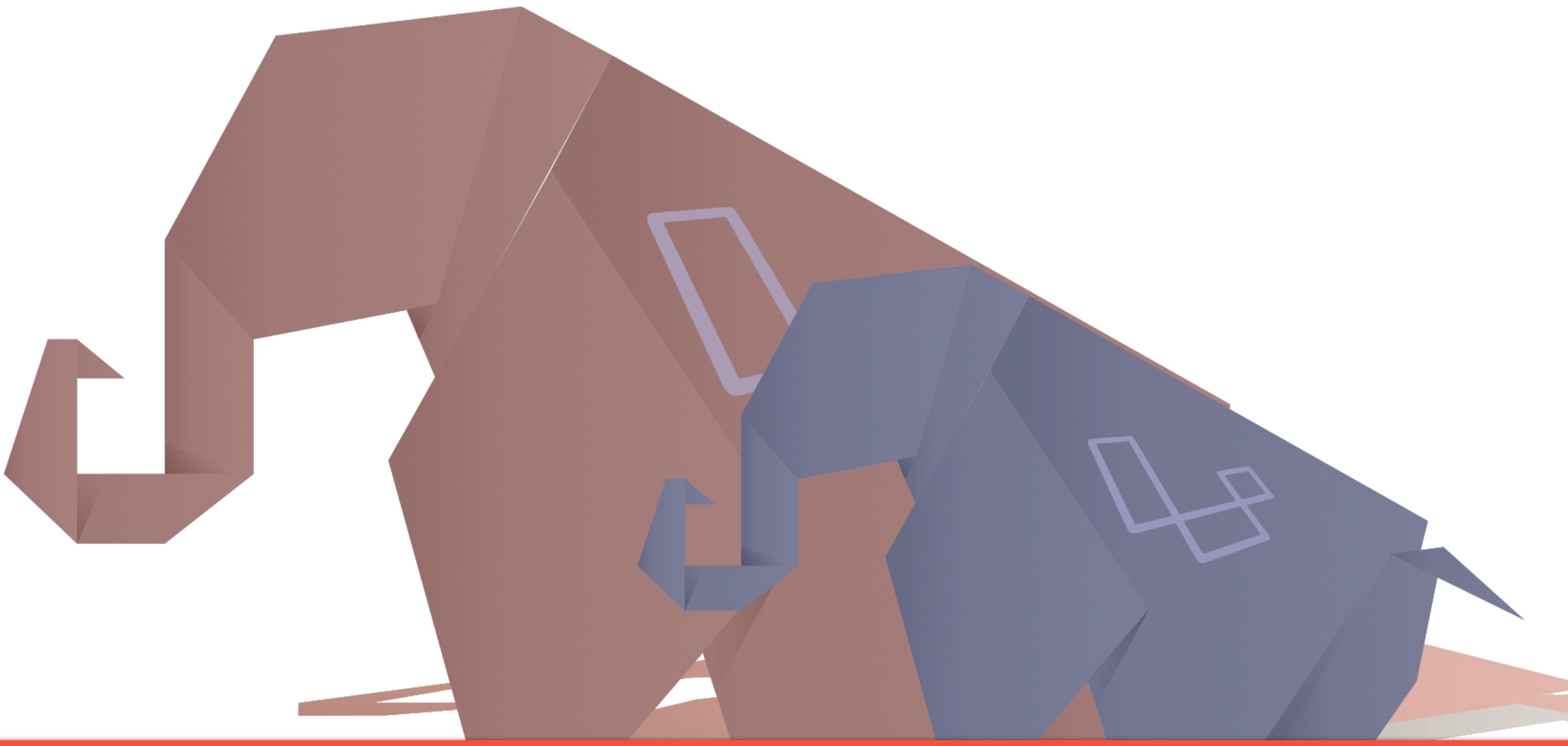
LARAVEL COMPANION

A GUIDE TO HELPERS, COLLECTIONS AND MORE



JOHNATHON KOSTER

Laravel Companion

A Guide to Helpers, Collections And More

Johnathon Koster

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

This version was published on 2016-08-25

Contents

Who is This Book For?

This book is for anyone who uses the Laravel framework and would like to learn more about the frameworks helper functions, Collection class, as well as many other helper features provided by the framework.

Symbols Used In This Book

This icon will be used when defining a term. This icon is not used often, and usually only applied when there may be confusion about or term, or when sufficient ambiguity already exists around a term.

This icon is used to present the thoughts or opinions of the author(s) and acts as an invitation for conversation. These sections also feature a title styling that is more subtle.

Look for this icon in the text to learn about any appendices that may additional reference material for a given section.

This icon indicates information that is important or the cause of general errors.

This icon is used to point out or explain the cause of specific errors or problems that may arise during development.

This icon is used to point out additional information that might be useful in the context of a given chapter, but is not required.

This icon is used to provide additional contextually-relevant information to help clarify ideas or to point out multiple ways of doing something.

What Was Used to Write This Book?

The book publication and rendering was done using Leanpub (https://leanpub.com/[1]). The book was written using a custom variant of LeanPub's markdown that is more suited for writing large programming books (with specific tooling for writing about Laravel). The actual writing took place in Adobe Brackets (http://brackets.io/[2]) using custom markdown and spell checking plugins (the spell checking plugin is available for free on GitHub[3]).

[1]https://leanpub.com/

[2]http://brackets.io/

[3]https://github.com/JohnathonKoster/brackets-spellcheck

Here be dragons!

```
                                                 .~))>>
                                                .~)>>
                                              .~))))>>>
                                            .~))>>             ___
                                          .~))>>)))>>      .-~))>>
                                        .~)))))>>       .-~))>>)>
                                      .~)))>>))))>>  .-~)>>)>
                  )                 .~))>>))))>>  .-~)))))>>)>
               ( )@@*)             //)>))))))  .-~))))>>)>
             ).@(@@               //))>>))) .-~))>>)))))>>)>
           (( @.@).              //))))) .-~)>>)))))>>)>
         ))  )@@*.@@ )          //)>))) //))))))>>))))>>)>
      ((  ((@@@.@@             |/))))) //)))))>>)))>>)>
     )) @@*. )@@ )   (\_(\-\b  |))>)) //)))>>)))))))>>)>
   (( @@@(.@(@ .    _/`-`  ~|b |>))) //)>>)))))))>>)>
    )* @@@ )@*     (@)  (@) /\b|))) //))))))>>))))>>
  (( @. )@( @ .   _/  /    /  \b)) //))>>)))))>>>_._
   )@@ (@@*)@@.  (6///6)- / ^  \b)//))))))>>)))>>   ~~-.
( @jgs@@. @@@.*@_ VvvvvV//  ^  \b/)>>))))>>      _.     `bb
 ((@@ @@@*.(@@ . - | o |' \ (  ^   \b)))>>        .'       b`,
  ((@@).*@@ )@ )   \^^^/  ((   ^  ~)_        \  /           b `,
    (@@. (@@ ).     `-'   (((   ^    `\ \ \ \ \|             b  `.
      (*.@*              / ((((        \| | |  \       .       b `.
                        / / (((((  \    \ /  _.-~\     Y,      b  ;
                       / / / (((((( \    \.-~   _.`" _.-~`,    b  ;
                      /   /   `(((((()    )    (((((~      `,  b  ;
                    _/  _/       `"""/  /'                  ; b   ;
                _.-~_.-~           /  /'                _.'~bb _.'
              ((((~~              / /'              _.'~bb.--~
                                 ((((          __.-~bb.-~
                                             .'  b .~~
                                             :bb ,'
                                             ~~~~
```

I Helper Functions

“Science makes people reach selflessly for truth and objectivity; it teaches people to accept reality, with wonder and admiration, not to mention the deep awe and joy that the natural order of things brings to the true scientist.”

– Lise Meitner (1878 - 1968), Physicist

Laravel has a number of helper functions that will make developing applications even easier. Most of them are utility based, such as determining if an array has an item, or if a string starts with a particular string or character. All functions will be organized by category and then grouped by their behavior. Functions that do not have a logical grouping will then appear in the order they appear in when viewing the helper files.

1. String Helper Functions

String operations in PHP often seem complicated to developers coming to the language from other language, specifically languages that have a robust object-oriented approach to the provided data types. Operations such as checking if a particular string contains a substring, or if a string starts or ends with another string are not impossible, nor are they difficult, they just appear over-complicated or obscure. Luckily for us, Laravel has an amazing collection of string helper functions that we can use to perform some of the most common actions. The string helper functions are kept in the static class `Illuminate\Support\Str`. The following functions are not necessarily in the same order that they will appear in the class. They have been grouped based on their functionality and purpose.

1.1 Immutable Strings

Laravel's string helper functions treat string values as immutable. This means that any given string passed as a function argument will be unchanged, and a new, modified copy of the string will be returned from the function (unless stated otherwise).

```php
use Illuminate\Support\Str;

$firstString = 'my words';

// Returns `MyWords`
Str::studly($firstString);

// Displays 'my words'
echo $firstString;
```

1.2 `ascii($value)`

The `ascii` helper method accepts only one argument: `$value`. `$value` should always be a string, or something that can be cast into a string. The function converts a string in the UTF-8[1] encoding into it's ASCII[2] equivalent.

This function is useful when communicating with other software platforms that require ASCII, or when complying with protocols (such as HTTP) about how headers, or some values, need to be formatted.

Example use:

[1]http://en.wikipedia.org/wiki/UTF-8
[2]http://en.wikipedia.org/wiki/ASCII

```php
use Illuminate\Support\Str;

// The following will output "a test string"
// in the ASCII encoding.

echo Str::ascii('a test string');
```

As of Laravel 5, the `ascii` function internally uses the stringy[3] library. Previous versions of Laravel have used the patchwork/utf8[4] package.

1.3 `studly($value)`

Studly caps is a way of formatting text with capital letters, according to some pattern. Laravel's pattern is to remove the following word separators and capitalize the first letter of each word it finds (while not affecting the case of any other letters):

- Any number of spaces ' '
- The underscore _
- The hyphen -

Let's take a look at a few examples to see how this would format a few example strings. The string returned will appear above the function call as a comment. In fact, all of the function calls below will return the string `MyWords`:

```php
use Illuminate\Support\Str;

// MyWords
echo Str::studly('my words');

// MyWords
echo Str::studly('my    words');

// MyWords
echo Str::studly(' my-words');

// MyWords
echo Str::studly(' my-words_');
```

[3]https://github.com/danielstjules/Stringy
[4]https://github.com/tchwork/utf8

Pascal Case

Laravel's `studly` method can also be used to generate Pascal Cased style strings. Both styles are the same as camel case with the first letter capitalized.

Because the `studly` method does not affect the case of any letters after the first letter, we can arrive at output that some users might not expect. Again, the output string will appear above the method call in a comment.

```php
use Illuminate\Support\Str;

// MyWORDS
echo Str::studly('my-WORDS');

// MYWORDS
echo Str::studly('MY_WORDS');
```

1.3.1 `studly_case($value)`

The `studly_case` function is a shortcut to calling `Str::studly`. This function is declared in the global namespace.

1.4 `camel($value)`

Camel casing is similar to studly case such that each word starts with a capitalized letter, with the difference being the first character is lowercased. Like the `studly` method, the `camel` method will not affect the casing of the rest of the word.

The following examples will all return the string `myWords`:

```php
use Illuminate\Support\Str;

// myWords
echo Str::camel('my words');

// myWords
echo Str::camel('my-words');

// myWords
echo Str::camel('my_words');

// myWords
echo Str::camel('   my-words_');
```

Again, the `camel` function does not affect the casing of any characters after the first one.

```php
use Illuminate\Support\Str;

// mYWORDS
echo Str::camel('MY WORDS');

// mywords
echo Str::camel('mywords');
```

1.4.0.1 `camel_case($value)`

The `camel_case` function is a shortcut to calling `Str::camel`. This function is declared in the global namespace.

1.5 `snake($value, $delimiter = '_')`

The `snake` helper method replaces all uppercase letters within the string with the lowercased variant prefixed with the given `$delimiter`. The only exception to this rule is that if the first character of the string is capitalized, it will be replaced by the lowercased variant without the `$delimiter` being prefixed. Because of this behavior, the entire string will be lowercased. The `snake` method trims extraneous whitespace from any string being converted:

Here are a few examples with the result above the function call in comments:

```php
use Illuminate\Support\Str;

// my_words
echo Str::snake('MyWords');

// my-words
echo Str::snake('MyWords', '-');

// m_y_w_o_r_d_s
echo Str::snake('MYWORDS');

// this is _pretty_cool
echo Str::snake('this_is_PrettyCool');
```

1.5.1 `snake_case($value, $delimiter = '_')`

The `snake_case` function is a shortcut to calling `Str::snake`. This function is declared in the global namespace.

1.6 replaceFirst($search, $replace, $subject)

The replaceFirst helper method is used to replace the first occurrence of a given $search string with the provided $replace string in the $subject string. All three parameters are required and must be strings.

The following examples highlight the basic usage of the replaceFirst method. The results of the method call will appear above the call as a comment.

```php
use Illuminate\Support\Str;

// //maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css
Str::replaceFirst(
    'http://', '//',
    'http://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css'
);

// Hello, there!
Str::replaceFirst('there', 'Hello', 'there, there!');
```

1.6.1 str_replace_first($search, $replace, $subject)

The str_replace_first function is a shortcut to calling Str::replaceFirst. This function is declared in the global namespace.

1.7 trans($id = null, $parameters = [], $domain = 'messages', $locale = null)

The trans helper function is used to return the translation for the given key. It defines a $key parameter which corresponds to an array key within the group file. It also accepts an array of replacements (which are passed through the $parameters array parameter) and a $locale parameter, which can be used to specify the desired locale.

Replacements are passed as a key/value pair. The replacement keys will be matched to any placeholders within the translation. Placeholders begin with a single colon (:) followed by a name and a space. For example, in the validation.php language file, the following lines can be found:

```php
<?php

return [

    ...

    'accepted'             => 'The :attribute must be accepted.',
    'array'                => 'The :attribute must be an array.',

    ...

];
```

In the above translation lines, `:attribute` is a placeholder than can be replaced using the `$parameters` parameter. The following code examples will demonstrate the results of using the `$parameters` parameter. The results will appear above the method call as a comment.

```php
// The :attribute must be an array.
trans('validation.array');

// The provided value must be an array.
trans('validation.array', ['attribute' => 'provided value']);
```

It should be noted that the placeholder name in the `$parameters` parameter should not contain the leading `:` character.

1.8 `trans_choice($id, $number, array $parameters = [], $domain = 'messages', $locale = null)`

The `trans_choice` helper function is used to pluralize a given `$id`, translating it for the given `$locale`. The `choice` method accepts the `$number` of some collection of objects that it should use when making pluralization decisions. Like the `trans` helper function, it also accepts and array of replacements, using the `$parameters` array.

Assuming the following group file is located at `/resource/lang/en/plural.php`:

```php
<?php

return [
    'books' => 'There is one book.|There are :count many books.',
];
```

It should be noted that pluralized strings allow for multiple messages to be stored in one translation line. Messages are separated by the `|` character, and are ordered such that the message that corresponds to the lower `$number` appear first and messages that appear for any higher `$number` appear sequentially afterwards.

The following code example would select the first message:

```
// There is one book.
trans_choice('plural.books', 1);
```

The following code example would select the second message:

```
// There are 4 books.
trans_choice('plural.books', 4, ['choice' => 4]);
```

The following code examples highlight some more ways the `trans_choice` helper function can be called, and ways they will mostly appear within applications:

```
// There are 4 books.
trans_choice('plural.books', 4, ['choice' => 4]);

// There is one book.
trans_choice('plural.books', 1, ['choice' => 1]);
```

1.9 `e($value)`

The `e` function is a simple wrapper of PHP's `htmlentities` function. The `e` function utilizes the `UTF-8` character encoding. The `e` function will sanitize user input when displaying it to the browser.

Let's assume that a malicious user was posting on a forum and set the subject of their post to this:

```
<script>alert("hello everyone");</script>
```

If the forum software did not sanitize user output, perfectly valid JavaScript code would be sent to the browser. Since browsers are overly happy to execute any JavaScript included in the document, any forum visitor would see an alert box with `hello everyone` every time a page was loaded that displayed that particular forum post.

To prevent this, use the `e` function to sanitize user input when sending it to the browser:

```
$unsafeClientCode = '<script>alert("hello everyone");</script>';

$safeClientCode   = e($unsafeClientCode);
```

The value of `$safeClientCode` would then be:

```
&lt;script&gt;alert(&quot;hello everyone&quot;);&lt;/script&gt;gt;
```

At this point the browser will render a string that literally represents what they had typed.

2. Array Helper Functions

Laravel provides many helper functions for interacting and manipulating array data structures. Laravel's array functions offer additional functionality on top of PHP's built in array functions. The helper functions in this section are located within the `Illuminate\Support\Arr` static class. There are functions defined in the static class that have counterparts defined in the global namespace. These functions appear below each static function.

2.1 Immutable Arrays

All of the array helper functions below, unless stated otherwise, treat arrays as an immutable data structure. This means that any changes made to an array are returned as a copy of the original array, with the original array remaining unchanged.

Here is a quick example:

```php
use Illuminate\Support\Arr;

$originalArray = [
    'fruit'     => 'orange',
    'vegetable' => 'carrot'
];

$anotherArray = array_add($originalArray, 'vehicle', 'car');
```

Executing `var_dump` on `$originalArray` would produce the following output:

```
array(2) {
    ["fruit"] "orange"
    ["vegetable"] "carrot"
}
```

This is because Laravel's functions do not affect the original array (unless stated otherwise).

Running `var_dump` on `$anotherArray`, however, would produce the expected results:

```
array(3) {
    ["fruit"] "orange"
    ["vegetable"] "carrot"
    ["vehicle"] "car"
}
```

2.2 "Dot" Notation

In addition to treating arrays as immutable, Laravel's functions support "dot" notation when accessing items in an array. This is similar to JavaScript's dot notation when accessing object properties.

JavaScript:

```
forms = document.forms;
```

PHP, using Laravel's helper functions:

```
use Illuminate\Support\Arr;

// Create a multi-dimensional array, to simulate a document and forms.
$testDocument = [
    'document' => [
        'forms' => [

        ]
    ]
];

// Get the forms.
$forms = Arr::get($testDocument, 'document.forms');

// Traditionally, with vanilla PHP:
$forms = $testDocument['document']['forms'];
```

Using dot notation with Laravel's functions will be explored in more detail in the following sections.

2.3 `add($array, $key, $value)`

The `add` helper method adds the given `$key` and `$value` to an `$array` if the `$key` doesn't already exist within the given `$array`.

Consider the following code snippet:

```php
use Illuminate\Support\Arr;

$myArray = [
    'animal' => 'Araripe Manakin',
    'plant' => 'Pineland Wild Petunia'
];

$myArray  = Arr::add($myArray, 'insect', 'Extatosoma Tiaratum');
```

The the end result would be:

```
array(3) {
    ["animal"] "Araripe Manakin"
    ["plant"] "Pineland Wild Petunia"
    ["insect"] "Extatosoma Tiaratum"
}
```

2.3.1 `array_add($array, $key, $value)`

The `array_add` function is a shortcut to calling `Arr::add`. This function is declared in the global namespace.

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

The `get` helper method will retrieve an item from the given `$array` using dot notation. This allows developers to retrieve items from the array at arbitrary depths quickly, without having to use PHP's array syntax.

Assuming the following array:

```php
$anArray = [
    'nested_array_one' => [
        'nested_array_two' => [
            'key' => 'value'
        ]
    ]
];
```

We can quickly retrieve the value for `key` like so:

```php
use Illuminate\Support\Arr;

$value = Arr::get($anArray, 'nested_array_one.nested_array_two.key');
```

Where the alternative syntax would be:

```php
$value = $anArray['nested_array_one']['nested_array_two']['key'];
```

While PHP's array access syntax may be a little shorter, using the dot notation is easier to read. We can also specify a `$default` value, which will be returned if there is no matching `$key` found in the array.

```php
use Illuminate\Support\Arr;

$value = Arr::get(
            $anArray,
            'nested_array_one.nested_array_two.does_not_exist',
            'default value'
);
```

`$value` would then have the value of `default value`. Using this method is *shorter* than using PHP's array access syntax and the `isset` function, where we would have to check at each level if the key exists or not.

2.4.1 `array_get($array, $key, $default = null)`

The `array_get` function is a shortcut to calling `Arr::get`. This function is declared in the global namespace.

2.5 `pull(&$array, $key, $default = null)`

The `pull` helper method is similar to the `get` method in that it will return a value for the given `$key` in the `$array` (or the `$default`, if it is specified and the `$key` does not exists). The difference is that the `pull` method will remove the item it finds from the array.

Mutable Function

This function affects the original `$array`.

Consider the following array, which is probably familiar to most people these days:

```
$weekDays = [
    1 => 'Sunday',
    2 => 'Monday',
    3 => 'Tuesday',
    4 => 'Wednesday',
    5 => 'Thursday',
    6 => 'Friday',
    7 => 'Saturday'
];
```

A lot of people do not like Mondays. Let's demote `Monday` to the `$theWorstDayOfTheWeek` while also removing it from our `$weekDays`:

```
use Illuminate\Support\Arr;

// Demote 'Monday'
$worstDayOfTheWeek = Arr::pull($weekDays, 2);
```

The `$weekDays` array would now look like this:

```
array(6) {
    [1] "Sunday"
    [3] "Tuesday"
    [4] "Wednesday"
    [5] "Thursday"
    [6] "Friday"
    [7] "Saturday"
}
```

2.5.1 `array_pull(&$array, $key, $default = null)`

The `array_pull` function is a shortcut to calling `Arr::pull`. This function is declared in the global namespace.

3. Application and User Flow Helper Functions

3.1 `app($make = null, $parameters = [])`

The app function provides access to the `Illuminate\Container\Container` instance. Because `Container` is a singleton, any call to `app()` will return the *same* `Container` instance.

The `app` function can also be used to resolve registered dependencies from the `Container` instance. For example, to return an instance of the `Illuminate\Auth\AuthManager` class (the class behind the `Auth` facade), invoke `app` with the `auth` argument:

```php
// $authManager will be an instance of
// Illuminate\Auth\AuthManager
$authManager = app('auth');
```

3.2 `auth($guard = null)`

Used without supplying any arguments for `$guard`, the `auth` helper function is an alternative to using the `Auth` facade. With no arguments, the `auth` function returns an instance of `Illuminate\Contracts\Auth\Factory` (which by default is an instance of `Illuminate\Auth\AuthManager`).

The following example shows to equivalent ways of retrieving the currently logged in user:

```php
use Illuminate\Support\Facades\Auth;

// Get the current user using the `Auth` facade.
$currentUser = Auth::user();

// Get the current user using the `auth` helper.
$currentUser = auth()->user();
```

You can quickly access any custom authentication guards by supplying an argument for the `$guard` parameter. For example if a custom guard was added with the name `customGuard`:

```php
if (auth('customGuard')->attempt([
        'email'    => $email,
        'password' => $password
    ])) {
    // Authentication passed.
}
```

When `$guard` is set to `null`, the `AuthManager` instance will internally set `$guard` equal to the value of the `auth.defaults.guard` configuration key (by default this is `web`). The following code samples would produce the same results:

```php
// Not passing anything for `$guard`.
if (auth()->attempt([
        'email'    => $email,
        'password' => $password
    ])) {

}

// Passing the default configuration value for `$guard`.
if (auth('web')->attempt([
        'email'    => $email,
        'password' => $password
    ])) {

}
```

3.3 `policy($class)`

The `policy` helper function can be used to retrieve a policy (a policy class can be any valid PHP class) instance for a given `$class`. The `$class` can be either a string or an object instance. If no policies for the given `$class` have been registered with the application an instance of `InvalidArgumentException` will be thrown.

3.4 `event($event, $payload = [], $halt = false)`

The `event` helper is a convenient way to dispatch a particular `$event` to its listeners. An optional `$payload` of data can also be supplied when dispatching the event. When the event is dispatched, all it's listeners will be called and any responses from the listener will be added to an array. The array of responses is generally the return value of the `event` function.

If the `$halt` parameter is set to `true`, the dispatcher will stop calling listeners after any of the listeners return a value that is not `null`. If no listeners have returned a response and all listeners were called, the `event` function will return `null`.

The following examples will highlight the various ways to use the `event` function:

```php
/**
 * The following code is simply creating a new event class
 * to signal that a video was watched.
 */

namespace App\Events;

use App\Video;
use App\Events\Event;
use Illuminate\Queue\SerializesModels;

class VideoWasWatched extends Event
{
    use SerializesModels;

    public $video;

    /**
     * Create a new event instance.
     *
     * @param  Video  $video
     * @return void
     */
    public function __construct(Video $video)
    {
        $this->video = $video;
    }
}
```

An event can be fired by the name of a class like so:

```php
event('\App\Events\VideoWasWatched', $video);
```

Alternatively, an event can be fired by instantiating a new instance of the event class itself:

```php
event(new VideoWasWatched($video));
```

In the above example, a `$payload`was not supplied. This is because the event instance *itself* will become the payload. It is equivalent to the following:

```php
$event = new VideoWasWatched($video);
event('App\Events\VideoWasWatched', $event);
```

3.5 dispatch($command)

The dispatch helper function can be used to push a new job onto the job queue (or dispatch the job). The function resolves the configured Illuminate\Contracts\Bus\Dispatcher implementation from the Service Container and then calls the dispatch method in the Dispatcher instance.

Assuming a variable $job exists that contains a valid queueable Job the following examples are all equivalent:

```
use Illuminate\Contracts\Bus\Dispatcher;
use Illuminate\Support\Facades\Bus;

dispatch($job);
app(Dispatcher::class)->dispatch($job);
Bus::dispatch($job);
```

3.6 factory()

The factory function is used to create Eloquent models, generally used when testing an application. The factory method can generally take one of four forms:

```
// Return an instance of the 'User' model.
$user = factory('App\User')->make();

// Return two instances of the 'User' model.
$users = factory('App\User', 2)->make();

// Return one instance of the 'User' model, with some modifications
// that were defined earlier and bound to the 'admin' description.
$adminUser = factory('App\User', 'admin')->make();

// Return two instances of the 'User' model, with some modifications
// that were defined earlier and bound to the 'admin' description.
$adminUsers = factory('App\User', 'admin', 2)->make();
```

3.7 method_field($method)

The method_field helper function is a simple function that returns a new instance of Illuminate\Support\HtmlString. The contents of the HtmlString instance is a hidden HTML input field with the name _method and the a value of $method.

The following example highlights the results of the method_field function:

```
method_field('PUT');
method_field('POST');
method_field('GET');
method_field('PATCH');
method_field('DELETE');
```

The return value of the `method_field` function is an instance of `HtmlString`. The corresponding HTML string value for each function call above would be:

```
<input type="hidden" name="_method" value="PUT">
<input type="hidden" name="_method" value="POST">
<input type="hidden" name="_method" value="GET">
<input type="hidden" name="_method" value="PATCH">
<input type="hidden" name="_method" value="DELETE">
```

3.8 validator(array $data = [], array $rules = [], array $messages = [], array $customAttributes = [])

The `validator` helper function is a versatile helper function. If no arguments are supplied to the `validator` function (all arguments are optional), the function will return a `Illuminate\Contracts\Validation\Factory` implementation instance (the default implementation will be an instance of `Illuminate\Validation\Factory`).

Using the the `validator` function without arguments and gaining access to the instance methods is essentially the same as using the `Illuminate\Support\Facades\Validator` facade. The following examples are equivalent; all the examples will make use of the `request` helper function.

```
use Illuminate\Support\Facades\Validator;

// Using the Validator facade.
$validator = Validator::make(request()->all(), [
    'title' => 'required|unique:posts|max:255',
    'body'  => 'required'
]);

// Using the validator helper function.
$anotherValidator = validator()->make(request()->all(), [
    'title' => 'required|unique:posts|max:255',
    'body'  => 'required'
]);
```

The second example above can be shortened even further. You can supply the arguments for the `make` instance method directly to the `validator` helper function. When you supply arguments to the `validator` helper function it will call and return the value of the `make` instance method:

```php
$validator = validator(request()->all, [
    'title' => 'required|unique:posts|max:255',
    'body'  => 'required'
]);
```

4. Configuration, Environment, Security and Logging Helper Functions

4.1 `bcrypt($value, $options = [])`

The `bcrypt` function will return a hashed representation of the given `$value`. The `bcrypt` function also accepts an array of `$options` which can be used to affect how the hash is computed. Each invocation of the `bcrypt` function should produce a different result, even if the input remains the same.

The following example demonstrates how to call the `bcrypt` function:

```php
for ($i = 0; $i < 10; $i++)
{
    // echo the hash and an HTML line break
    echo bcrypt('test'),'<br>';
}
```

The above example would output something similar to this:

```
$2y$10$6b8WZt.Ugwnjjb3JZQH51ecaG.VSjOOO2xCZ3t4s/MGGHU112hhD2
$2y$10$o/uJXcnrNDQraGgk1.VG9.LwssnANCyOEO8tCuiL5RlO33CpGo.Lq
$2y$10$7qWDkO43obCCN4hpNDt2Hut2xbg8xmKQHzZF/m4EdsGUHApXcKLyi
$2y$10$e4srCMoCOaIl9qd2wuk.8e2pBGTxaAu/bDi2CrNlRcNyXxvtYePIy
$2y$10$1MhsM.KaYpwoODuoBi7wmO6jUrMJ0xGaigL6/JMKAgb48CgyFz8tK
$2y$10$wTdq3XAG7/UKT0aO4u9lO.ZRcDiaF5p4fXMViticodID9oC/CTsJO
$2y$10$yHwchZ9HCKZjfnqqulQ7eu61noEwIVZBXKwSZ8.rvYyk9p0SXFNKG
$2y$10$5XvPyJE9EQ6DpOdYzM.NYeR4eDjAzntn2ogytDh1tNU4ebWrHaYvS
$2y$10$V1yb7D7rqqUL7BZkR2c3HOjYHvVB/lRg5cvrL/Hl/KYzrKrTV/tvC
$2y$10$6DAP/IjDTOH3iezOzx/CyuH37ZEDtc6.ADkDEfJUUn/msgUGe5A4S
```

4.1.1 `bcrypt` Options

The following table lists all of the options that are available to use in with the `bcrypt` function.

`bcrypt` Options

Option	Description
`rounds`	Determines how many iterations the `bcrypt` function will internally use.

4.1.1.1 `rounds`

The `rounds` option is used to control how many iterations the underlying Blowfish implementation will use when generating the final hash. The `rounds` value can be any positive integer between `4` and `31` (including both `4` and `31`). If an integer outside the stated range is supplied an `ErrorException` will be thrown. The `rounds` option is synonymous with the `cost` option when using PHP's `password_hash` or `crypt` functions. Increasing the rounds will also increase the time required to compute the hash. The default value for this option is `10`.

```
// Compute a hash with a 'cost' of 4
bcrypt('test', ['rounds' => 4]);

// Compute a hash with a 'cost' of 5
bcrypt('test', ['rounds' => 4]);

// These will throw an `ErrorException`
bcrypt('test', ['rounds' => 3]);
bcrypt('test', ['founds' => 32]);
```

The number of iterations that will be used can be determined with the following equation:

$$iterations = 2^{rounds}$$

Calling `bcrypt` with the `rounds` option set to `10` will use 2^10 or `1024` iterations.

4.2 `encrypt($value)`

The `encrypt` helper function can be used to encrypt a given `$value`. The function resolves the configured `Illuminate\Contracts\Encryption\Encrypter` implementation from the Service Container and then calls the `encrypt` method on the `Encrypter` instance.

The following examples are all equivalent:

```
use Illuminate\Support\Facades\Crypt;

// Encrypt using the Crypt facade:
Crypt::encrypt('Hello, Universe');

// Encrypt using the encrypt helper:
encrypt('Hello, Universe');
```

4.3 `info($message, $context = [])`

The `info` helper will write an information entry into Laravel's log files. It allows a `$message` to be set, as well as an optional `$context` (which is an array of data). Whole the `$context` must be an array, the actual elements of the array do not have to be arrays themselves.

The following example shows the usage of the `info` helper. An example of what the function calls would produce in the log files follows the code examples.

```
// An example message context.
$context = ['name' => 'John Doe', 'email' => 'you@homestead'];

// A log message without context.
info('This is a log message without context');

// A log message where the context is an array.
info('This is a log message', $context);

// A log message where the context is an array of objects.
info('This is another log message', [(object) $context]);
```

The above code would produce results similar to the following (some lines have been indented to prevent wrapping and improve readability):

```
...
[2015-06-15 02:14:43] local.INFO: This is a log message without context
[2015-06-15 02:14:43] local.INFO: This is a log message
  {"name":"John Doe","email":"you@homestead"}
[2015-06-15 02:14:43] local.INFO: This is another log message
  [
   "[object] (stdClass: {\"name\":\"John Doe\",\"email\":\"you@homestead\"})"
  ]
...
```

5. URL Generation Helper Functions

5.1 action($name, $parameters = [], $absolute = true)

The action helper function can be used to generate a URL to a controller action, which is supplied as an argument to the $name parameter. If the controller action requires or accepts parameters these can be supplied by utilizing the $parameters parameter.

The following examples will use the following example controller (stored in app/Http/Controllers/ExampleController.php):

```php
<?php

namespace App\Http\Controllers;

class ExampleController extends Controller {

    public function sayHello($name) {
        return "Hello, {$name}!";
    }

}
```

as well as the following route definition (added to the web middleware group in app/Http/routes.php):

```php
Route::get('sayHello/{name}', 'ExampleController@sayHello');
```

Calling the action helper function like so:

```php
action('ExampleController@sayHello', ['name' => 'Jim']);
```

would return the following URL:

```
http://laravel.artisan/sayHello/Jim
```

The exact URL that is generated will depend on the current domain of the site.

Notice that when the action function was called, we did not need to include the root namespace of the controller (App\Http\Controllers). Laravel will automatically add the namespace for you. Any namespaces that are not part of the root namespace must be included (such as Auth), however.

5.1.1 Relative URLs

If you do want to generate the full URL to a controller action, simply supply a truth value of `false` for the `$absolute` parameter. The following example demonstrates this:

```php
action('ExampleController@sayHello', ['name' => 'Jim'], false)
```

The resulting URL of the function call would then be:

```
/sayHello/Jim
```

6. Miscellaneous Helper Functions

6.1 `dd()`

The dd function is a debug utility that will *dump* a set of values and then die. This means it will create a human-readable representation of the values and then stop execution of the script.

The dd function will take into account if the application is currently running in the console, or if it is returning a response to a client (such as a browser), and then update it's output accordingly.

6.2 `data_get($target, $key, $default = null)`

The `data_get` function is identical in behavior to both the `array_get` and `object_get` function. The difference is that the `data_get` function will accept both objects *and* arrays as it's `$target`.

6.3 `windows_os`

The `windows_os` helper function can be used to determine if the host server is running a Microsoft Windows® operating system. The function returns either `true`—if the server is running Windows®—or `false`—if the server is not running Windows®.

Using this function you can write code like so:

```php
<?php

if (windows_os()) {
    // Windows⊠ specific commands or instructions.
} else {
    // Anything that is not Windows⊠.
}
```

instead of code like this:

```php
<?php

if (strtolower(substr(PHP_OS, 0, 3)) === 'win') {
    // Windows&reg; specific commands or instructions.
} else {
    // Anything that is not Windows&reg;.
}
```

6.4 `tap($value, $callback)`

The `tap` helper function is used to call a given Closure (provided as an argument to the `$callback` parameter) with the given `$value`. The `tap` helper function will return the reference to the original `$value` as its return value. This allows you to call any number of methods with any type of return values within the `$callback` while maintaining a reference to the original object. This behavior is particularly useful during method chaining. Another way to think of this functions behavior is that it allows you to specify and fix the return value of a series of method calls, regardless of what methods are called within the internal block.

> The `tap` helper function was added in Laravel version 5.3 and defined in the `src/Illuminate/Support/helpers.php` file. This function can easily be integrated into older Laravel code bases as it does not depend on any new fundamental framework components.

The following example PHP classes will be used to demonstrate a trivial usage of the `tap` helper function. The classes represent a crude "talker" system where one can add messages and retrieve the final message back:

Stored in the `app/Message.php` file:

```php
<?php

namespace App;

class Message
{

    /**
     * The actual message.
     *
     * @var string
     */
    protected $message = '';

    /**
     * Gets the original message.
     *
     * @return string
     */
    public function getOriginal()
    {
        return $this->message;
    }

    /**
     * Changes the message.
     *
     * @param  $message
     * @return Message
     */
    public function change($message)
    {
        $this->message = $message;

        return $this;
    }

    public function __toString()
    {
        return strtolower($this->message);
    }

}
```

The `Message` class if fairly simple. It is simply a value object that wraps PHP's native string data type. It provides a few methods that can be used to retrieve the original value and change (mutate) the internal value of the message. When the message is cast back to a string it is converted to its lowercased variant. The important thing to notice is that the `change` method returns a reference back to itself.

Stored in the `app/Talker.php` file:

```
<?php

namespace App;

class Talker
{

    /**
    * The actual parts of the message.
    *
```

```
     * @var array
     */
    protected $messageParts = [];

    /**
     * Adds a message part to the end of the message.
     *
     * @param  $message
     * @return Message
     */
    public function atEnd(Message $message)
    {
        array_push($this->messageParts, $message);
        return $message;
    }

    /**
     * Adds a message part to the start of the message.
     *
     * @param  $message
     * @return Message
     */
    public function atBeginning(Message $message)
    {
        array_unshift($this->messageParts, $message);
        return $message;
    }

    /**
     * Returns the string representation of the message.
     *
     * @return string
     */
    public function talk()
    {
        return ucfirst(implode(' ', $this->messageParts));
    }

}
```

The `Talker` class is also very simple. It maintains an array of `Message` instances and allows you to add them to either the beginning or ending of the array of messages. In addition, it allows you to retrieve a string made up of all the individual messages with the first character upper cased. Annoyingly the `atEnd` and `atBeginning` methods return the `Message` instance instead of the `Talker` instance, which makes chaining rather difficult.

The following example will create a simple helper function to remove some of the steps of using the `Talker` and `Message` APIs:

```php
<?php

use App\Talker;
use App\Message;

/**
 * Say something.
 *
 * string  $message
 * @return string
 */
function saySomething($message) {
    $talker = new Talker;
    $talker->atBeginning(new Message)->change($message);
    return $talker->talk();
}
```

Using the new helper function might look like this:

```php
<?php

echo saySomething('Hello World');
```

The user would see the following text in the browser:

```
Hello world
```

It would be nice if we could return the value of the method chain from our function like so:

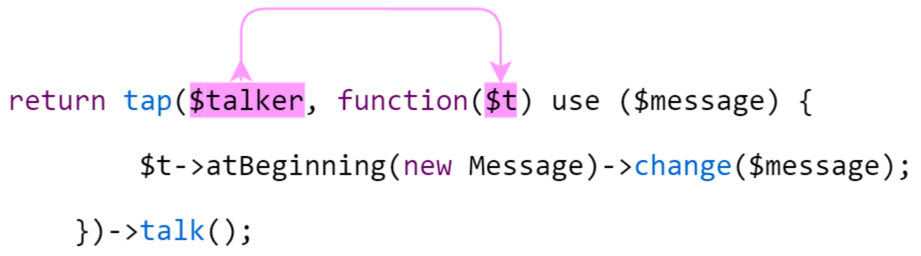
```php
<?php

// ...

function saySomething($message) {
    $talker = new Talker;
    return $talker->atBeginning(new Message)->change($message)->talk();
}

// ...
```

However, that code would not work. This is because the `atBeginning` method on the `Talker` class returns an instance of `Message`. The `talk` method does not exist on the `Message` class, and if it did would probably not have the same behavior as the `Talker` instance method. We can use the `tap` function to get around this quite easily:

```php
<?php

// ...

function saySomething($message) {
    $talker = new Talker;
    return tap($talker, function($t) use ($message) {
        $t->atBeginning(new Message)->change($message);
    })->talk();
}

// ...
```

Tap Function Closure Arguments

In the previous example the Closure defines a parameter named `$t`. When the `tap` function executes, the value of `$talker` will be used as the inner value of `$t` and they will refer to the same `Talker` instance. The following diagram visually demonstrates the flow of data in the `tap` helper function.



```php
return tap($talker, function($t) use ($message) {

        $t->atBeginning(new Message)->change($message);

    })->talk();
```

In the above example we are using the `tap` function to keep our method chain going. In addition we are passing the `$message` parameter into the inner callback function so that we can use it. The above example could actually be simplified further by using the `with` helper function to return a new instance of `Talker`:

```php
<?php

// ...

function saySomething($message) {
    return tap(with(new Talker), function($t) use ($message) {
        $t->atBeginning(new Message)->change($message);
    })->talk();
}
```

```

// ...
```

The previous example is very contrived but does demonstrate a fairly basic use of the `tap` helper function. More common use cases might be when passing around Eloquent models with extensive method chaining.

The following example will make use of the Eloquent model factory builder to create a new instance, modify an attribute, save the model and then return the created model instance:

```php
<?php

use App\User;

$user = tap(factory(User::class)->make(), function($user) {
    $user->setAttribute('name', 'No such a random name')
         ->save();
});
```

After the above example has executed, the attributes of the `$user` model instance might look similar to the following output:

```
array [
    "name"           => "No such a random name"
    "email"          => "Batz.Bridget@example.com"
    "password"       => "$2y$10$aXsXCEx3pkgmNcWpJFkAE.AMHoyk3o8XC6Kth3..."
    "remember_token" => "nayR2BKwaQ"
    "updated_at"     => "2016-06-03 22:29:01"
    "created_at"     => "2016-06-03 22:29:01"
    "id"             => 7
  ]
```

The actual attribute values will differ as they are randomly assigned a value from the model factory. Because of the way we used the `tap` helper function we were able to modify the `name` attribute elegantly, save the model and get the exact return value we wanted.

II Collections

"I was told I'd never make it to VP rank because I was too outspoken. Maybe so, but I think men will always find an excuse for keeping women in their 'place.' So, let's make that place the executive suite and start more of our own companies."

– Jean Bartik (1924 - 2011), Computer Programmer

The `Illuminate\Support\Collection` class provides a wrapper around PHP's arrays. The `Collection` class has numerous methods for retrieving, setting and manipulating the data of the underlying array. Each `Collection` instance has an internal array which is used to store the actual collection items.

7. Basic Usage - Collections as Arrays

Because Laravel's `Collection` class is an extravagant wrapper around PHP arrays, it's basic usage should be very familiar to most PHP developers. The `Collection` class also implements PHP's `ArrayAccess`[1] interface so that developers can use PHP's array syntax when working with collections:

```
// Creating a new collection instance.
$collection = new Collection;

// Appending to the collection.
$collection[] = 'First';
$collection[] = 'Second';
$collection[] = 'Third';

// Adding key/value pairs to the collection.
$collection['key']  = 'Some value';
$collection['key2'] = 'Some other value';

foreach ($collection as $key => $value) {
    // Iterate through the array with access to key/value pairs.
}

// Retrieve the value associated with a known key.
$someValue = $collection['key'];
```

After the above code executes, the variable `$someValue` would hold the value `Some value`. It should be noted that even though array syntax works with collections, PHP's array specific functions will not work on an instance of a collection directly. Instead we have to retrieve the underlying array from the collection.

The following code will throw an instance of `ErrorException` stating that `array_values` expects the first parameter to be an object:

[1] http://php.net/manual/en/class.arrayaccess.php

```php
use Illuminate\Support\Collection;

$collection = new Collection;
$collection[] = 'First';
$collection[] = 'Second';
$collection[] = 'Third';

$values = array_values($collection);
```

Instead, the `toArray` method can be used to retrieve a representation of the underlying array:

```php
use Illuminate\Support\Collection;

$collection = new Collection;
$collection[] = 'First';
$collection[] = 'Second';
$collection[] = 'Third';

$values = array_values($collection->toArray());
```

After the above code executes, `$values` would contain a value similar to the following:

```
array (size=3)
  0 => string 'First'  (length=5)
  1 => string 'Second' (length=6)
  2 => string 'Third'  (length=5)
```

7.1 Notes on Basic Usage

Any caveats that apply when working with arrays in PHP also apply to `Collection` instances when treating them as arrays. This means that developers have to check for the existence of items themselves when accessing or iterating over the collection. For example, the following code will produce an `ErrorException` stating that the index `does_not_exist` does not exist:

```php
use Illuminate\Support\Collection;

$collection = new Collection;

$value = $collection['does_not_exist'];
```

Interacting with collections in this manner should be relatively uncommon in practice. The `Collection` class provides many methods that aid developers in such situations. As an example, the `get` method allows retrieval of data from the collection, with the option of returning a default value:

```php
use Illuminate\Support\Collection;

$collection = new Collection;

$value = $collection->get('does_not_exist', 'But I do');
```

After the above code is executed the `$value` variable would have the value `But I do`, and the code will not throw an exception. The chapter Collections: The Public API contains more information about the public methods available.

7.2 Creating a New Collection Instance

There are a few different ways to create a new `Collection` instance with existing data. The first way to create a new instance is to pass an array of items to the `Collection` class's constructor:

```php
use Illuminate\Support\Collection;

// An array of test data.
$testData = [
    'first'  => 'This is the first',
    'second' => 'This is the second',
    'third'  => 'This is third'
];

// Create a new collection instance.
$collection = new Collection($testData);
```

Another way to create a `Collection` instance is to call the static `make` method on the `Collection` class itself:

```php
use Illuminate\Support\Collection;

// An array of test data.
$testData = [
    'first'  => 'This is the first',
    'second' => 'This is the second',
    'third'  => 'This is third'
];

// Create a new collection instance static 'make' method.
$collection = Collection::make($testData);
```

A convenient way to create a new `Collection` instance is to use the `collect` helper function. The `collect` helper function internally returns a new instance of `Collection` passing in any arguments to the class's constructor. It's most basic usage is:

```php
// An array of test data.
$testData = [
    'first'  => 'This is the first',
    'second' => 'This is the second',
    'third'  => 'This is third'
];

// Create a new collection instance using the 'collect' helper.
$collection = collect($testData);
```

8. Collections: The Public API

The `Collection` class exposes a generous public API, consisting of 64 public methods. The `Collection` API exposes methods for retrieving values from a collection, transforming the underlying array, pagination and simple aggregate functions, amongst others. This section will cover most of the methods in the public API, excluding those methods required to implement PHP's `ArrayAccess`[1] and `IteratorAggregate`[2]interfaces, as well as the `getCachingIterator` method.

8.1 `all`

The `all` method can be used to retrieve the underlying array that the collection is using to hold its data. The following code demonstrates the usage of the `all` method:

```
use Illuminate\Support\Collection;

$items = [
    'first' => 'I am first',
    'second' => 'I am second'
];

$collection = Collection::make($items);

$returnedItems = $collection->all();
```

After the above code has been executed, the `$returnedItems` variable would hold the following value:

```
array (size=2)
  'first'  => string 'I am first'  (length=10)
  'second' => string 'I am second' (length=11)
```

We can also verify that the two arrays are indeed equal:

```
// true
$areEqual = $items === $returnedItems;
```

It should also be noted that the `all` method will preserve any nested collections:

[1]http://php.net/manual/en/class.arrayaccess.php
[2]http://php.net/manual/en/class.iteratoraggregate.php

```
use Illuminate\Support\Collection;

$items = [
    'first' => 'I am first',
    'second' => 'I am second',
    'third' => new Collection([
            'first' => 'I am nested'
    ])
];

$collection = Collection::make($items);

$returnedItems = $collection->all();
```

At this point, the `$returnedItems` variable would a value similar to the following:

```
array (size=3)
  'first'  => string 'I am first'         (length=10)
  'second' => string 'I am second'        (length=11)
  'third'  =>
    object(Illuminate\Support\Collection)[132]
      protected 'items' =>
        array (size=1)
          'first' => string 'I am nested' (length=11)
```

To return an array, and have any nested collections converted to arrays, see the `toArray` method.

8.2 toArray

The `toArray` method is similar to the `all` method in that it will return the underlying array that the collection instance is using. The difference, however, is that the `toArray` method will convert any object instance it can into arrays (namely any object that implements the `Illuminate\Contracts\Support\Arrayable` interface). Consider the following code:

```
use Illuminate\Support\Collection;

$items = [
    'first'  => 'I am first',
    'second' => 'I am second',
    'third'  => new Collection([
            'first' => 'I am nested'
    ])
];

```

```
$collection = Collection::make($items);

$returnedItems = $collection->toArray();
```

After the above code has executed, the `$returnedItems` variable would contain a value similar to the following output:

```
array (size = 3)
  'first'  => string 'I am first'     (length=10)
  'second' => string 'I am second'    (length=11)
  'third'  =>
    array (size=1)
      'first' => string 'I am nested' (length=11)
```

8.3 `chunk($size, $preserveKeys = false)`

The `chunk` method is useful when working with large collections in that it allows developers to create smaller collections to work with. The `chunk` method defines two parameters: `$size`, which is used to control how large each newly created collection will be; and `$preserveKeys`, which indicates if the `chunk` method will preserve array keys when creating the new collections.

Assuming the following code:

```
$testArray = [
    'one'   => '1',
    'two'   => '2',
    'three' => '3',
    'four'  => '4',
    'five'  => '5',
    'six'   => '6'
];

$collection = new Collection($testArray);

$chunks = $collection->chunk(3);
```

The `$chunks` variable will contain a value similar to the following:

```
object(Illuminate\Support\Collection)[135]
  protected 'items' =>
    array (size=2)
      0 =>
        object(Illuminate\Support\Collection)[133]
          protected 'items' =>
            array (size=3)
              0 => string '1' (length=1)
              1 => string '2' (length=1)
              2 => string '3' (length=1)
      1 =>
        object(Illuminate\Support\Collection)[134]
          protected 'items' =>
            array (size=3)
              0 => string '4' (length=1)
              1 => string '5' (length=1)
              2 => string '6' (length=1)
```

As you can see, the `chunk` method returned a new collection that contains two new collections, each containing three elements. It should also be noted that the original array keys are missing (there are no numerical word names). This can be changed by passing `true` as the second argument:

```
$chunks = $collection->chunk(3, true);
```

Now the `$chunks` variable would have a value similar to the following:

```
object(Illuminate\Support\Collection)[135]
  protected 'items' =>
    array (size=2)
      0 =>
        object(Illuminate\Support\Collection)[133]
          protected 'items' =>
            array (size=3)
              'one'   => string '1' (length=1)
              'two'   => string '2' (length=1)
              'three' => string '3' (length=1)
      1 =>
        object(Illuminate\Support\Collection)[134]
          protected 'items' =>
            array (size=3)
              'four' => string '4' (length=1)
              'five' => string '5' (length=1)
              'six'  => string '6' (length=1)
```

The resulting collections now have the original array keys.

8.4 only($keys)

The `only` method is the logical opposite of the `except` method. The `only` method is used to return all the key/value pairs in the collection where the keys in the collection are *in* the supplied `$keys` array. Internally, this method makes a call to the `Illuminate\Support\Arr::only($array, $keys)` helper function.

Using the same example from the `except` method section, we can get only the `first_name` and `last_name` of the users in a collection:

```
// Create a new collection.
$collection = collect([
    'first_name' => 'John',
    'last_name'  => 'Doe',
    'password'   => 'some_password'
]);

// Retrieve only the first and last
// names from the collection.
$data = $collection->only(['first_name', 'last_name']);
```

After the above code has executed, the `$data` variable would contain a value similar to the following output:

```
Collection {
  #items: array:2 [
    "first_name" => "John"
    "last_name"  => "Doe"
  ]
}
```

8.5 keyBy($keyBy)

The `keyBy` method is used to create a new `Collection` instance where the keys of the new key/value pairs are determined by a `$keyBy` callback function (a `string` value can also be supplied instead of a function). The signature of the callback function is `keyBy($item)`, where the `$item` is the actual item in the collection. This method does not change the original `Collection` instance will will return a new, modified, `Collection` instance.

The following sample will be used to demonstrate the `keyBy` method:

```php
// Create a new collection.
$people = collect([
    ['id' => 12, 'name' => 'Alice', 'age' => 26],
    ['id' => 52, 'name' => 'Bob',   'age' => 34],
    ['id' => 14, 'name' => 'Chris', 'age' => 26]
]);
```

In the following example, we will create a new collection instance where the key of each item is the persons `id`. We will not supply a callback function in this example, but rather pass the name of the value we want to use as the key as as string:

```php
$results = $people->keyBy('id');
```

After the above code has executed, the `$results` variable will contain a value similar to the following output:

```
Collection {
  #items: array:3 [
    12 => array:3 [
      "id"   => 12
      "name" => "Alice"
      "age"  => 26
    ]
    52 => array:3 [
      "id"   => 52
      "name" => "Bob"
      "age"  => 34
    ]
    14 => array:3 [
      "id"   => 14
      "name" => "Chris"
      "age"  => 26
    ]
  ]
}
```

We can also achieve the exact same results by using a callback function like so:

```php
$results = $people->keyBy(function($item) {
    return $item['id'];
});
```

However, the above example is overly complicated if all we were interested in was just the `id` of the person. We could do something a little more interesting, such as returning a simple hash of each persons `id` to use as the new key:

```
$results = $people->keyBy(function($item) {
    return md5($item['id']);
});
```

After the above code has executed, the new `$results` variable would contain a value similar to the following output:

```
Collection {
  #items: array:3 [
    "c20ad4d76fe97759aa27a0c99bff6710" => array:3 [
      "id"   => 12
      "name" => "Alice"
      "age"  => 26
    ]
    "9a1158154dfa42caddbd0694a4e9bdc8" => array:3 [
      "id"   => 52
      "name" => "Bob"
      "age"  => 34
    ]
    "aab3238922bcc25a6f606eb525ffdc56" => array:3 [
      "id"   => 14
      "name" => "Chris"
      "age"  => 26
    ]
  ]
}
```

The same method could just as easily be applied to other hashing libraries, such as the hashids.php[3] library.

[3] https://github.com/ivanakimov/hashids.php

9. Message Bags

The `Illuminate\Suport\MessageBag` class is an elaborate key/value storage system for storing different types of messages. It it also allows developers to specify a format which is used when returning the messages. The format makes it incredibly simple to format messages on HTML forms, emails, etc. The `MessageBag` class implements both the `Illuminate\Contracts\Support\MessageProvider` and `Illuminate\Contracts\Support\MessageBag` interfaces.

9.1 `Illuminate\Contracts\Support\MessageProvider` Interface

The `MessageProvider` interface defines only one method, `getMessageBag`. The expected behavior of the `getMessageBag` method is to return an instance of an implementation of the `Illuminate\Contracts\Support\MessageBag` interface. Any class that implements the `MessageProvider` interface can be expected to be capable of returning `MessageBag` implementations.

The following classes extend, implement, or make use of the `MessageProvider` interface by default:

Class	Description
`Illuminate\Contracts\Validation\ValidationException`	The `ValidationException` is often thrown when a validation error occurs by some process within an applications request life-cycle.
`Illuminate\Contracts\Validation\Validation`	The `Validator` interface defines the methods and behaviors of the frameworks validator component. The `Validator` component actually extends the `MessageProvider` interface.
`Illuminate\Http\RedirectResponse`	The `RedirectReponse` class is responsible for setting the correct headers required to redirect a user's browser to another URL.
IlluminateSupportMessageBag	The `MessageBag` class is the default implementation of the `Illuminate\Contracts\Support\MessageBag` interface. It also implements the `MessageProvider` interface to return a reference to itself.
`Illuminate\View\View`	The `View` class uses instances of `MessageProvider` implementations to facilitate the communication of errors between the application logic and the front-end code of an application.

9.2 View Error Bags

The `Illuminate\Support\ViewErrorBag` class is generally used to communicate error messages with views and responses. The `ViewErrorBag` itself is essentially a container for `Illuminate\Contracts\Support\MessageBag` implementation instances.

An instance of `ViewErrorBag` is shared with views if the current request contains any errors. This functionality is facilitated by the `Illuminate\View\Middleware\ShareErrorsFromSession` middleware. The `ViewErrorBag` instance that is shared with views is given the name `errors`. Interacting with the `ViewErrorBag` instance should look familiar to most developers:

```
@if($errors->count())
<div class="alert alert-danger">
    <p>There were some errors that need to be fixed!</p>
    <ul>
        {!! implode($errors->all('<li>:message</li>')) !!}
    </ul>
</div>
@endif
```

The following examples will assume that a form exists that asks a user for their `username` and a `secret`. The form will be validated from a controller using the following code sample (in this case there is no response when validation was successful):

```
$this->validate($request, [
    'username' => 'required|max:200',
    'secret'   => 'required'
]);
```

If a user supplied only the `username`, the above Blade template code would generate the following HTML:

```
<div class="alert alert-danger">
    <p>There were some errors that need to be fixed!</p>
    <ul>
        <li>The secret field is required.</li>
    </ul>
</div>
```

If the user did not supply either a `username` or a `secret`, the following HTML code would have been generated:

```
<div class="alert alert-danger">
    <p>There were some errors that need to be fixed!</p>
    <ul>
        <li>The username field is required.</li>
        <li>The secret field is required.</li>
    </ul>
</div>
```

9.2.1 `ViewErrorBag` Public API

The `ViewErrorBag` exposes few methods itself in its public API (at the time of writing, only five methods exist intended for public use). However, the `ViewErrorBag` will redirect calls to methods that do not exist explicitly on the `ViewErrorBag` instance to the `default` `MessageBag` instance. Any public method that can be called on a `MessageBag` instance can be called on the `default` `MessageBag` instance without having to do any extra work:

```
use Illuminate\Support\ViewErrorBag;

// Create a new ViewErrorBag instance.
$viewErrorBag = new ViewErrorBag;

// Determine if the default MessageBag
// instance is empty, which in this case
// is true.
$viewErrorBag->isEmpty();
```

9.2.1.1 count

The `count` method returns the number of messages stored within the `default` `MessageBag` instance. It is also the only `MessageBag` method that specifically handled by the `ViewErrorBag` instance itself because it needs to be declared within the `ViewErrorBag` class itself to satisfy the requirements of PHP's `Countable`[1] interface.

The following code example demonstrates the behavior of the `count` method:

```
use Illuminate\Support\ViewErrorBag;
use Illuminate\Support\MessageBag;

// Create a new ViewErrorBag instance.
$viewErrorBag = new ViewErrorBag;

// Create a new MessageBag instance.
$messageBag = new MessageBag([
    'username' => [
```

[1] http://php.net/manual/en/class.countable.php

```
        'The username is required.'
    ]
]);

// Add the new MessageBag instance to
// the ViewErrorBag
$viewErrorBag->put('formErrors', $messageBag);

// Get the count from $viewErrorBag
//
// 0
$messageCount = $viewErrorBag->count();

// Change the 'default' MessageBag to the one
// created earlier.
$viewErrorBag->put('default', $messageBag);

// Get the count from $viewErrorBag
//
// 1
$messageCount = count($viewErrorBag);
```

As can be seen in the comments in the above example, the `count` method only operates on the `default` `MessageBag` instance. Because of this adding the `MessageBag` `formErrors` did not affect the count, but setting the same `MessageBag` instance as the value for the `default` key did change the results of the `count` method.

9.2.1.2 `getBag($key = null)`

The `getBag` method can be used to retrieve a `MessageBag` instance associated with the provided `$key`. If a `MessageBag` instance does not exist with the provided `$key`, a new instance of `Illuminate\Support\MessageBag` will returned instead.

The following code sample will demonstrate the usage of the `getBag` method. It also shows that because of the way PHP internally handles objects and references, that the `$messageBag` is the *same* as the value returned from the `getBag` method.

```
use Illuminate\Support\ViewErrorBag;
use Illuminate\Support\MessageBag;

// Create a new ViewErrorBag instance.
$viewErrorBag = new ViewErrorBag;

// Create a new MessageBag instance.
$messageBag = new MessageBag([
    'username' => [
        'The username is required.'
    ]
]);

$viewErrorBag->put('formErrors', $messageBag);

// Demonstrate that the object returned by the getBag
// method is the same as $messageBag.
//
// true
$areTheSame = $messageBag === $viewErrorBag->getBag('formErrors');
```

Additionally, you can request a `MessageBag` instance with any `$key`, even if they have not been set:

```
// Request a MessageBag that has not been set yet.
$messageBagInstance = $viewErrorBag->getBag('paymentErrors');
```

It should be noted that the `getBag` method does not set the `MessageBag` instance that is returning. This behavior can lead to some confusion, and can be observed in the following code sample:

```
$messageBagInstance = $viewErrorBag->getBag('paymentErrors');

// Add a message to the $messageBagInstance
$messageBagInstance->add('ccn', 'The credit card number is invalid');

// Get the number of messages.
//
// 1
$messageCount = $messageBagInstance->count();

// Get the number of messages.
//
// 0
$messageCount = $viewErrorBag->getBag('paymentErrors')->count();
```

If the `ViewErrorBag` instance had set the `MessageBag` instance before returning it from the `getBag` method, both `$messageCount` variables would have contained the value `1`.

Another way to retrieve `MessageBag` instances from the `ViewErrorBag` instance is dynamically access a property, where the property name is the intended key. This technique exhibits the same behavior as using the `getBag` method.

```php
// Get the 'formErrors' MessageBag instance.
$formErrors = $viewErrorBag->formErrors;
```

9.2.1.3 getBags

The getBags method is used to return an associative array containing all the MessageBag instances that are currently stored within the ViewErrorBag instance.

```php
use Illuminate\Support\ViewErrorBag;
use Illuminate\Support\MessageBag;

// Create a new ViewErrorBag instance.
$viewErrorBag = new ViewErrorBag;

// Create a new MessageBag instance.
$messageBag = new MessageBag([
    'username' => [
        'The username is required.'
    ]
]);

// Add some message bags to $viewErrorBag
$viewErrorBag->put('formErrors', $messageBag);
$viewErrorBag->put('paymentErrors', new MessageBag);

// Get the message bags as an array.
$messageBags = $viewErrorBag->getBags();
```

After the above code has executed, the $messageBags variable would be an array and contain a value similar to the following output:

```
array (size=2)
  'formErrors' =>
    object(Illuminate\Support\MessageBag)[142]
      protected 'messages' =>
        array (size=1)
          'username' =>
            array (size=1)
              ...
      protected 'format' => string ':message' (length=8)
  'paymentErrors' =>
    object(Illuminate\Support\MessageBag)[143]
      protected 'messages' =>
        array (size=0)
          empty
      protected 'format' => string ':message' (length=8)
```

9.2.1.4 `hasBag($key = 'default')`

The `hasBag` method is used to determine if a `MessageBag` instance exists within the `ViewErrorBag` instance with the given `$key`. The `$key` is set to `default` unless it is changed. The following example will highlight the usage of the `hasBag` method.

```php
use Illuminate\Support\ViewErrorBag;
use Illuminate\Support\MessageBag;

// Create a new ViewErrorBag instance.
$viewErrorBag = new ViewErrorBag;

// Check if the 'default' MessageBag instance
// exists.
//
// false
$doesExist = $viewErrorBag->hasBag();
$doesExist = $viewErrorBag->hasBag('default');

// Check if a 'payments' MessageBag instance
// exists.
//
// false
$doesExist = $viewErrorBag->hasBag('payments');

// Add a new 'payments' MessageBag instance.
$viewErrorBag->put('payments', new MessageBag);

// Check again if the 'payments' MessageBag instance
// exists.
//
// true
$doesExist = $viewErrorBag->hasBag('payments');
```

9.2.1.5 `put($key, Illuminate\Contracts\Support\MessageBag $bag)`

The put method is used to add a new `MessageBag` implementation instance to the `ViewErrorBag` instance, supplied as the argument to the `$bag` parameter with some name determined by the `$key` argument. The following code demonstrates how to add a new `MessageBag` instance to a `ViewErrorBag`:

```php
use Illuminate\Support\ViewErrorBag;
use Illuminate\Support\MessageBag;

// Create a new ViewErrorBag instance.
$viewErrorBag = new ViewErrorBag;

// Create a new MessageBag instance.
$messageBag = new MessageBag;

// Add the $messageBag to the $viewErrorBag
// with some key.
$viewErrorBag->put('somekey', $messageBag);

// Get the number of MessageBag instances.
$messageBagCount = count($viewErrorBag->getBags());
```

After the above code has executed, the `$messageBagCount` variable would contain the value `1`.

Another way to add `MessageBag` instances to the `ViewErrorBag` is by dynamically setting an instance property:

```php
// Add a new MessageBag instance without
// using the 'put' method.
$viewErrorBag->anotherKey = new MessageBag;
```

At this point, the `$viewErrorBag` instance would now contain two `MessageBag` instances with the keys `somekey` and `anotherKey`.

9.2.1.5.1 Resetting `MessageBag` Messages

Because neither `ViewErrorBag` or `MessageBag` provide a method to quickly remove all the messages from a `MessageBag` instance, the `put` method can be used to remove all `MessageBag` messages, or supply new ones, by changing the `MessageBag` instance for a given key:

```php
// Create a different MessageBag instance.
$newMessageBag = new MessageBag([
    'username' => [
        'The username is required.'
    ]
]);

// Get the 'somekey' MessageBag before changing it.
$beforeChange = $viewErrorBag->getBag('somekey');

// Change the MessageBag instance.
$viewErrorBag->put('somekey', $newMessageBag);
```

```

// Get the 'somekey' MessageBag after changing it.
$afterChange = $viewErrorBag->getBag('somekey');
```

After the above code has executed, the `$beforeChange` variable will contain the old `Message-Bag` instance and the `$afterChange` variable will contain the new `MessageBag` instance.

$beforeChange MessageBag Output

```
object(Illuminate\Support\MessageBag)[142]
  protected 'messages' =>
    array (size=0)
      empty
  protected 'format' => string ':message' (length=8)
```

$afterChange MessageBag Output

```
object(Illuminate\Support\MessageBag)[143]
  protected 'messages' =>
    array (size=1)
      'username' =>
        array (size=1)
          0 => string 'The username is required.' (length=25)
  protected 'format' => string ':message' (length=8)
```

10. Fluent

The `Illuminate\Support\Fluent` class is a useful data type. It allows for the construction of a data “container” similar to an array or instance of `stdClass`. However, the `Fluent` class makes it easier to make assumptions about the data the class instance contains. The following array and `stdClass` instance will be used for the next couple of examples:

```php
// Create a new array containing sample data
$testArray = [
    'first'  => 'The first value',
    'second' => 'The second value',
    'third'  => 'The third value'
];

// Create a new stdClass instance containing sample data
$testObject = new stdClass;
$testObject->first  = 'The first value';
$testObject->second = 'The second value';
$testObject->third  = 'The third value';
```

We can access data from the `$testArray` like so:

```php
// Retrieve the 'first' item
$value = $testArray['first'];
```

The `$value` variable would now contain the value `The first value`. Similarly, data can be retrieved from the `stdClass` instance using object operator (often called the “arrow”):

```php
// Retrieve the 'first' property
$value = $testObject->first;
```

Like in the previous example, the `$value` variable would now contain the value `The first value`. There is nothing surprising going on here. However, what happens when a developer needs to make assumptions about the data their code is working with? Uncertain developers often litter code with unwieldy `if` statements and similar constructs. Failure to do so generally results in fatal errors.

For example, attempting to retrieve data from an array that does not exist results in an instance of `ErrorException` being thrown:

```php
// Will raise an exception
$value = $testArray['does_not_exist'];
```

The exact error message will differ between single or multi-dimensional arrays, but the principal is the same. PHP does not like it when code attempts to access array elements that do not exist.

The same can be said for accessing an object's properties:

```php
// Will raise an exception
$value = $testObject->doesNotExist;
```

The above code example will again throw an instance of `ErrorException`, with the error message being something similar to "Undefined property: stdClass::$doesNotExit". To work around this, the following code can be written:

```php
// Get a value from an array, or a default value
// if it does not exist.

if (array_key_exists('does_not_exist', $testArray))
{
    $value = $testArray['does_not_exist'];
} else {
    $value = 'Some default value';
}


// Get a value from an object, or a default value
// if it does not exist.

if (property_exists('doesNotExist', $testObject))
{
    $objectValue = $testObject->doesNotExist;
} else {
    $objectValue = 'Some default value';
}
```

The above code example can be simplified using Laravel's array and object helper functions. Specifically see the sections on `array_get`, `object_get` and `data_get`.

In the above code example, we checked to see if an object instance has a value by using the `property_exists`[a] function instead of the `isset`[b] function. This is because the `property_-exist` function will return `true` if the property exists and has a value of `null`. The `isset` function will return `false` if the property exists but has a value of `null`.

[a]http://php.net/manual/en/function.property-exists.php

[b]http://php.net/manual/en/function.isset.php

Developers need to assume things about code quite often. In a perfect world, developers would know exactly what data an array or object their code interacts with contains. However, when dealing with remote data, such as data from external APIs, or when interfacing with code from multiple development teams, this is not always possible. The `Fluent` class can be used to simplify things for developers.

The following example will create a new `Fluent` instance with some data:

```php
// Some example data, which could be obtained from
// any number of sources.
$testArray = [
    'first'  => 'The first value',
    'second' => 'The second value',
    'third'  => 'The third value'
];

// Create a new Fluent instance.
$fluent = new Fluent($testArray);
```

Now that we have a `Fluent` instance, data can be retrieved just like an object or an array:

```php
// Accessing a value like an array.
$value = $fluent['first'];

// Accessing a value like an object.
$secondValue = $fluent->first;
```

Both `$value` and `$secondValue` would contain the value `The first value`. Accessing data that does not exist now simply returns `null`, without raising an error:

```php
$value = $fluent['does_not_exist'];

$secondValue = $fluent->doesNotExist;
```

Both `$value` and `$secondValue` would contain the value `null`. The `Fluent` class does expose public methods in its API to custom the default value returned.

10.1 `Fluent` Public API

The following sections will highlight the usage of the various public methods that the `Fluent` class provides. There are, however, some omissions in this section, namely the methods required to implement PHP's `ArrayAccess`[1] interface. The examples that follow will assume the following test array and object, unless stated otherwise:

[1]http://php.net/manual/en/class.arrayaccess.php

```php
// A test array for use with Fluent.
$testArray = [
    'first'  => 'The first value',
    'second' => 'The second value',
    'third'  => 'The third value'
];

// A test object for use with Fluent.
$testObject = new stdClass;
$testObject->first  = 'The first value';
$testObject->second = 'The second value';
$testObject->third  = 'The third value';
```

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

The `get` method will return the value associated with the provided `$key`. If the `$key` does not exist, the `$default` value will be returned (which is `null` by default).

Retrieving values from a Fluent instance:

```php
$fluent = new Fluent($testArray);

// The first value
$message = $fluent->get('first');

$fluent = new Fluent($testObject);

// The first value
$message = $fluent->get('first');

// null
$willBeNull = $fluent->get('does_not_exist');
```

The `$default` value is evaluated using the `value` helper function, meaning that it can be the result of a function:

```php
$fluent = new Fluent($testArray);

// Does not exist yet!
$message = $fluent->get('does_not_exist', function() {
    return 'Does not exist yet!';
});
```

10.1.1.1 Fluent and Closures

If we look at the following code example, one might be tempted to say that the value of `$message` would be `Hello, world!`, but that would be incorrect:

```php
$testObject = new stdClass;
$testObject->method = function() {
    return 'Hello, world!';
};

$fluent = new Fluent($testObject);

$message = $fluent->method;

// Or even this:

$message = $fluent->get('method');
```

However, that would be incorrect. It is important to remember that the `Fluent` object is an elaborate key/value storage container that can behave like an array or an object. When the fluent container is created for an object containing a closure, such as the above example, the closure instance is stored as the value. The following code will quickly prove this:

```php
// true
$isClosure = ($fluent->method instanceof Closure):
```

To evaluate the closure and get the results, the `value` helper function can be used:

```php
// Hello, world!
$message = value($fluent->method);

// Hello, world!
$message = value($fluent->get('method'));
```

10.1.2 `getAttributes()`

The `getAttributes` method simply returns an array containing all key/value pairs, representing the underlying data contained within the `Fluent` instance.

After the following code is executed:

```php
$fluent = new Fluent($testObject);

$attributes = $fluent->getAttributes();
```

The `$attributes` variable would look have a value similar to the following:

```
array (size=4)
  'first'  => string 'The first value' (length=15)
  'second' => string 'The second value' (length=16)
  'third'  => string 'The third value' (length=15)
```

10.1.3 `toArray()`

The `toArray` method returns the exact same values as the `getAttributes` method.

10.1.4 `jsonSerialize()`

The `jsonSerialize` method internally returns a call to `toArray`. Because of this, `toArray`, `jsonSerialize` and `getAttributes` are all functionally equivalent. The `jsonSerialize` method exists to satisfy PHP's `JsonSerializable`[2] interface, which allows developers to customize how a class is represented when using the `json_encode` function.

10.1.5 `toJson($options = 0)`

The `toJson` method will return a JSON encoded version of the data stored within the fluent instance. It internally does this by returning a call to PHP's `json_encode`[3] function, passing in any `$options` that were supplied. Like the `json_encode` function, the `$options` parameter is a bitmask of the predefined JSON constants[4].

The following code:

```php
$fluent = new Fluent($testObject);

$jsonValue = $fluent->toJson();
```

would be converted into the following JSON, stored in the `$jsonValue` variable:

```
{"first":"The first value","second":"The second value","third":
  "The third value","method":{}}
```

Alternatively, a well-formatted value can be returned by passing in the `JSON_PRETTY_PRINT` constant:

```php
$fluent = new Fluent($testObject);

$jsonValue = $fluent->toJson(JSON_PRETTY_PRINT);
```

This time, the `$jsonValue` would contain the following value:

[2]http://php.net/manual/en/class.jsonserializable.php
[3]http://php.net/manual/en/function.json-encode.php
[4]http://php.net/manual/en/json.constants.php

```
{
    "first": "The first value",
    "second": "The second value",
    "third": "The third value",
    "method": {}
}
```

10.1.5.1 `toJson` and Deeply Nested Data Structures

The `toJson` method internally makes a call to PHP's `json_encode` function. Unlike `json_encode`, `toJson` does not provide a way to specify the depth (essentially how many arrays are nested inside of each other) to which data will be encoded, which is by default set to `512`. To convert a fluent object into its JSON equivalent with a depth greater than `512`, the following method will be sufficient:

```
$fluent = new Fluent($testObject);

// Replace 512 with the desired depth.
$jsonValue = json_encode($fluent->jsonSerialize(), 0, 512);
```

11. Facades

Facades are a convenient way to access Laravel's components. Each facade is bound to some component already registered in the service container. Facades are typically used to provide a simpler interface to an complex underlying subsystem. While some facades provide "shortcut" methods, most of Laravel's facades act like proxies by providing a static interface to an actual class instance.

All of the default facades are located within the `Illuminate\Support\Facades` namespace and can be located within the `vendor/laravel/framework/src/Illuminate/Support/Facades` directory. Every facade extends the abstract `Facade` (`Illuminate\Support\Facades\Facade`) class and *must* provide a `getFacadeAccessor()` method which is defined to be *protected*.

The `getFacadeAccessor()` method indicates what class or component should be resolved from the service container when the facade is accessed. All method calls are then redirected to that class instance[1]. Laravel refers to this class instance as the *facade root*.

Facade roots are the class instance that is bound in the service container. For example, the `Auth` facade redirects all method calls to an instance of `Illuminate\Auth\AuthManager`, which the service container resolves with the `auth` binding.

11.1 Facade Aliases and Importing Facades

Each facade has an *alias*. An alias in PHP is just a shortcut to a longer class name. For example, when we use something *as* something else, we are creating a class alias:

```php
use Some\ReallyReallyLongClassName as ShorterName;
```

In the above example `ShorterName` is an alias of `ReallyReallyLongClassName`. Laravel creates aliases for all of the facades automatically, and the entire list of them can be found in the `aliases` array in the `config/app.php` file. The aliases that Laravel creates are in the global namespace, which means that developers can write code that looks like this:

```php
<?php namespace App\Http\Controllers;

use Input;

// Other code here.
```

instead of having to write code that references the full namespace:

[1]Some facades define additional static methods that may are not necessarily part of the underlying component or class. In these instances, the static method (that belongs to the facade) is called directly. One such example is the `Cookie` facade.

```php
<?php namespace App\Http\Controllers;

use Illuminate\Support\Facades\Input;

// Other code here.
```

Fully Qualified Names vs. Aliases

Fully qualified names, such as `Illuminate\Support\Facades\Input` are longer, but offer greater clarity on what class is being imported. At the end of the day, it comes down to personal preference.

11.2 Using Facades

Laravel's facades are used just like any other static class, with the exception that facades are redirecting method calls to an actual class instance. Facades are typically imported at the top of the source file, along with any other classes that are required.

For example, using a facade to retrieve an item from the cache:

```php
<?php namespace App\Http\Controllers;

use Illuminate\Support\Facades\Cache;

class ExampleController extends Controller {

    public function getIndex()
    {
        $cachedItem = Cache::get('some_cache_item');
    }

}
```

The alternative[2] is to *inject* the cache repository directly into the controller:

[2]Dependency injection is not the only alternative to using facades, it is, however, arguably the most common in Laravel projects.

```php
<?php namespace App\Http\Controllers;

use Illuminate\Cache\Repository as CacheRepository;

class ExampleController extends Controller {

    /**
     * The cache repository.

     * @var CacheRepository
     */
    protected $cacheRepository = null;

    public function __construct(CacheRepository $cacheRepository)
    {
        $this->cacheRepository = $cacheRepository;
    }

    public function getIndex()
    {
        $cachedItem = $this->cacheRepository->get('some_cache_item');
    }

}
```

Both code examples would accomplish the same thing: retrieve `some_cache_item` from the cache storage. The facade example allows for shorter code, and code that is definitely easier to follow along with. The second example has its own advantages too, which will not be covered in this section.

11.3 Creating Facades

It may become necessary when developing certain applications, or when creating packages to *write* a facade class. Creating a facade class is fairly simple. The first thing that is required is some actual concrete class the facade will be accessing. For this example, the following class will be used:

```php
<?php

class Calculator {

    /**
     * Adds two numbers.

     * @param  mixed $firstNumber
     * @param  mixed $secondNumber
     * @return mixed
     */
    public function add($firstNumber, $secondNumber)
    {
        return $firstNumber + $secondNumber;
    }

    // Possibly many more methods.

}
```

The next thing that needs to happen is the class needs to be registered with the service container:

```php
App::singleton('math', function()
{
    return new Calculator;
});
```

Components are typically registered with the service container through the use of Service Providers.

The `Calculator` class instance was bound as a singleton because there does not need to possibly hundreds of instances of `Calculator` created: many consumers of the class can share a single instance (note that not all classes should be bound as a singleton). The important part is that we have an service container binding: `math` which will resolve to an instance of `Calculator`.

It is at this point a facade can be created. A facade is created by creating a new class and extending Laravel's abstract `Facade` class:

```php
<?php

use Illuminate\Support\Facades\Facade;

class Math extends Facade {

    /**
     * Get the registered name of the component.
     *
     * @return string
     */
    protected static function getFacadeAccessor() { return 'math'; }

}
```

It is important to note that the value returned by the `getFacadeAccessor()` function matches the name of the service container binding: `math`. The facade will use that value to request a class instance from the service container and redirect method calls to that instance.

We will now examine how we would have used the `Calculator` class without the facade:

```php
<?php namespace App\Http\Controllers;

use Calculator;

class TestController extends Controller {

    public function getIndex()
    {
        // We will create an instance, instead of having it supplied
        // through the constructor.
        $calculator = new Calculator;

        $result = $calculator->add(1, 2);
    }

}
```

now using the facade:

```php
<?php namespace App\Http\Controllers;

use Math;

class TestController extends Controller {

    public function getindex()
    {
        $result = Math::add(1, 2);
    }

}
```

11.3.1 Creating a Facade Alias

Creating a facade alias is completely optional. Package developers often include the following steps in their package installation instructions, but it is not required for a facade to work.

In the `config/app.php` file, there is an `aliases` configuration entry. By default it will look like this:

```php
// Other configuration items.

'aliases' => [

    'App'        => 'Illuminate\Support\Facades\App',
    'Artisan'    => 'Illuminate\Support\Facades\Artisan',
    'Auth'       => 'Illuminate\Support\Facades\Auth',
    'Blade'      => 'Illuminate\Support\Facades\Blade',
    'Bus'        => 'Illuminate\Support\Facades\Bus',
    'Cache'      => 'Illuminate\Support\Facades\Cache',
    'Config'     => 'Illuminate\Support\Facades\Config',
    'Cookie'     => 'Illuminate\Support\Facades\Cookie',
    'Crypt'      => 'Illuminate\Support\Facades\Crypt',
    'DB'         => 'Illuminate\Support\Facades\DB',
    'Event'      => 'Illuminate\Support\Facades\Event',
    'File'       => 'Illuminate\Support\Facades\File',
    'Gate'       => 'Illuminate\Support\Facades\Gate',
    'Hash'       => 'Illuminate\Support\Facades\Hash',
    'Input'      => 'Illuminate\Support\Facades\Input',
    'Lang'       => 'Illuminate\Support\Facades\Lang',
    'Log'        => 'Illuminate\Support\Facades\Log',
    'Mail'       => 'Illuminate\Support\Facades\Mail',
    'Password'   => 'Illuminate\Support\Facades\Password',
    'Queue'      => 'Illuminate\Support\Facades\Queue',
    'Redirect'   => 'Illuminate\Support\Facades\Redirect',
    'Redis'     => 'Illuminate\Support\Facades\Redis',
    'Request'   => 'Illuminate\Support\Facades\Request',
    'Response'  => 'Illuminate\Support\Facades\Response',
    'Route'     => 'Illuminate\Support\Facades\Route',
    'Schema'    => 'Illuminate\Support\Facades\Schema',
    'Session'   => 'Illuminate\Support\Facades\Session',
    'Storage'   => 'Illuminate\Support\Facades\Storage',
    'URL'       => 'Illuminate\Support\Facades\URL',
    'Validator' => 'Illuminate\Support\Facades\Validator',
    'View'      => 'Illuminate\Support\Facades\View',
],

// Other configuration items.
```

Examining the above code sample, it can be deduced that to add a new facade alias we simply need to add a new entry to the `aliases` array. The key of the entry will become the name of the alias and the value of the entry will become the class being aliased. To make this clearer, in the above list `Response` is aliasing `Illuminate\Support\Facades\Response`, and both classes can be used interchangeably.

We will use our `Math` facade from earlier, and we will assume it was defined in the `Our\Applications\Namespace`. We could create an alias like so:

```
    // Previous alias entries.

    'Math' => 'Our\Applications\Namespace\Math',
```

Aliasing Other Classes

Classes, other than facades, can be added to the `aliases` configuration entry. This will cause them to be available under whatever name was provided, and in the global namespace. Although this can be convenient and useful thing to do, other options should be examined first.

11.4 Facade Class Reference

The following tables will list all the facades that are available by default. In addition, they will display the name of the service container binding as well as the class behind the facade. If a particular facade provides additional methods (that are not necessarily available in the underlying class), it will appear in the third table "Facades Providing Additional Methods". Any additional methods will be explained later in the section.

Facade Service Container Binding

Facade	Service Container Binding
App	`app`
Artisan	`Illuminate\Contracts\Console\Kernel`
Auth	`auth`
Blade	See “Notes on Blade”
Bus	`Illuminate\Contracts\Bus\Dispatcher`
Cache	`cache`
Config	`config`
Cookie	`cookie`
Crypt	`encrypter`
DB	`db`
Event	`events`
File	`files`
Gate	`Illuminate\Contracts\Auth\Access\Gate`
Hash	`hash`
Input	`request`
Lang	`translator`
Log	`log`
Mail	`mailer`
Password	`auth.password`
Queue	`queue`
Redirect	`redirect`
Redis	`redis`
Request	`request`
Response	`Illuminate\Contracts\Routing\ResponseFactory`
Route	`router`
Schema	See “Notes on Schema”
Session	`session`
Storage	`filesystem`
URL	`url`
Validator	`validator`
View	`view`

Facade Class Reference

Facade	Resolved Class
App	`Illuminate\Foundation\Application`
Artisan	`App\Console\Kernel`
Auth	`Illuminate\Auth\AuthManager`
Blade	`Illuminate\View\Compilers\BladeCompiler`
Bus	`Illuminate\Bus\Dispatcher`
Cache	`Illuminate\Cache\CacheManager`
Config	`Illuminate\Config\Repository`
Cookie	`Illuminate\Cookie\CookieJar`
Crypt	`Illuminate\Encryption\Encrypter`
DB	`Illuminate\Database\DatabaseManager`

Facade Class Reference

Facade	Resolved Class
Event	`Illuminate\Events\Dispatcher`
File	`Illuminate\Filesystem\Filesystem`
Gate	`Illuminate\Auth\Access\Gate`
Hash	`Illuminate\Hashing\BcryptHasher`
Input	`Illuminate\Http\Request`
Lang	`Illuminate\Translation\Translator`
Log	`Illuminate\Log\Writer`
Mail	`Illuminate\Mail\Mailer`
Password	`Illuminate\Auth\Passwords\PasswordBroker`
Queue	`Illuminate\Queue\QueueManager`
Redirect	`Illuminate\Routing\Redirector`
Redis	`Illuminate\Redis\Database`
Request	`Illuminate\Http\Request`
Response	`Illuminate\Routing\ResponseFactory`
Route	`Illuminate\Routing\Router`
Schema	`Illuminate\Database\Schema\MySqlBuilder`
Session	`Illuminate\Session\SessionManager`
Storage	`Illuminate\Contracts\Filesystem\Factory`
URL	`Illuminate\Routing\UrlGenerator`
Validator	`Illuminate\Validator\Factory`
View	`Illuminate\View\Factory`

Facades Providing Additional Methods

Facade	Number of Additional Methods
Cookie	2
Input	2
Schema	2

11.4.1 Notes on `Blade`

Most facades request a concrete class implementation from the service container based off of some abstract string representation. However, the `Blade` facade retrieve an `Illuminate\View\Engines\EngineResolver` instance from the `Illuminate\View\Factory`.

The `Illuminate\View\Engines\EngineResolver` is a class that returns template compilers based on a given key name. By default, the following compilers and engines are available:

Compiler/Engine Name	Concrete Class Implementation
`php`	`Illuminate\View\Engines\PhpEngine`
`blade`	`Illuminate\View\Compilers\BladeCompiler`

Developers can manually create an instance of the `BladeCompiler` themselves like so (this sample is provided for demonstration purposes):

```php
<?php

use Illuminate\Support\Facades\App;

$bladeCompiler = App::make('view')->getEngineResolver()
                     ->resolve('blade')->getCompiler();
```

11.4.2 Notes on `Schema`

Like the `Blade` facade, the `Schema` facade does not simply resolve some instance from the service container. The `Schema` facade returns an instance of `Illuminate\Database\Schema\Builder` configured to use the default connection that appears in the database database configuration.

11.4.3 Additional `Cookie` Methods

The `Cookie` facade defines two additional methods. These methods access other, related, components. These methods exist to simplify accessing related components.

11.4.3.1 `has($key)`

The `has` function will check if a cookie with the given `$key` exists for the current request.

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

The `get` function will retrieve a cookie from the current request with the given `$key`. A `$default` value can be supplied and will be returned if a cookie with the given `$key` does not exist.

11.4.4 Additional `Input` Methods

The `Input` facade defines one extra method. Facades define extra methods to provide simpler access to underlying sub-systems, or to call functions on other, related components.

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

The `get` method will *get* an item from the input data, such as when a user posts a form or an API request is being processed. The `get` method can be used for requests with the following HTTP verbs:

- GET
- POST
- PUT

- DELETE

The `get` method will invoke the `input` method on an instance of the `Illuminate\Http\Request` class.

The `get` method looks up the data based on the given `$key`. A `$default` value can be supplied and will be returned if the given `$key` is not found in the request data.

11.4.5 Additional `Schema` Methods

The `Schema` facade defines one extra method. Facades define extra methods to provide simpler access to underlying sub-systems, or to call functions on other, related components.

11.4.5.1 `connection($name)`

The `connection` method will return a new schema builder (`Illuminate\Database\Schema\Builder`) instance for the given connection. The `$name` is the name of the connection as it appears in the database configuration file.

11.5 Resolving the Class Behind a Facade

It is possible to quickly resolve the class behind a facade. Facades expose a public method `getFacadeRoot` which will return the instance of the underlying object the facade is forwarding method calls to. It is convenient that `getFacadeRoot` returns an object instance because PHP's `get_class`[3] method can then be used to retrieve the fully-qualified name of the facade's underlying class implementation.

```
// Getting the class name of the underlying facade instance.
$className = get_class(Auth::getFacadeRoot());
```

In the above code example, the `$className` variable would contain the value `Illuminate\Auth \AuthManager`.

This method of determining a facade's underlying class can be expanded on to create a function that will list every facade's underlying class for the current Laravel installation:

[3]http://php.net/manual/en/function.get-class.php

```php
/**
 * Generates an HTML table containing all registered
 * facades and the underlying class instances.
 *
 * @return string
 */
function getFacadeReferenceTable()
{
    $html = '';

    // An array of all the facades that should be printed in the table.
    $facades = [
        'App',  'Artisan', 'Auth', 'Blade', 'Bus',
        'Cache', 'Config', 'Cookie', 'Crypt', 'DB',
        'Event', 'File', 'Hash', 'Input', 'Lang', 'Log',
        'Mail', 'Password', 'Queue', 'Redis', 'Redirect',
        'Request', 'Response', 'Route', 'Schema', 'Session',
        'Storage', 'URL', 'Validator', 'View'
    ];

    // Boilerplate HTML to open an HTML table.
    $html =  '<table><thead><tr><th>Facade</th>';
    $html .= '<th>Underlying Class</th></tr><tbody>';

    foreach ($facades as $facade)
    {
        $html .= '<tr><td>',$facade,'</td><td>';
        $html .= get_class(call_user_func($facade.'::getFacadeRoot'));
        $html .= '</td></tr>';
    }

    // Boilerplate HTML to close an HTML table.
    $html .= '</tbody></table>';

    return $html;
}
```