));
// yields an array(2,4)
```

Pagination:

Laravel has a wonderful set of pagination functions that work with the QueryBuilder and Eloquent. That said, there are times when you may wish to use Illuminate\Pagination\Paginator to create your own paginator. Most likely you will still want to add limits and offsets to your query, but occasionally you may prefer to cache a complex resultset and slice it. `forPage($page, $perPage)` use array_slice() but helps you by automatically calculating the lenths and offsets so you can just work with clean code.

`take($limit)` also uses array_slice(), but in a simpler form to simply bring back the requested number of items. Use a negative $limit to grab from the end.

`forget($keys)` will simply `unset()` the keys from the original collection. If you need to preserve the original, be sure to call this on a copy.

Transformation:

These functions are for "brute force" collection of all the values present, whether a single level deep or multi-dimensional. I imagine their greatest use would be in searching or tracing through a complex multi-level hierarchy.

`collapse()` - turns a collection of arrays into a single collection of all values.

`flatten()` - like `collapse()`, but works on multi-dimensional coleections.

`zip($items)` - will array merge on keys; i.e.,

```
collect( [1,2,3] )->zip( [4,5,6] );
```

```
Results: array( array(1,4), array(2,5), array(3,6) )
```

`transform(callable $callback)` - this is just like `map()`, with the diffeence being that `map()` returns a new collection with the callback results, but `transform()` actually applies them to the original collection.

Support collections vs. Eloquent collections

Working through these examples you're sure to have noticed a lot of familiar method names from other objects. The Request and Config (actually the Illuminate\Config\Repository) are two that should come to mind, but the framework is full of them. However, if you look more closely you'll see that these are nothing more than "cousins" of the Collection - they all implement ArrayAccess and use Illuminate\Support\Arr class, but have no tighter relationship than that.

The Eloquent Collection, on the other hand, is a directly inherited class. All of the methods reviewed in this book are available, although a few have been overridden to function slightly differently, as well as some others added that are only available in the child class. To be honest, this can make things rather confusing at times.

We will limit how deeply we get into the Eloquent Collection class (perhaps a second edition of this book will spend more time here), but there are some areas we definitely want to visit if we are going to to fully master Collections.

Our first indication that something is different can be seen with the following experiment:

```
1  $users = App\User::all();
2  dump($users->find(2));
3
4  $users = collect(App\User::all());
5  dump($users->find(2));
```

The first `dump()` finds the model we are searching for; the second throws an error since Illuminate\Support\Collection::find() doesn't exist. There is no great mystery behind this - Eloquent has its own set of functions and classes, and while we say that `App\User::all()` returns a collection, we really mean the Eloquent Collection subclass is being called. Confusing, though, if you are used to referring to everything as a "collection" without qualification.

While most of the differences are minor and pertain more toward making the collection object work with a group of Eloquent models, there are a few functions that you might want to know about and that are not (currently) documented:

`add()` - this works just as `push()` does, and adds a model onto the end of the collection

`modelKeys()` - this will grab the full set of returned primary keys, either `id` by default or whatever you may have designated in your model.

`only()` and `except()` sound just like the Request object methods of those names, but actually work on the primary keys to filter the models

`withHidden()` - will include hidden fields in the results

To be Continued...

Yes, to be continued. New versions of Laravel will come out; new uses will be found for this material. I hope this was worth the read, and that it has piqued your interest to explore!

Thank you very much for reading.

Jeff Madsen