

Laravel Collections: PHP Arrays On Steroids

Laravel collections are one of the most powerful provisions of the Laravel framework. They are what PHP arrays should be, but better.

Laravel collections are what PHP arrays should be, but better.

In this tutorial, we will be going through a few tricks that may be handy when you are working with collections.

The Collection Class

The `Illuminate\Support\Collection` class provides a convenient wrapper for working with arrays.

The `Collection` class implements some PHP and Laravel interfaces such as :-

- [ArrayAccess](#) - Interface to provide accessing objects as arrays.
- [IteratorAggregate](#) - Interface to create an external iterator.
- [JsonSerializable](#)

You can check out the rest of the implemented interfaces [here](#).

Creating A New Collection

A collection can be created from an array using the `collect()` helper method or by instantiating the `Illuminate\Support\Collection` class.

A really simple example using the `collect()` helper method:

```
$newCollection = collect([1, 2, 3, 4, 5]);
```

And a more hashed out one:

```
<?php

namespace app\Http\Controllers;

use Illuminate\Support\Collection;

class TestController extends Controller
{
    /**
     * Create a new collection using the collect helper method.
     */
}
```

```

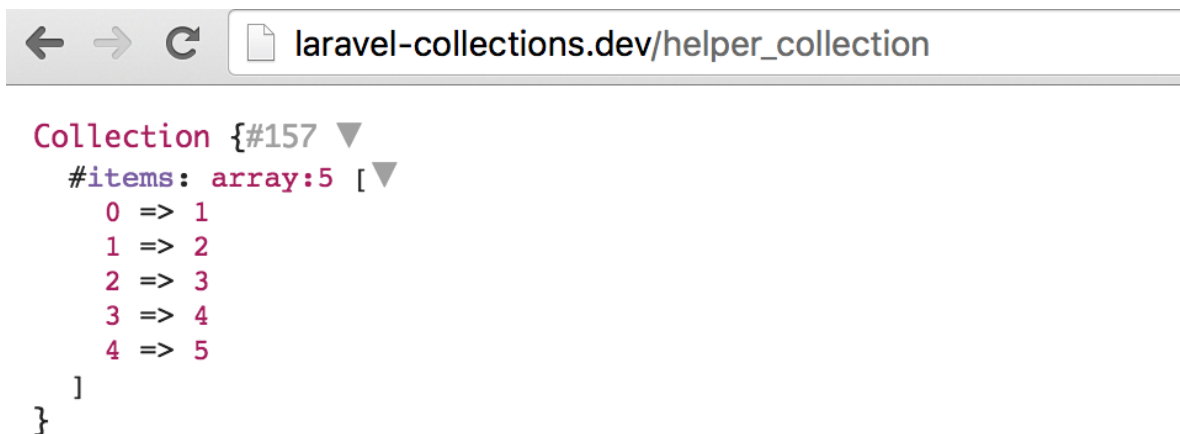
        */
public function helperCollection()
{
    $newCollection = collect([1, 2, 3, 4, 5]);
    dd($newCollection);
}

/**
 * Create a new collection with a Collection class instance.
 */
public function classCollection()
{
    $newCollection = new Collection([1, 2, 3, 4, 5]);
    dd($newCollection);
}
}

```

The helper method is much easier to work with since you do not need to instantiate the `Illuminate\Support\Collection` class.

I also used the `dd()` helper method to display the collection on the browser. This should look a little like this.



[Eloquent ORM Collections](#)

The Laravel Eloquent ORM also returns data as collections.

Eloquent ORM calls return data as collections

To demonstrate such feedback, I will set up an sqlite database.

We will then create a users table with the default migrations that come with Laravel and seed 10 records into the users table.

```

/**
 * Get a list of users from the users table
 */

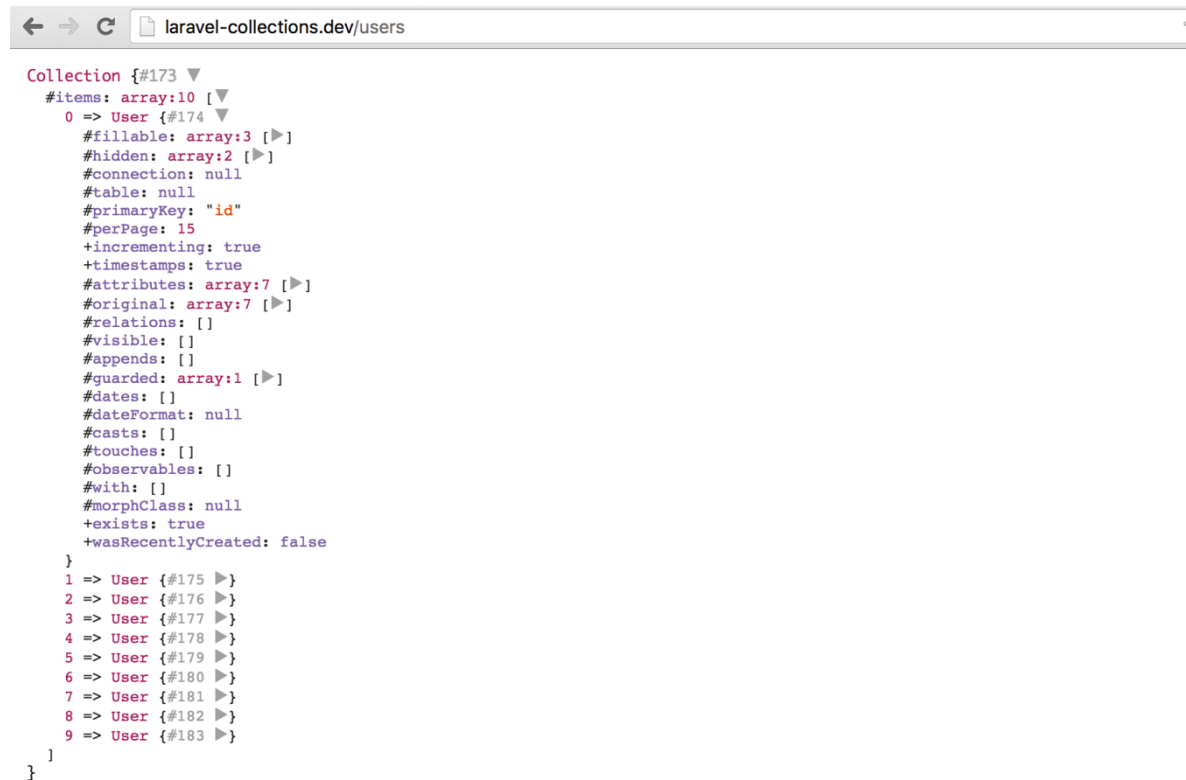
```

```

public function getUsers()
{
    $users = User::all();
    dd($users);
}

```

The controller method below returns a laravel collection with a list of all users as shown below.



The screenshot shows a web browser window with the address bar displaying 'laravel-collections.dev/users'. The main content area shows a Laravel Collection object. The collection has 10 items, each being a User model instance. The first item is expanded, showing its attributes: #fillable: array:3, #hidden: array:2, #connection: null, #table: null, #primaryKey: 'id', #perPage: 15, +incrementing: true, +timestamps: true, #attributes: array:7, #original: array:7, #relations: [], #visible: [], #appends: [], #guarded: array:1, #dates: [], #dateFormat: null, #casts: [], #touches: [], #observables: [], #with: [], #morphClass: null, +exists: true, +wasRecentlyCreated: false. The collection is then shown as an array of 10 User objects, indexed from 0 to 9, with IDs ranging from 174 to 183.

```

Collection {#173 ▼
  #items: array:10 [▼]
    0 => User {#174 ▼
      #fillable: array:3 [▶]
      #hidden: array:2 [▶]
      #connection: null
      #table: null
      #primaryKey: "id"
      #perPage: 15
      +incrementing: true
      +timestamps: true
      #attributes: array:7 [▶]
      #original: array:7 [▶]
      #relations: []
      #visible: []
      #appends: []
      #guarded: array:1 [▶]
      #dates: []
      #dateFormat: null
      #casts: []
      #touches: []
      #observables: []
      #with: []
      #morphClass: null
      +exists: true
      +wasRecentlyCreated: false
    }
    1 => User {#175 ▶}
    2 => User {#176 ▶}
    3 => User {#177 ▶}
    4 => User {#178 ▶}
    5 => User {#179 ▶}
    6 => User {#180 ▶}
    7 => User {#181 ▶}
    8 => User {#182 ▶}
    9 => User {#183 ▶}
  }
}

```

You can then simply access a collection attribute using the arrow notation. For instance, to get the first user's name from the `$users` collection, we can have the following.

```

/**
 * Get the name of the first user
 */
public function firstUser()
{
    $user = User::first();
    dd($user->name);
}

```

Creating Our Sample Collection

We will go through some of the most useful collections tricks that you may find handy.

For the next few sections, I will use the following set of data from the users table and some custom collections for demonstration purposes. While we're creating information here manually, we could also use Laravel's [model factory](#)

```

Array
(
  [0]=>Array
  (
    [id]=>1
    [name]=>Chasity Tillman
    [email]=> qleuschke@example.org
    [age]=>51
    [created_at]=>2016-06-07 15:50:50
    [updated_at]=>2016-06-07 15:50:50
  )
  ...
)

```

Finding Data

There are a number of ways to find data in a collection.

contains

The `contains()` method takes a single value, a key-value pair of parameters or a callback function and returns a boolean value of the value is present in the collection or not.

```

/**
 * Check if a collection contains a given key value pair
 * value or callback parameter
 *
 * @return true or false
 */
public function contains()
{
    $users=User::all();
    $users->contains('name','Chasity Tillman');
    //true

    $collection=collect(['name'=>'John','age'=>23]);
    $collection->contains('Jane');
    //false

    $collection=collect([1,2,3,4,5]);
    $collection->contains(function($key,$value){
        return $value<=5;
    });
    //true
}

```

where

You can use the where method to search a collection by a given key, pair value.

You can also chain your `where()` methods . How cool is that?

```

/**

```

```

        * Use the where method to find data that matches a given
        * criteria.
        *
        * Chain the methods for fine-tuned criteria
        */
public function where()
{
    $users = User::all();
    $user = $users->where('id', 2);
    //Collection of user with an ID of 2

    $user = $users->where('id', 1)
    ->where('age', '51')
    ->where('name', 'Chasity Tillman');

    //collection of user with an id of 1, age 51
    //and named Chasity Tillman
}

```

There are a few more *where-like* methods that I will not delve into but you can check them out on the Laravel documentation.

Most notably, you can take a look at:

- [whereIn\(\)](#) - Takes a key value pair to search but accepts an array of values to search.
- [search\(\)](#) - Searches for a value in a collection, returns the index if the value is present, and `false` if it is not.
- [has\(\)](#) - returns a boolean value if a key value is present in a collection or not.

Filtering Data

You have probably guessed it by now that filtering in collections is done using a `filter()` method.

You have also figured it out on your own that the filter method takes a callback function that subjects our collection to a filter test, right? Right?

```

/**
 * Use the filter method to get a list of all the users that
 * are below the age of 35.
 */
public function filter()
{
    $users = User::all();
    $youngsters = $users->filter(function ($value, $key) {
return $value->age < 35;
    });

    $youngsters->all();
    //list of all users that are below the age of 35
}

```

The filter method takes a key and a value as parameters in the callback function. If your condition tests true, the value is added into the assigned retained.

I have also introduced an `all()` method which returns the underlying array from a collection. Awesome, right? Right?

Sorting / Ordering Data

Collections enable us to sort data using two simple methods :-

- `sortBy()` - Sort data in ascending order given a criteria
- `sortByDesc()` - Sort data in descending order given a criteria

The sort methods takes a key or callback function parameter which is used to sort a collection.

```
/**
 * The sort methods takes a key or callback function parameter
 * which is used to sort a collection.
 */
public function sortData()
{
    $users = User::all();

    $youngestToOldest = $users->sortBy('age');
    $youngestToOldest->all();
    //list of all users from youngest to oldest

    $movies = collect([
        [
            'name' => 'Back To The Future',
            'releases' => [1985, 1989, 1990]
        ],
        [
            'name' => 'Fast and Furious',
            'releases' => [2001, 2003, 2006, 2009, 2011, 2013, 2015, 2017]
        ],
        [
            'name' => 'Speed',
            'releases' => [1994]
        ]
    ]);

    $mostReleases = $movies->sortByDesc(function ($movie, $key) {
return count($movie['releases']);
    });

    $mostReleases->toArray();
    //list of movies in descending order of most releases.

    dd($mostReleases->values()->toArray());
    /*
list of movies in descending order of most releases
```

```

but with the key values reset
    */
}

```

The sort methods maintain the keys for each value. While this may be important for your application, you can reset them to the default zero based incremental values by chaining the `values()` method.

As usual, I have also thrown in a new collection method `toArray()` which simply converts a collection to an array.

Grouping Data

groupBy

Grouping a collection helps makes sense of your data. The `groupBy` method takes either a key or callback function and returns a grouped collection based on on the key value or the returned callback value.

```

/**
 * groupBy returns data grouped based on a key or callback function
 * logic
 */
public function grouping()
{
    $movies = collect([
        ['name' => 'Back To the Future', 'genre' => 'scifi', 'rating'
=> 8],
        ['name' => 'The Matrix', 'genre' => 'fantasy', 'rating' =>
9],
        ['name' => 'The Croods', 'genre' => 'animation', 'rating' =>
8],
        ['name' => 'Zootopia', 'genre' => 'animation', 'rating' => 4],
        ['name' => 'The Jungle Book', 'genre' => 'fantasy', 'rating'
=> 5],
    ]);

    $genre = $movies->groupBy('genre');
    /*
    [
        "scifi" => [
            ["name" => "Back To the Future", "genre" => "scifi",
"rating" => 8,],
        ],
        "fantasy" => [
            ["name" => "The Matrix", "genre" => "fantasy", "rating" =>
9,],
            ["name" => "The Jungle Book", "genre" => "fantasy",
"rating" => 5, ],
        ],
        "animation" => [
            ["name" => "The Croods", "genre" => "animation", "rating"
=> 8,],

```

```

        ["name" => "Zootopia", "genre" => "animation", "rating" =>
4, ],
    ],
    ]
    */

    $rating = $movies->groupBy(function ($movie, $key) {
return $movie['rating'];
    });

    /*
    [
        8 => [
            ["name" => "Back To the Future", "genre" => "scifi",
"rating" => 8, ],
            ["name" => "The Croods", "genre" => "animation", "rating" =>
8, ],
        ],
        9 => [
            ["name" => "The Matrix", "genre" => "fantasy", "rating" =>
9, ],
        ],
        4 => [
            ["name" => "Zootopia", "genre" => "animation", "rating" =>
4, ],
        ],
        5 => [
            ["name" => "The Jungle Book", "genre" => "fantasy", "rating"
=> 5, ],
        ],
    ]
    */
}

```

[Getting A Subset Of Data](#)

Given an array of data, and subsequently a collection, you may want to get a section of the it. This could be:

- The first 2 records
- The last 2 records
- All the records but in groups of 2.

Collections bless us with a few methods that do just that.

take

The take method takes an integer value and returns the specified number of items. Given a negative number, `take()` returns the specified number of items from the end of the collection.

```

/**
 * The take method returns n number of items in a collection.

```



```

        * Given -n, it returns the last n items
        */
public function takeMe()
{
    $list = collect([
        'Albert', 'Ben', 'Charles', 'Dan', 'Eric', 'Xavier', 'Yuri',
        'Zane'
    ]);

    //Get the first two names
    $firstTwo = $list->take(2);
    //['Albert', 'Ben']

    //Get the last two names
    $lastTwo = $list->take(-2);
    //['Yuri', 'Zane']
}

```

chunk

The chunk method breaks a collection into smaller collections of the size n.

```

/**
 * Chunk(n) returns smaller collections of sizes n each from the
 * original collection.
 */
public function chunkMe()
{
    $list = collect([
        'Albert', 'Ben', 'Charles', 'Dan', 'Eric', 'Xavier', 'Yuri',
        'Zane'
    ]);

    $chunks = $list->chunk(3);
    $chunks->toArray();
/*
    [
        ["Albert", "Ben", "Charles",],
        [3 => "Dan", 4 => "Eric", 5 => "Xavier",],
        [6 => "Yuri", 7 => "Zane",],
    ]
    */
}

```

There are quite a number of ways in which this may come in handy.

When you pass the data to a blade view, you can chunk it to get n rows at a time, say, to fit every 3 names into row.

```

@foreach($list->chunk(3) as $names)
<div class="row">
    @foreach($names as $name)
        {{ $name }}
    @endforeach
</div>

```

```
@endforeach
```

You can also do the reverse of `chunk()` by combining smaller collections into a larger one using the `collapse()` method. Check it out [here](#).

Iterating through Data

`map`

The map function iterates through a collection and subjects each value to a callback function.

We will create a collection of peoples names and return a collection of the length of each of the names.

```
/**
 * map function iterates a collection through a callback
 * function and performs an operation on each value.
 */
publicfunctionmapMe()
{
  $names=collect([
    'Albert', 'Ben', 'Charles', 'Dan', 'Eric', 'Xavier', 'Yuri', 'Zane'
  ]);

  $lengths=$names->map(function($name,$key){
    returnstrlen($name);
  });

  $lengths->toArray();
  //[6, 3, 7, 3, 4, 6, 4, 4,]
}
```

`transform`

While the map method creates a new collection, sometimes you would want to edit the original collection. The transform method takes a callback method and performs an action on the same collection.

Since transforming does not create a new collection, you do not need to assign it to a new value.

```
/**
 * Transform perfoms an action on an original collection.
 */
publicfunctiontransformMe()
{
  $names=collect([
    'Albert', 'Ben', 'Charles', 'Dan', 'Eric', 'Xavier', 'Yuri', 'Zane'
  ]);

  $names->transform(function($name,$key){
```

```

returnstrlen($name);
});

$names->toArray();
//[6, 3, 7, 3, 4, 6, 4, 4,]
}

```

reduce

Unlike the map and transform methods, the reduce method returns a single value. It passes the result of each iteration to the next iteration.

For instance, to get the sum of integer values in a collection, reduce passes the sum of subsequent numbers and performs an addition of the result to the next number iteratively.

```

/**
 * Get the sum of numbers in a collection
 */
publicfunctionreduceMe()
{
$numbers=collect([
1,2,3,4,5,6,7,8,9,10
]);

$sum=$numbers->reduce(function($sum,$number){
return$sum+$number;
});
//55
}

```

each

The each method passes each of the items through a callback function.

The most interesting section about the each method is that you can easily break out of the iteration by simply returning false in the callback function.

```

/**
 * Print out a list of numbers that are lesser than or
 * equal to five
 */
publicfunctioneachMethod()
{
$numbers=collect([1,2,3,4,5,6,7,8,9,10]);
$smallNumbers=$numbers->each(function($num,$key){
if($num>5){
returnfalse;
}
echo$num.", ";
});
//1, 2, 3, 4, 5,
}

```

every

The every method creates a new collection that is made up of every n-th element in a collection.

Using A Collection As A Set

The collection class also provides methods that help us work with data as sets. This means we can compare two data sets and perform actions based on that.

union

The union() method is used to add values to a collection from an array. If a value already exists in the collection, the value in the array is ignored.

```
/**
 * add array values to a collection using union
 */
publicfunctionunion()
{
    $coolPeople=collect([
    1=>'John',2=>'James',3=>'Jack'
    ]);

    $allCoolPeople=$coolPeople->union([
    4=>'Sarah',1=>'Susan',5=>'Seyi'
    ]);
    $allCoolPeople->all();
}
/*
    [
        1 => "John", 2 => "James", 3 => "Jack", 4 => "Sarah", 5 =>
    "Seyi",
    ]
 */
}
```

intersect

The intersect() method removes the elements in a collection that are not in a passed array or collection.

```
/**
 * Return a list of very cool people in collection that
 * are in the given array
 */
publicfunctionintersect()
{
    $coolPeople=collect([
    1=>'John',2=>'James',3=>'Jack'
    ]);

    $veryCoolPeople=$coolPeople->intersect(['Sarah','John','James']);
}
```

```
$veryCoolPeople->toArray();  
//[1 => "John" 2 => "James"]  
}
```

You will notice that the intersect method preserves the keys for the returned values.

Conclusion

I have tried to cover most of the collection methods that you might find yourself using but there is still much left out there to learn.

Most notably, I left out the following

- Common maths methods such as [sum](#) and [avg](#).
- Methods that involve updating a collection such as [splice](#), [prepend](#), [push](#) and [pop](#)

There is still much more on the [Laravel documentation](#) and the [Laravel API documentation](#) that you can do with collections that you may want to take a look at.

To follow up on this tutorials code, check out the github repo [here](#). Feel free to make your contributions on it.