Collections

Introduction

The `Illuminate\Support\Collection` class provides a fluent, convenient wrapper for working with arrays of data. For example, check out the following code. We'll use the `collect` helper to create a new collection instance from the array, run the `strtoupper` function on each element, and then remove all empty elements:

```
$collection = collect(['taylor', 'abigail', null])->map(function ($name) {
    return strtoupper($name);
})
->reject(function ($name) {
    return empty($name);
});
```

As you can see, the `Collection` class allows you to chain its methods to perform fluent mapping and reducing of the underlying array. In general, collections are immutable, meaning every `Collection` method returns an entirely new `Collection` instance.

Creating Collections

As mentioned above, the `collect` helper returns a new `Illuminate\Support\Collection` instance for the given array. So, creating a collection is as simple as:

```
$collection = collect([1, 2, 3]);
```

The results of Eloquent queries are always returned as `Collection` instances.

Extending Collections

Collections are "macroable", which allows you to add additional methods to the `Collection` class at run time. For example, the following code adds a `toUpper` method to the `Collection` class:

```
use Illuminate\Support\Str;

Collection::macro('toUpper', function () {
    return $this->map(function ($value) {
        return Str::upper($value);
    });
});

$collection = collect(['first', 'second']);

$upper = $collection->toUpper();

// ['FIRST', 'SECOND']
```

Typically, you should declare collection macros in a service provider.

Available Methods

For the remainder of this documentation, we'll discuss each method available on the `Collection` class. Remember, all of these methods may be chained to fluently manipulate the underlying array. Furthermore, almost every method returns a new `Collection` instance, allowing you to preserve the original copy of the collection when necessary:

Method Listing

`all()`

The `all` method returns the underlying array represented by the collection:

```
collect([1, 2, 3])->all();

// [1, 2, 3]
```

`average()`

Alias for the `avg` method.

`avg()`

The `avg` method returns the average value of a given key:

```
$average = collect([['foo' => 10], ['foo' => 10], ['foo' => 20], ['foo' => 40]])->

// 20

$average = collect([1, 1, 2, 4])->avg();

// 2
```

`chunk()`

The `chunk` method breaks the collection into multiple, smaller collections of a given size:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7]);

$chunks = $collection->chunk(4);

$chunks->toArray();
```

```
// [[1, 2, 3, 4], [5, 6, 7]]
```

This method is especially useful in views when working with a grid system such as Bootstrap. Imagine you have a collection of Eloquent models you want to display in a grid:

```
@foreach ($products->chunk(3) as $chunk)
    <div class="row">
        @foreach ($chunk as $product)
            <div class="col-xs-4">{{ $product->name }}</div>
        @endforeach
    </div>
@endforeach
```

`collapse()`

The `collapse` method collapses a collection of arrays into a single, flat collection:

```
$collection = collect([[1, 2, 3], [4, 5, 6], [7, 8, 9]]);

$collapsed = $collection->collapse();

$collapsed->all();

// [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

`combine()`

The `combine` method combines the values of the collection, as keys, with the values of another array or collection:

```
$collection = collect(['name', 'age']);

$combined = $collection->combine(['George', 29]);

$combined->all();

// ['name' => 'George', 'age' => 29]
```

`collect()`

The `collect` method returns a new `Collection` instance with the items currently in the collection:

```
$collectionA = collect([1, 2, 3]);

$collectionB = $collectionA->collect();

$collectionB->all();

// [1, 2, 3]
```

The `collect` method is primarily useful for converting lazy collections into standard `Collection` instances:

```
$lazyCollection = LazyCollection::make(function () {
    yield 1;
    yield 2;
    yield 3;
});

$collection = $lazyCollection->collect();

get_class($collection);

// 'Illuminate\Support\Collection'

$collection->all();

// [1, 2, 3]
```

The `collect` method is especially useful when you have an instance of `Enumerable` and need a non-lazy collection instance. Since `collect()` is part of the `Enumerable` contract, you can safely use it to get a `Collection` instance.

concat()

The `concat` method appends the given `array` or collection values onto the end of the collection:

```
$collection = collect(['John Doe']);

$concatenated = $collection->concat(['Jane Doe'])->concat(['name' => 'Johnny Doe']

$concatenated->all();

// ['John Doe', 'Jane Doe', 'Johnny Doe']
```

contains()

The `contains` method determines whether the collection contains a given item:

```
$collection = collect(['name' => 'Desk', 'price' => 100]);

$collection->contains('Desk');

// true

$collection->contains('New York');

// false
```

You may also pass a key / value pair to the `contains` method, which will determine if the given pair exists in the collection:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
]);

$collection->contains('product', 'Bookcase');

// false
```

Finally, you may also pass a callback to the `contains` method to perform your own truth test:

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

$collection->contains(function ($value, $key) {
    return $value > 5;
});

// false
```

The `contains` method uses "loose" comparisons when checking item values, meaning a string with an integer value will be considered equal to an integer of the same value. Use the `containsStrict` method to filter using "strict" comparisons.

containsStrict()

This method has the same signature as the `contains` method; however, all values are compared using "strict" comparisons.

count()

The `count` method returns the total number of items in the collection:

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

$collection->count();

// 4
```

countBy()

The `countBy` method counts the occurrences of values in the collection. By default, the method counts the occurrences of every element:

```php
$collection = collect([1, 2, 2, 2, 3]);

$counted = $collection->countBy();

$counted->all();

// [1 => 1, 2 => 3, 3 => 1]
```

However, you pass a callback to the `countBy` method to count all items by a custom value:

```php
$collection = collect(['alice@gmail.com', 'bob@yahoo.com', 'carlos@gmail.com']);

$counted = $collection->countBy(function ($email) {
    return substr(strrchr($email, "@"), 1);
});

$counted->all();

// ['gmail.com' => 2, 'yahoo.com' => 1]
```

`crossJoin()`

The `crossJoin` method cross joins the collection's values among the given arrays or collections, returning a Cartesian product with all possible permutations:

```php
$collection = collect([1, 2]);

$matrix = $collection->crossJoin(['a', 'b']);

$matrix->all();

/*
    [
        [1, 'a'],
        [1, 'b'],
        [2, 'a'],
        [2, 'b'],
    ]
*/

$collection = collect([1, 2]);

$matrix = $collection->crossJoin(['a', 'b'], ['I', 'II']);

$matrix->all();

/*
    [
        [1, 'a', 'I'],
        [1, 'a', 'II'],
        [1, 'b', 'I'],
        [1, 'b', 'II'],
        [2, 'a', 'I'],
        [2, 'a', 'II'],
        [2, 'b', 'I'],
        [2, 'b', 'II'],
    ]
*/
```

`dd()`

The `dd` method dumps the collection's items and ends execution of the script:

```php
$collection = collect(['John Doe', 'Jane Doe']);

$collection->dd();

/*
    Collection {
        #items: array:2 [
            0 => "John Doe"
            1 => "Jane Doe"
        ]
    }
*/
```

If you do not want to stop executing the script, use the `dump` method instead.

`diff()`

The `diff` method compares the collection against another collection or a plain PHP `array` based on its values. This method will return the values in the original collection

that are not present in the given collection:

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

$diff = $collection->diff([2, 4, 6, 8]);

$diff->all();

// [1, 3, 5]
```

`diffAssoc()`

The `diffAssoc` method compares the collection against another collection or a plain PHP `array` based on its keys and values. This method will return the key / value pairs in the original collection that are not present in the given collection:

```
$collection = collect([
    'color' => 'orange',
    'type' => 'fruit',
    'remain' => 6
]);

$diff = $collection->diffAssoc([
    'color' => 'yellow',
    'type' => 'fruit',
    'remain' => 3,
    'used' => 6,
]);

$diff->all();

// ['color' => 'orange', 'remain' => 6]
```

`diffKeys()`

The `diffKeys` method compares the collection against another collection or a plain PHP `array` based on its keys. This method will return the key / value pairs in the original collection that are not present in the given collection:

```
$collection = collect([
    'one' => 10,
    'two' => 20,
    'three' => 30,
    'four' => 40,
    'five' => 50,
]);

$diff = $collection->diffKeys([
    'two' => 2,
    'four' => 4,
    'six' => 6,
    'eight' => 8,
]);

$diff->all();

// ['one' => 10, 'three' => 30, 'five' => 50]
```

`dump()`

The `dump` method dumps the collection's items:

```
$collection = collect(['John Doe', 'Jane Doe']);

$collection->dump();

/*
    Collection {
        #items: array:2 [
            0 => "John Doe"
            1 => "Jane Doe"
        ]
    }
*/
```

If you want to stop executing the script after dumping the collection, use the `dd` method instead.

`duplicates()`

The `duplicates` method retrieves and returns duplicate values from the collection:

```php
$collection = collect(['a', 'b', 'a', 'c', 'b']);

$collection->duplicates();

// [2 => 'a', 4 => 'b']
```

If the collection contains arrays or objects, you can pass the key of the attributes that you wish to check for duplicate values:

```php
$employees = collect([
    ['email' => 'abigail@example.com', 'position' => 'Developer'],
    ['email' => 'james@example.com', 'position' => 'Designer'],
    ['email' => 'victoria@example.com', 'position' => 'Developer'],
])

$employees->duplicates('position');

// [2 => 'Developer']
```

`duplicatesStrict()`

This method has the same signature as the `duplicates` method; however, all values are compared using "strict" comparisons.

`each()`

The `each` method iterates over the items in the collection and passes each item to a callback:

```php
$collection->each(function ($item, $key) {
    //
});
```

If you would like to stop iterating through the items, you may return `false` from your callback:

```php
$collection->each(function ($item, $key) {
    if (/* some condition */) {
        return false;
    }
});
```

`eachSpread()`

The `eachSpread` method iterates over the collection's items, passing each nested item value into the given callback:

```php
$collection = collect([['John Doe', 35], ['Jane Doe', 33]]);

$collection->eachSpread(function ($name, $age) {
    //
});
```

You may stop iterating through the items by returning `false` from the callback:

```php
$collection->eachSpread(function ($name, $age) {
    return false;
});
```

`every()`

The `every` method may be used to verify that all elements of a collection pass a given truth test:

```php
collect([1, 2, 3, 4])->every(function ($value, $key) {
    return $value > 2;
});

// false
```

If the collection is empty, `every` will return true:

```php
$collection = collect([]);
```

```
$collection->every(function($value, $key) {
    return $value > 2;
});

// true
```

`except()`

The `except` method returns all items in the collection except for those with the specified keys:

```
$collection = collect(['product_id' => 1, 'price' => 100, 'discount' => false]);

$filtered = $collection->except(['price', 'discount']);

$filtered->all();

// ['product_id' => 1]
```

For the inverse of `except`, see the only method.

`filter()`

The `filter` method filters the collection using the given callback, keeping only those items that pass a given truth test:

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

$filtered = $collection->filter(function ($value, $key) {
    return $value > 2;
});

$filtered->all();

// [3, 4]
```

If no callback is supplied, all entries of the collection that are equivalent to `false` will be removed:

```
$collection = collect([1, 2, 3, null, false, '', 0, []]);

$collection->filter()->all();

// [1, 2, 3]
```

For the inverse of `filter`, see the reject method.

`first()`

The `first` method returns the first element in the collection that passes a given truth test:

```
collect([1, 2, 3, 4])->first(function ($value, $key) {
    return $value > 2;
});

// 3
```

You may also call the `first` method with no arguments to get the first element in the collection. If the collection is empty, `null` is returned:

```
collect([1, 2, 3, 4])->first();

// 1
```

`firstWhere()`

The `firstWhere` method returns the first element in the collection with the given key / value pair:

```
$collection = collect([
    ['name' => 'Regena', 'age' => null],
    ['name' => 'Linda', 'age' => 14],
    ['name' => 'Diego', 'age' => 23],
    ['name' => 'Linda', 'age' => 84],
```

```
]);

$collection->firstWhere('name', 'Linda');

// ['name' => 'Linda', 'age' => 14]
```

You may also call the `firstWhere` method with an operator:

```
$collection->firstWhere('age', '>=', 18);

// ['name' => 'Diego', 'age' => 23]
```

Like the where method, you may pass one argument to the `firstWhere` method. In this scenario, the `firstWhere` method will return the first item where the given item key's value is "truthy":

```
$collection->firstWhere('age');

// ['name' => 'Linda', 'age' => 14]
```

`flatMap()`

The `flatMap` method iterates through the collection and passes each value to the given callback. The callback is free to modify the item and return it, thus forming a new collection of modified items. Then, the array is flattened by a level:

```
$collection = collect([
    ['name' => 'Sally'],
    ['school' => 'Arkansas'],
    ['age' => 28]
]);

$flattened = $collection->flatMap(function ($values) {
    return array_map('strtoupper', $values);
});

$flattened->all();

// ['name' => 'SALLY', 'school' => 'ARKANSAS', 'age' => '28'];
```

`flatten()`

The `flatten` method flattens a multi-dimensional collection into a single dimension:

```
$collection = collect(['name' => 'taylor', 'languages' => ['php', 'javascript']]);

$flattened = $collection->flatten();

$flattened->all();

// ['taylor', 'php', 'javascript'];
```

You may optionally pass the function a "depth" argument:

```
$collection = collect([
    'Apple' => [
        ['name' => 'iPhone 6S', 'brand' => 'Apple'],
    ],
    'Samsung' => [
        ['name' => 'Galaxy S7', 'brand' => 'Samsung']
    ],
]);

$products = $collection->flatten(1);

$products->values()->all();

/*
    [
        ['name' => 'iPhone 6S', 'brand' => 'Apple'],
        ['name' => 'Galaxy S7', 'brand' => 'Samsung'],
    ]
*/
```

In this example, calling `flatten` without providing the depth would have also flattened the nested arrays, resulting in `['iPhone 6S', 'Apple', 'Galaxy S7', 'Samsung']`. Providing a depth allows you to restrict the levels of nested arrays that will be flattened.

`flip()`

The `flip` method swaps the collection's keys with their corresponding values:

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$flipped = $collection->flip();

$flipped->all();

// ['taylor' => 'name', 'laravel' => 'framework']
```

`forget()`

The `forget` method removes an item from the collection by its key:

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$collection->forget('name');

$collection->all();

// ['framework' => 'laravel']
```

Unlike most other collection methods, `forget` does not return a new modified collection; it modifies the collection it is called on.

`forPage()`

The `forPage` method returns a new collection containing the items that would be present on a given page number. The method accepts the page number as its first argument and the number of items to show per page as its second argument:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9]);

$chunk = $collection->forPage(2, 3);

$chunk->all();

// [4, 5, 6]
```

`get()`

The `get` method returns the item at a given key. If the key does not exist, `null` is returned:

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$value = $collection->get('name');

// taylor
```

You may optionally pass a default value as the second argument:

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$value = $collection->get('foo', 'default-value');

// default-value
```

You may even pass a callback as the default value. The result of the callback will be returned if the specified key does not exist:

```
$collection->get('email', function () {
    return 'default-value';
});

// default-value
```

`groupBy()`

The `groupBy` method groups the collection's items by a given key:

```php
$collection = collect([
    ['account_id' => 'account-x10', 'product' => 'Chair'],
    ['account_id' => 'account-x10', 'product' => 'Bookcase'],
    ['account_id' => 'account-x11', 'product' => 'Desk'],
]);

$grouped = $collection->groupBy('account_id');

$grouped->toArray();

/*
    [
        'account-x10' => [
            ['account_id' => 'account-x10', 'product' => 'Chair'],
            ['account_id' => 'account-x10', 'product' => 'Bookcase'],
        ],
        'account-x11' => [
            ['account_id' => 'account-x11', 'product' => 'Desk'],
        ],
    ]
*/
```

Instead of passing a string `key`, you may pass a callback. The callback should return the value you wish to key the group by:

```php
$grouped = $collection->groupBy(function ($item, $key) {
    return substr($item['account_id'], -3);
});

$grouped->toArray();

/*
    [
        'x10' => [
            ['account_id' => 'account-x10', 'product' => 'Chair'],
            ['account_id' => 'account-x10', 'product' => 'Bookcase'],
        ],
        'x11' => [
            ['account_id' => 'account-x11', 'product' => 'Desk'],
        ],
    ]
*/
```

Multiple grouping criteria may be passed as an array. Each array element will be applied to the corresponding level within a multi-dimensional array:

```php
$data = new Collection([
    10 => ['user' => 1, 'skill' => 1, 'roles' => ['Role_1', 'Role_3']],
    20 => ['user' => 2, 'skill' => 1, 'roles' => ['Role_1', 'Role_2']],
    30 => ['user' => 3, 'skill' => 2, 'roles' => ['Role_1']],
    40 => ['user' => 4, 'skill' => 2, 'roles' => ['Role_2']],
]);

$result = $data->groupBy([
    'skill',
    function ($item) {
        return $item['roles'];
    },
], $preserveKeys = true);

/*
[
    1 => [
        'Role_1' => [
            10 => ['user' => 1, 'skill' => 1, 'roles' => ['Role_1', 'Role_3']],
            20 => ['user' => 2, 'skill' => 1, 'roles' => ['Role_1', 'Role_2']],
        ],
        'Role_2' => [
            20 => ['user' => 2, 'skill' => 1, 'roles' => ['Role_1', 'Role_2']],
        ],
        'Role_3' => [
            10 => ['user' => 1, 'skill' => 1, 'roles' => ['Role_1', 'Role_3']],
        ],
    ],
    2 => [
        'Role_1' => [
            30 => ['user' => 3, 'skill' => 2, 'roles' => ['Role_1']],
        ],
        'Role_2' => [
            40 => ['user' => 4, 'skill' => 2, 'roles' => ['Role_2']],
        ],
    ],
];
*/
```

`has()`

The `has` method determines if a given key exists in the collection:

```
$collection = collect(['account_id' => 1, 'product' => 'Desk', 'amount' => 5]);

$collection->has('product');

// true

$collection->has(['product', 'amount']);

// true

$collection->has(['amount', 'price']);

// false
```

`implode()`

The `implode` method joins the items in a collection. Its arguments depend on the type of items in the collection. If the collection contains arrays or objects, you should pass the key of the attributes you wish to join, and the "glue" string you wish to place between the values:

```
$collection = collect([
    ['account_id' => 1, 'product' => 'Desk'],
    ['account_id' => 2, 'product' => 'Chair'],
]);

$collection->implode('product', ', ');

// Desk, Chair
```

If the collection contains simple strings or numeric values, pass the "glue" as the only argument to the method:

```
collect([1, 2, 3, 4, 5])->implode('-');

// '1-2-3-4-5'
```

`intersect()`

The `intersect` method removes any values from the original collection that are not present in the given `array` or collection. The resulting collection will preserve the original collection's keys:

```
$collection = collect(['Desk', 'Sofa', 'Chair']);

$intersect = $collection->intersect(['Desk', 'Chair', 'Bookcase']);

$intersect->all();

// [0 => 'Desk', 2 => 'Chair']
```

`intersectByKeys()`

The `intersectByKeys` method removes any keys from the original collection that are not present in the given `array` or collection:

```
$collection = collect([
    'serial' => 'UX301', 'type' => 'screen', 'year' => 2009
]);

$intersect = $collection->intersectByKeys([
    'reference' => 'UX404', 'type' => 'tab', 'year' => 2011
]);

$intersect->all();

// ['type' => 'screen', 'year' => 2009]
```

`isEmpty()`

The `isEmpty` method returns `true` if the collection is empty; otherwise, `false` is returned:

```
collect([])->isEmpty();
```

```
// true
```

isNotEmpty()

The `isNotEmpty` method returns `true` if the collection is not empty; otherwise, `false` is returned:

```
collect([])->isNotEmpty();

// false
```

join()

The `join` method joins the collection's values with a string:

```
collect(['a', 'b', 'c'])->join(', '); // 'a, b, c'
collect(['a', 'b', 'c'])->join(', ', ', and '); // 'a, b, and c'
collect(['a', 'b'])->join(', ', ' and '); // 'a and b'
collect(['a'])->join(', ', ' and '); // 'a'
collect([])->join(', ', ' and '); // ''
```

keyBy()

The `keyBy` method keys the collection by the given key. If multiple items have the same key, only the last one will appear in the new collection:

```
$collection = collect([
    ['product_id' => 'prod-100', 'name' => 'Desk'],
    ['product_id' => 'prod-200', 'name' => 'Chair'],
]);

$keyed = $collection->keyBy('product_id');

$keyed->all();

/*
    [
        'prod-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],
        'prod-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],
    ]
*/
```

You may also pass a callback to the method. The callback should return the value to key the collection by:

```
$keyed = $collection->keyBy(function ($item) {
    return strtoupper($item['product_id']);
});

$keyed->all();

/*
    [
        'PROD-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],
        'PROD-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],
    ]
*/
```

keys()

The `keys` method returns all of the collection's keys:

```
$collection = collect([
    'prod-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],
    'prod-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],
]);

$keys = $collection->keys();

$keys->all();

// ['prod-100', 'prod-200']
```

last()

The `last` method returns the last element in the collection that passes a given truth test:

```php
collect([1, 2, 3, 4])->last(function ($value, $key) {
    return $value < 3;
});

// 2
```

You may also call the `last` method with no arguments to get the last element in the collection. If the collection is empty, `null` is returned:

```php
collect([1, 2, 3, 4])->last();

// 4
```

`macro()`

The static `macro` method allows you to add methods to the `Collection` class at run time. Refer to the documentation on extending collections for more information.

`make()`

The static `make` method creates a new collection instance. See the Creating Collections section.

`map()`

The `map` method iterates through the collection and passes each value to the given callback. The callback is free to modify the item and return it, thus forming a new collection of modified items:

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

$multiplied = $collection->map(function ($item, $key) {
    return $item * 2;
});

$multiplied->all();

// [2, 4, 6, 8, 10]
```

Like most other collection methods, `map` returns a new collection instance; it does not modify the collection it is called on. If you want to transform the original collection, use the `transform` method.

`mapInto()`

The `mapInto()` method iterates over the collection, creating a new instance of the given class by passing the value into the constructor:

```php
class Currency
{
    /**
     * Create a new currency instance.
     *
     * @param  string  $code
     * @return void
     */
    function __construct(string $code)
    {
        $this->code = $code;
    }
}

$collection = collect(['USD', 'EUR', 'GBP']);

$currencies = $collection->mapInto(Currency::class);

$currencies->all();

// [Currency('USD'), Currency('EUR'), Currency('GBP')]
```

`mapSpread()`

The `mapSpread` method iterates over the collection's items, passing each nested item value into the given callback. The callback is free to modify the item and return it, thus forming a new collection of modified items:

```php
$collection = collect([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]);

$chunks = $collection->chunk(2);

$sequence = $chunks->mapSpread(function ($even, $odd) {
    return $even + $odd;
});

$sequence->all();

// [1, 5, 9, 13, 17]
```

`mapToGroups()`

The `mapToGroups` method groups the collection's items by the given callback. The callback should return an associative array containing a single key / value pair, thus forming a new collection of grouped values:

```php
$collection = collect([
    [
        'name' => 'John Doe',
        'department' => 'Sales',
    ],
    [
        'name' => 'Jane Doe',
        'department' => 'Sales',
    ],
    [
        'name' => 'Johnny Doe',
        'department' => 'Marketing',
    ]
]);

$grouped = $collection->mapToGroups(function ($item, $key) {
    return [$item['department'] => $item['name']];
});

$grouped->toArray();

/*
    [
        'Sales' => ['John Doe', 'Jane Doe'],
        'Marketing' => ['Johnny Doe'],
    ]
*/

$grouped->get('Sales')->all();

// ['John Doe', 'Jane Doe']
```

`mapWithKeys()`

The `mapWithKeys` method iterates through the collection and passes each value to the given callback. The callback should return an associative array containing a single key / value pair:

```php
$collection = collect([
    [
        'name' => 'John',
        'department' => 'Sales',
        'email' => 'john@example.com'
    ],
    [
        'name' => 'Jane',
        'department' => 'Marketing',
        'email' => 'jane@example.com'
    ]
]);

$keyed = $collection->mapWithKeys(function ($item) {
    return [$item['email'] => $item['name']];
});

$keyed->all();

/*
    [
        'john@example.com' => 'John',
        'jane@example.com' => 'Jane',
    ]
*/
```

`max()`

The `max` method returns the maximum value of a given key:

```
$max = collect([['foo' => 10], ['foo' => 20]])->max('foo');

// 20

$max = collect([1, 2, 3, 4, 5])->max();

// 5
```

`median()`

The `median` method returns the median value of a given key:

```
$median = collect([['foo' => 10], ['foo' => 10], ['foo' => 20], ['foo' => 40]])->m

// 15

$median = collect([1, 1, 2, 4])->median();

// 1.5
```

`merge()`

The `merge` method merges the given array or collection with the original collection. If a string key in the given items matches a string key in the original collection, the given items's value will overwrite the value in the original collection:

```
$collection = collect(['product_id' => 1, 'price' => 100]);

$merged = $collection->merge(['price' => 200, 'discount' => false]);

$merged->all();

// ['product_id' => 1, 'price' => 200, 'discount' => false]
```

If the given items's keys are numeric, the values will be appended to the end of the collection:

```
$collection = collect(['Desk', 'Chair']);

$merged = $collection->merge(['Bookcase', 'Door']);

$merged->all();

// ['Desk', 'Chair', 'Bookcase', 'Door']
```

`mergeRecursive()`

The `mergeRecursive` method merges the given array or collection recursively with the original collection. If a string key in the given items matches a string key in the original collection, then the values for these keys are merged together into an array, and this is done recursively:

```
$collection = collect(['product_id' => 1, 'price' => 100]);

$merged = $collection->mergeRecursive(['product_id' => 2, 'price' => 200, 'discoun

$merged->all();

// ['product_id' => [1, 2], 'price' => [100, 200], 'discount' => false]
```

`min()`

The `min` method returns the minimum value of a given key:

```
$min = collect([['foo' => 10], ['foo' => 20]])->min('foo');

// 10

$min = collect([1, 2, 3, 4, 5])->min();

// 1
```

`mode()`

The `mode` method returns the mode value of a given key:

```
$mode = collect([['foo' => 10], ['foo' => 10], ['foo' => 20], ['foo' => 40]])->mod

// [10]

$mode = collect([1, 1, 2, 4])->mode();

// [1]
```

`nth()`

The `nth` method creates a new collection consisting of every n-th element:

```
$collection = collect(['a', 'b', 'c', 'd', 'e', 'f']);

$collection->nth(4);

// ['a', 'e']
```

You may optionally pass an offset as the second argument:

```
$collection->nth(4, 1);

// ['b', 'f']
```

`only()`

The `only` method returns the items in the collection with the specified keys:

```
$collection = collect(['product_id' => 1, 'name' => 'Desk', 'price' => 100, 'disco

$filtered = $collection->only(['product_id', 'name']);

$filtered->all();

// ['product_id' => 1, 'name' => 'Desk']
```

For the inverse of `only`, see the except method.

`pad()`

The `pad` method will fill the array with the given value until the array reaches the specified size. This method behaves like the array_pad PHP function.

To pad to the left, you should specify a negative size. No padding will take place if the absolute value of the given size is less than or equal to the length of the array:

```
$collection = collect(['A', 'B', 'C']);

$filtered = $collection->pad(5, 0);

$filtered->all();

// ['A', 'B', 'C', 0, 0]

$filtered = $collection->pad(-5, 0);

$filtered->all();

// [0, 0, 'A', 'B', 'C']
```

`partition()`

The `partition` method may be combined with the `list` PHP function to separate elements that pass a given truth test from those that do not:

```
$collection = collect([1, 2, 3, 4, 5, 6]);

list($underThree, $equalOrAboveThree) = $collection->partition(function ($i) {
    return $i < 3;
});

$underThree->all();

// [1, 2]
```

```php
$equalOrAboveThree->all();

// [3, 4, 5, 6]
```

`pipe()`

The `pipe` method passes the collection to the given callback and returns the result:

```php
$collection = collect([1, 2, 3]);

$piped = $collection->pipe(function ($collection) {
    return $collection->sum();
});

// 6
```

`pluck()`

The `pluck` method retrieves all of the values for a given key:

```php
$collection = collect([
    ['product_id' => 'prod-100', 'name' => 'Desk'],
    ['product_id' => 'prod-200', 'name' => 'Chair'],
]);

$plucked = $collection->pluck('name');

$plucked->all();

// ['Desk', 'Chair']
```

You may also specify how you wish the resulting collection to be keyed:

```php
$plucked = $collection->pluck('name', 'product_id');

$plucked->all();

// ['prod-100' => 'Desk', 'prod-200' => 'Chair']
```

If duplicate keys exist, the last matching element will be inserted into the plucked collection:

```php
$collection = collect([
    ['brand' => 'Tesla',  'color' => 'red'],
    ['brand' => 'Pagani', 'color' => 'white'],
    ['brand' => 'Tesla',  'color' => 'black'],
    ['brand' => 'Pagani', 'color' => 'orange'],
]);

$plucked = $collection->pluck('color', 'brand');

$plucked->all();

// ['Tesla' => 'black', 'Pagani' => 'orange']
```

`pop()`

The `pop` method removes and returns the last item from the collection:

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

$collection->pop();

// 5

$collection->all();

// [1, 2, 3, 4]
```

`prepend()`

The `prepend` method adds an item to the beginning of the collection:

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

$collection->prepend(0);
```

```
$collection->all();

// [0, 1, 2, 3, 4, 5]
```

You may also pass a second argument to set the key of the prepended item:

```
$collection = collect(['one' => 1, 'two' => 2]);

$collection->prepend(0, 'zero');

$collection->all();

// ['zero' => 0, 'one' => 1, 'two' => 2]
```

`pull()`

The `pull` method removes and returns an item from the collection by its key:

```
$collection = collect(['product_id' => 'prod-100', 'name' => 'Desk']);

$collection->pull('name');

// 'Desk'

$collection->all();

// ['product_id' => 'prod-100']
```

`push()`

The `push` method appends an item to the end of the collection:

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

$collection->push(5);

$collection->all();

// [1, 2, 3, 4, 5]
```

`put()`

The `put` method sets the given key and value in the collection:

```
$collection = collect(['product_id' => 1, 'name' => 'Desk']);

$collection->put('price', 100);

$collection->all();

// ['product_id' => 1, 'name' => 'Desk', 'price' => 100]
```

`random()`

The `random` method returns a random item from the collection:

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

$collection->random();

// 4 - (retrieved randomly)
```

You may optionally pass an integer to `random` to specify how many items you would like to randomly retrieve. A collection of items is always returned when explicitly passing the number of items you wish to receive:

```
$random = $collection->random(3);

$random->all();

// [2, 4, 5] - (retrieved randomly)
```

If the Collection has fewer items than requested, the method will throw an `InvalidArgumentException`.

reduce()

The `reduce` method reduces the collection to a single value, passing the result of each iteration into the subsequent iteration:

```
$collection = collect([1, 2, 3]);

$total = $collection->reduce(function ($carry, $item) {
    return $carry + $item;
});

// 6
```

The value for `$carry` on the first iteration is `null`; however, you may specify its initial value by passing a second argument to `reduce`:

```
$collection->reduce(function ($carry, $item) {
    return $carry + $item;
}, 4);

// 10
```

reject()

The `reject` method filters the collection using the given callback. The callback should return `true` if the item should be removed from the resulting collection:

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

$filtered = $collection->reject(function ($value, $key) {
    return $value > 2;
});

$filtered->all();

// [1, 2]
```

For the inverse of the `reject` method, see the `filter` method.

replace()

The `replace` method behaves similarly to `merge`; however, in addition to overwriting matching items with string keys, the `replace` method will also overwrite items in the collection that have matching numeric keys:

```
$collection = collect(['Taylor', 'Abigail', 'James']);

$replaced = $collection->replace([1 => 'Victoria', 3 => 'Finn']);

$replaced->all();

// ['Taylor', 'Victoria', 'James', 'Finn']
```

replaceRecursive()

This method works like `replace`, but it will recur into arrays and apply the same replacement process to the inner values:

```
$collection = collect(['Taylor', 'Abigail', ['James', 'Victoria', 'Finn']]);

$replaced = $collection->replaceRecursive(['Charlie', 2 => [1 => 'King']]);

$replaced->all();

// ['Charlie', 'Abigail', ['James', 'King', 'Finn']]
```

reverse()

The `reverse` method reverses the order of the collection's items, preserving the original keys:

```
$collection = collect(['a', 'b', 'c', 'd', 'e']);

$reversed = $collection->reverse();

$reversed->all();

/*
```

```
    [
        4 => 'e',
        3 => 'd',
        2 => 'c',
        1 => 'b',
        0 => 'a',
    ]
*/
```

`search()`

The `search` method searches the collection for the given value and returns its key if found. If the item is not found, `false` is returned.

```
$collection = collect([2, 4, 6, 8]);

$collection->search(4);

// 1
```

The search is done using a "loose" comparison, meaning a string with an integer value will be considered equal to an integer of the same value. To use "strict" comparison, pass `true` as the second argument to the method:

```
$collection->search('4', true);

// false
```

Alternatively, you may pass in your own callback to search for the first item that passes your truth test:

```
$collection->search(function ($item, $key) {
    return $item > 5;
});

// 2
```

`shift()`

The `shift` method removes and returns the first item from the collection:

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

$collection->shift();

// 1

$collection->all();

// [2, 3, 4, 5]
```

`shuffle()`

The `shuffle` method randomly shuffles the items in the collection:

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

$shuffled = $collection->shuffle();

$shuffled->all();

// [3, 2, 5, 1, 4] - (generated randomly)
```

`skip()`

The `skip` method returns a new collection, without the first given amount of items:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);

$collection = $collection->skip(4);

$collection->all();

// [5, 6, 7, 8, 9, 10]
```

`slice()`

The `slice` method returns a slice of the collection starting at the given index:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);

$slice = $collection->slice(4);

$slice->all();

// [5, 6, 7, 8, 9, 10]
```

If you would like to limit the size of the returned slice, pass the desired size as the second argument to the method:

```
$slice = $collection->slice(4, 2);

$slice->all();

// [5, 6]
```

The returned slice will preserve keys by default. If you do not wish to preserve the original keys, you can use the `values` method to reindex them.

`some()`

Alias for the `contains` method.

`sort()`

The `sort` method sorts the collection. The sorted collection keeps the original array keys, so in this example we'll use the `values` method to reset the keys to consecutively numbered indexes:

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

$sorted = $collection->sort();

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

// [1, 2, 3, 4, 5]
```

If your sorting needs are more advanced, you may pass a callback to `sort` with your own algorithm. Refer to the PHP documentation on `uasort`, which is what the collection's `sort` method calls under the hood.

If you need to sort a collection of nested arrays or objects, see the `sortBy` and `sortByDesc` methods.

`sortBy()`

The `sortBy` method sorts the collection by the given key. The sorted collection keeps the original array keys, so in this example we'll use the `values` method to reset the keys to consecutively numbered indexes:

```
$collection = collect([
    ['name' => 'Desk', 'price' => 200],
    ['name' => 'Chair', 'price' => 100],
    ['name' => 'Bookcase', 'price' => 150],
]);

$sorted = $collection->sortBy('price');

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

/*
    [
        ['name' => 'Chair', 'price' => 100],
        ['name' => 'Bookcase', 'price' => 150],
        ['name' => 'Desk', 'price' => 200],
    ]
*/
```

You can also pass your own callback to determine how to sort the collection values:

```
$collection = collect([
```

```
    ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
    ['name' => 'Chair', 'colors' => ['Black']],
    ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown']],
]);

$sorted = $collection->sortBy(function ($product, $key) {
    return count($product['colors']);
});

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

/*
    [
        ['name' => 'Chair', 'colors' => ['Black']],
        ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
        ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown']],
    ]
*/
```

`sortByDesc()`

This method has the same signature as the `sortBy` method, but will sort the collection in the opposite order.

`sortKeys()`

The `sortKeys` method sorts the collection by the keys of the underlying associative array:

```
$collection = collect([
    'id' => 22345,
    'first' => 'John',
    'last' => 'Doe',
]);

$sorted = $collection->sortKeys();

$sorted->all();

/*
    [
        'first' => 'John',
        'id' => 22345,
        'last' => 'Doe',
    ]
*/
```

`sortKeysDesc()`

This method has the same signature as the `sortKeys` method, but will sort the collection in the opposite order.

`splice()`

The `splice` method removes and returns a slice of items starting at the specified index:

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

$chunk = $collection->splice(2);

$chunk->all();

// [3, 4, 5]

$collection->all();

// [1, 2]
```

You may pass a second argument to limit the size of the resulting chunk:

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

$chunk = $collection->splice(2, 1);

$chunk->all();

// [3]

$collection->all();

// [1, 2, 4, 5]
```

In addition, you can pass a third argument containing the new items to replace the items removed from the collection:

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

$chunk = $collection->splice(2, 1, [10, 11]);

$chunk->all();

// [3]

$collection->all();

// [1, 2, 10, 11, 4, 5]
```

`split()`

The `split` method breaks a collection into the given number of groups:

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

$groups = $collection->split(3);

$groups->toArray();

// [[1, 2], [3, 4], [5]]
```

`sum()`

The `sum` method returns the sum of all items in the collection:

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

// 15
```

If the collection contains nested arrays or objects, you should pass a key to use for determining which values to sum:

```
$collection = collect([
    ['name' => 'JavaScript: The Good Parts', 'pages' => 176],
    ['name' => 'JavaScript: The Definitive Guide', 'pages' => 1096],
]);

$collection->sum('pages');

// 1272
```

In addition, you may pass your own callback to determine which values of the collection to sum:

```
$collection = collect([
    ['name' => 'Chair', 'colors' => ['Black']],
    ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
    ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown']],
]);

$collection->sum(function ($product) {
    return count($product['colors']);
});

// 6
```

`take()`

The `take` method returns a new collection with the specified number of items:

```
$collection = collect([0, 1, 2, 3, 4, 5]);

$chunk = $collection->take(3);

$chunk->all();

// [0, 1, 2]
```

You may also pass a negative integer to take the specified amount of items from the end of the collection:

```php
$collection = collect([0, 1, 2, 3, 4, 5]);

$chunk = $collection->take(-2);

$chunk->all();

// [4, 5]
```

`tap()`

The `tap` method passes the collection to the given callback, allowing you to "tap" into the collection at a specific point and do something with the items while not affecting the collection itself:

```php
collect([2, 4, 3, 1, 5])
    ->sort()
    ->tap(function ($collection) {
        Log::debug('Values after sorting', $collection->values()->toArray());
    })
    ->shift();

// 1
```

`times()`

The static `times` method creates a new collection by invoking the callback a given amount of times:

```php
$collection = Collection::times(10, function ($number) {
    return $number * 9;
});

$collection->all();

// [9, 18, 27, 36, 45, 54, 63, 72, 81, 90]
```

This method can be useful when combined with factories to create Eloquent models:

```php
$categories = Collection::times(3, function ($number) {
    return factory(Category::class)->create(['name' => "Category No. $number"]);
});

$categories->all();

/*
    [
        ['id' => 1, 'name' => 'Category #1'],
        ['id' => 2, 'name' => 'Category #2'],
        ['id' => 3, 'name' => 'Category #3'],
    ]
*/
```

`toArray()`

The `toArray` method converts the collection into a plain PHP `array`. If the collection's values are Eloquent models, the models will also be converted to arrays:

```php
$collection = collect(['name' => 'Desk', 'price' => 200]);

$collection->toArray();

/*
    [
        ['name' => 'Desk', 'price' => 200],
    ]
*/
```

`toArray` also converts all of the collection's nested objects that are an instance of `Arrayable` to an array. If you want to get the raw underlying array, use the `all` method instead.

`toJson()`

The `toJson` method converts the collection into a JSON serialized string:

```php
$collection = collect(['name' => 'Desk', 'price' => 200]);

$collection->toJson();

// '{"name":"Desk", "price":200}'
```

`transform()`

The `transform` method iterates over the collection and calls the given callback with each item in the collection. The items in the collection will be replaced by the values returned by the callback:

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

$collection->transform(function ($item, $key) {
    return $item * 2;
});

$collection->all();

// [2, 4, 6, 8, 10]
```

Unlike most other collection methods, `transform` modifies the collection itself. If you wish to create a new collection instead, use the `map` method.

`union()`

The `union` method adds the given array to the collection. If the given array contains keys that are already in the original collection, the original collection's values will be preferred:

```php
$collection = collect([1 => ['a'], 2 => ['b']]);

$union = $collection->union([3 => ['c'], 1 => ['b']]);

$union->all();

// [1 => ['a'], 2 => ['b'], 3 => ['c']]
```

`unique()`

The `unique` method returns all of the unique items in the collection. The returned collection keeps the original array keys, so in this example we'll use the `values` method to reset the keys to consecutively numbered indexes:

```php
$collection = collect([1, 1, 2, 2, 3, 4, 2]);

$unique = $collection->unique();

$unique->values()->all();

// [1, 2, 3, 4]
```

When dealing with nested arrays or objects, you may specify the key used to determine uniqueness:

```php
$collection = collect([
    ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],
    ['name' => 'iPhone 5', 'brand' => 'Apple', 'type' => 'phone'],
    ['name' => 'Apple Watch', 'brand' => 'Apple', 'type' => 'watch'],
    ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'phone'],
    ['name' => 'Galaxy Gear', 'brand' => 'Samsung', 'type' => 'watch'],
]);

$unique = $collection->unique('brand');

$unique->values()->all();

/*
    [
        ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],
        ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'phone'],
    ]
*/
```

You may also pass your own callback to determine item uniqueness:

```php
$unique = $collection->unique(function ($item) {
    return $item['brand'].$item['type'];
});

$unique->values()->all();

/*
    [
        ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],
        ['name' => 'Apple Watch', 'brand' => 'Apple', 'type' => 'watch'],
        ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'phone'],
        ['name' => 'Galaxy Gear', 'brand' => 'Samsung', 'type' => 'watch'],
    ]
*/
```

The `unique` method uses "loose" comparisons when checking item values, meaning a string with an integer value will be considered equal to an integer of the same value. Use the `uniqueStrict` method to filter using "strict" comparisons.

`uniqueStrict()`

This method has the same signature as the `unique` method; however, all values are compared using "strict" comparisons.

`unless()`

The `unless` method will execute the given callback unless the first argument given to the method evaluates to `true`:

```php
$collection = collect([1, 2, 3]);

$collection->unless(true, function ($collection) {
    return $collection->push(4);
});

$collection->unless(false, function ($collection) {
    return $collection->push(5);
});

$collection->all();

// [1, 2, 3, 5]
```

For the inverse of `unless`, see the `when` method.

`unlessEmpty()`

Alias for the `whenNotEmpty` method.

`unlessNotEmpty()`

Alias for the `whenEmpty` method.

`unwrap()`

The static `unwrap` method returns the collection's underlying items from the given value when applicable:

```php
Collection::unwrap(collect('John Doe'));

// ['John Doe']

Collection::unwrap(['John Doe']);

// ['John Doe']

Collection::unwrap('John Doe');

// 'John Doe'
```

`values()`

The `values` method returns a new collection with the keys reset to consecutive integers:

```php
$collection = collect([
    10 => ['product' => 'Desk', 'price' => 200],
    11 => ['product' => 'Desk', 'price' => 200]
]);
```

```php
$values = $collection->values();

$values->all();

/*
    [
        0 => ['product' => 'Desk', 'price' => 200],
        1 => ['product' => 'Desk', 'price' => 200],
    ]
*/
```

`when()`

The `when` method will execute the given callback when the first argument given to the method evaluates to `true`:

```php
$collection = collect([1, 2, 3]);

$collection->when(true, function ($collection) {
    return $collection->push(4);
});

$collection->when(false, function ($collection) {
    return $collection->push(5);
});

$collection->all();

// [1, 2, 3, 4]
```

For the inverse of `when`, see the `unless` method.

`whenEmpty()`

The `whenEmpty` method will execute the given callback when the collection is empty:

```php
$collection = collect(['michael', 'tom']);

$collection->whenEmpty(function ($collection) {
    return $collection->push('adam');
});

$collection->all();

// ['michael', 'tom']

$collection = collect();

$collection->whenEmpty(function ($collection) {
    return $collection->push('adam');
});

$collection->all();

// ['adam']

$collection = collect(['michael', 'tom']);

$collection->whenEmpty(function($collection) {
    return $collection->push('adam');
}, function($collection) {
    return $collection->push('taylor');
});

$collection->all();

// ['michael', 'tom', 'taylor']
```

For the inverse of `whenEmpty`, see the `whenNotEmpty` method.

`whenNotEmpty()`

The `whenNotEmpty` method will execute the given callback when the collection is not empty:

```php
$collection = collect(['michael', 'tom']);

$collection->whenNotEmpty(function ($collection) {
    return $collection->push('adam');
});

$collection->all();
```

```
// ['michael', 'tom', 'adam']

$collection = collect();

$collection->whenNotEmpty(function ($collection) {
    return $collection->push('adam');
});

$collection->all();

// []

$collection = collect();

$collection->whenNotEmpty(function($collection) {
    return $collection->push('adam');
}, function($collection) {
    return $collection->push('taylor');
});

$collection->all();

// ['taylor']
```

For the inverse of `whenNotEmpty`, see the `whenEmpty` method.

`where()`

The `where` method filters the collection by a given key / value pair:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->where('price', 100);

$filtered->all();

/*
    [
        ['product' => 'Chair', 'price' => 100],
        ['product' => 'Door', 'price' => 100],
    ]
*/
```

The `where` method uses "loose" comparisons when checking item values, meaning a string with an integer value will be considered equal to an integer of the same value. Use the `whereStrict` method to filter using "strict" comparisons.

Optionally, you may pass a comparison operator as the second parameter.

```
$collection = collect([
    ['name' => 'Jim', 'deleted_at' => '2019-01-01 00:00:00'],
    ['name' => 'Sally', 'deleted_at' => '2019-01-02 00:00:00'],
    ['name' => 'Sue', 'deleted_at' => null],
]);

$filtered = $collection->where('deleted_at', '!=', null);

$filtered->all();

/*
    [
        ['name' => 'Jim', 'deleted_at' => '2019-01-01 00:00:00'],
        ['name' => 'Sally', 'deleted_at' => '2019-01-02 00:00:00'],
    ]
*/
```

`whereStrict()`

This method has the same signature as the `where` method; however, all values are compared using "strict" comparisons.

`whereBetween()`

The `whereBetween` method filters the collection within a given range:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 80],
```

```
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Pencil', 'price' => 30],
    ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->whereBetween('price', [100, 200]);

$filtered->all();

/*
    [
        ['product' => 'Desk', 'price' => 200],
        ['product' => 'Bookcase', 'price' => 150],
        ['product' => 'Door', 'price' => 100],
    ]
*/
```

`whereIn()`

The `whereIn` method filters the collection by a given key / value contained within the given array:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->whereIn('price', [150, 200]);

$filtered->all();

/*
    [
        ['product' => 'Bookcase', 'price' => 150],
        ['product' => 'Desk', 'price' => 200],
    ]
*/
```

The `whereIn` method uses "loose" comparisons when checking item values, meaning a string with an integer value will be considered equal to an integer of the same value. Use the `whereInStrict` method to filter using "strict" comparisons.

`whereInStrict()`

This method has the same signature as the `whereIn` method; however, all values are compared using "strict" comparisons.

`whereInstanceOf()`

The `whereInstanceOf` method filters the collection by a given class type:

```
$collection = collect([
    new User,
    new User,
    new Post,
]);

return $collection->whereInstanceOf(User::class);
```

`whereNotBetween()`

The `whereNotBetween` method filters the collection within a given range:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 80],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Pencil', 'price' => 30],
    ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->whereNotBetween('price', [100, 200]);

$filtered->all();

/*
    [
        ['product' => 'Chair', 'price' => 80],
        ['product' => 'Pencil', 'price' => 30],
    ]
*/
```

whereNotIn()

The `whereNotIn` method filters the collection by a given key / value not contained within the given array:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->whereNotIn('price', [150, 200]);

$filtered->all();

/*
    [
        ['product' => 'Chair', 'price' => 100],
        ['product' => 'Door', 'price' => 100],
    ]
*/
```

The `whereNotIn` method uses "loose" comparisons when checking item values, meaning a string with an integer value will be considered equal to an integer of the same value. Use the `whereNotInStrict` method to filter using "strict" comparisons.

whereNotInStrict()

This method has the same signature as the `whereNotIn` method; however, all values are compared using "strict" comparisons.

wrap()

The static `wrap` method wraps the given value in a collection when applicable:

```
$collection = Collection::wrap('John Doe');

$collection->all();

// ['John Doe']

$collection = Collection::wrap(['John Doe']);

$collection->all();

// ['John Doe']

$collection = Collection::wrap(collect('John Doe'));

$collection->all();

// ['John Doe']
```

zip()

The `zip` method merges together the values of the given array with the values of the original collection at the corresponding index:

```
$collection = collect(['Chair', 'Desk']);

$zipped = $collection->zip([100, 200]);

$zipped->all();

// [['Chair', 100], ['Desk', 200]]
```

Higher Order Messages

Collections also provide support for "higher order messages", which are short-cuts for performing common actions on collections. The collection methods that provide higher order messages are: `average`, `avg`, `contains`, `each`, `every`, `filter`, `first`, `flatMap`, `groupBy`, `keyBy`, `map`, `max`, `min`, `partition`, `reject`, `some`, `sortBy`, `sortByDesc`, `sum`, and `unique`.

Each higher order message can be accessed as a dynamic property on a collection instance. For instance, let's use the `each` higher order message to call a method on each object within a collection:

```
$users = User::where('votes', '>', 500)->get();

$users->each->markAsVip();
```

Likewise, we can use the `sum` higher order message to gather the total number of "votes" for a collection of users:

```
$users = User::where('group', 'Development')->get();

return $users->sum->votes;
```

Lazy Collections

Introduction

> Before learning more about Laravel's lazy collections, take some time to familiarize yourself with PHP generators.

To supplement the already powerful `Collection` class, the `LazyCollection` class leverages PHP's generators to allow you to work with very large datasets while keeping memory usage low.

For example, imagine your application needs to process a multi-gigabyte log file while taking advantage of Laravel's collection methods to parse the logs. Instead of reading the entire file into memory at once, lazy collections may be used to keep only a small part of the file in memory at a given time:

```
use App\LogEntry;
use Illuminate\Support\LazyCollection;

LazyCollection::make(function () {
    $handle = fopen('log.txt', 'r');

    while (($line = fgets($handle)) !== false) {
        yield $line;
    }
})->chunk(4)->map(function ($lines) {
    return LogEntry::fromLines($lines);
})->each(function (LogEntry $logEntry) {
    // Process the log entry...
});
```

Or, imagine you need to iterate through 10,000 Eloquent models. When using traditional Laravel collections, all 10,000 Eloquent models must be loaded into memory at the same time:

```
$users = App\User::all()->filter(function ($user) {
    return $user->id > 500;
});
```

However, the query builder's `cursor` method returns a `LazyCollection` instance. This allows you to still only run a single query against the database but also only keep one Eloquent model loaded in memory at a time. In this example, the `filter` callback is not executed until we actually iterate over each user individually, allowing for a drastic reduction in memory usage:

```
$users = App\User::cursor()->filter(function ($user) {
    return $user->id > 500;
});

foreach ($users as $user) {
    echo $user->id;
}
```

Creating Lazy Collections

To create a lazy collection instance, you should pass a PHP generator function to the collection's `make` method:

```
use Illuminate\Support\LazyCollection;

LazyCollection::make(function () {
    $handle = fopen('log.txt', 'r');

    while (($line = fgets($handle)) !== false) {
```

```
        yield $line;
    }
});
```

The Enumerable Contract

Almost all methods available on the `Collection` class are also available on the `LazyCollection` class. Both of these classes implement the `Illuminate\Support\Enumerable` contract, which defines the following methods:

all
average
avg
chunk
collapse
collect
combine
concat
contains
containsStrict
count
countBy
crossJoin
dd
diff
diffAssoc
diffKeys
dump
duplicates
duplicatesStrict
each
eachSpread
every
except
filter
first
firstWhere
flatMap
flatten
flip
forPage
get
groupBy
has
implode
intersect
intersectByKeys
isEmpty
isNotEmpty
join
keyBy
keys
last
macro
make
map
mapInto
mapSpread
mapToGroups
mapWithKeys
max
median
merge
mergeRecursive
min
mode
nth
only
pad
partition
pipe
pluck
random
reduce
reject
replace
replaceRecursive
reverse
search
shuffle
skip
slice
some
sort
sortBy
sortByDesc
sortKeys
sortKeysDesc
split
sum
take
tap
times
toArray
toJson
union
unique
uniqueStrict
unless
unlessEmpty
unlessNotEmpty
unwrap
values
when
whenEmpty
whenNotEmpty
where
whereStrict
whereBetween
whereIn
whereInStrict
whereInstanceOf
whereNotBetween
whereNotIn
whereNotInStrict
wrap
zip

Methods that mutate the collection (such as `shift`, `pop`, `prepend` etc.) are *not* available on the `LazyCollection` class.

Lazy Collection Methods

In addition to the methods defined in the `Enumerable` contract, the `LazyCollection` class contains the following methods:

`tapEach()`

While the `each` method calls the given callback for each item in the collection right away, the `tapEach` method only calls the given callback as the items are being pulled out of the list one by one:

```
$lazyCollection = LazyCollection::times(INF)->tapEach(function ($value) {
    dump($value);
});

// Nothing has been dumped so far...

$array = $lazyCollection->take(3)->all();

// 1
// 2
// 3
```

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

- Release Notes
- Getting Started
- Routing
- Blade Templates
- Authentication
- Authorization
- Artisan Console
- Database

Resources

- Laracasts
- Laravel News
- Laracon
- Laracon EU
- Laracon AU
- Jobs
- Certification
- Forums

Partners

- Vehikl
- Tighten Co.
- Kirschbaum
- Byte 5
- 64Robots
- Cubet
- DevSquad
- Ideil

Ecosystem

- Vapor
- Forge
- Envoyer
- Horizon
- Lumen
- Nova
- Echo
- Valet

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.

Eloquent ORM
Testing

Cyber-Duck
ABOUT YOU
Become A Partner

Mix
Spark
Cashier
Homestead
Dusk
Passport
Scout
Socialite