Helpers

Introduction

Laravel includes a variety of global "helper" PHP functions. Many of these functions are used by the framework itself; however, you are free to use them in your own applications if you find them convenient.

Available Methods

Arrays & Objects

Arr::add
Arr::collapse
Arr::divide
Arr::dot
Arr::except
Arr::first
Arr::flatten
Arr::forget
Arr::get
Arr::has
Arr::last
Arr::only
Arr::pluck
Arr::prepend
Arr::pull
Arr::random
Arr::set
Arr::sort
Arr::sortRecursive
Arr::where
Arr::wrap
data_fill
data_get
data_set
head
last

Paths

app_path
base_path
config_path
database_path
mix
public_path
resource_path
storage_path

Strings

__
class_basename
e
preg_replace_array
Str::after
Str::before
Str::camel
Str::contains
Str::containsAll
Str::endsWith
Str::finish
Str::is
Str::kebab
Str::limit
Str::orderedUuid
Str::plural
Str::random
Str::replaceArray
Str::replaceFirst
Str::replaceLast
Str::singular
Str::slug
Str::snake
Str::start
Str::startsWith
Str::studly
Str::title
Str::uuid
Str::words
trans
trans_choice

URLs

action
asset
route
secure_asset
secure_url
url

Miscellaneous

abort
abort_if
abort_unless
app
auth
back
bcrypt
blank
broadcast
cache
class_uses_recursive
collect
config
cookie
csrf_field
csrf_token
dd
decrypt
dispatch
dispatch_now
dump
encrypt
env
event
factory
filled
info
logger
method_field
now
old
optional
policy
redirect
report
request
rescue
resolve
response
retry
session
tap
throw_if
throw_unless
today
trait_uses_recursive
transform
validator
value
view
with

Method Listing

Arrays & Objects

`Arr::add()`

The `Arr::add` method adds a given key / value pair to an array if the given key doesn't already exist in the array or is set to `null`:

```
use Illuminate\Support\Arr;

$array = Arr::add(['name' => 'Desk'], 'price', 100);

// ['name' => 'Desk', 'price' => 100]

$array = Arr::add(['name' => 'Desk', 'price' => null], 'price', 100);

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

`Arr::collapse()`

The `Arr::collapse` method collapses an array of arrays into a single array:

```
use Illuminate\Support\Arr;

$array = Arr::collapse([[1, 2, 3], [4, 5, 6], [7, 8, 9]]);

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

`Arr::divide()`

The `Arr::divide` method returns two arrays, one containing the keys, and the other containing the values of the given array:

```
use Illuminate\Support\Arr;

[$keys, $values] = Arr::divide(['name' => 'Desk']);

// $keys: ['name']

// $values: ['Desk']
```

`Arr::dot()`

The `Arr::dot` method flattens a multi-dimensional array into a single level array that uses "dot" notation to indicate depth:

```
use Illuminate\Support\Arr;

$array = ['products' => ['desk' => ['price' => 100]]];

$flattened = Arr::dot($array);

// ['products.desk.price' => 100]
```

`Arr::except()`

The `Arr::except` method removes the given key / value pairs from an array:

```
use Illuminate\Support\Arr;

$array = ['name' => 'Desk', 'price' => 100];

$filtered = Arr::except($array, ['price']);

// ['name' => 'Desk']
```

`Arr::first()`

The `Arr::first` method returns the first element of an array passing a given truth test:

```
use Illuminate\Support\Arr;

$array = [100, 200, 300];

$first = Arr::first($array, function ($value, $key) {
    return $value >= 150;
});

// 200
```

A default value may also be passed as the third parameter to the method. This value will be returned if no value passes the truth test:

```
use Illuminate\Support\Arr;

$first = Arr::first($array, $callback, $default);
```

`Arr::flatten()`

The `Arr::flatten` method flattens a multi-dimensional array into a single level array:

```
use Illuminate\Support\Arr;

$array = ['name' => 'Joe', 'languages' => ['PHP', 'Ruby']];

$flattened = Arr::flatten($array);

// ['Joe', 'PHP', 'Ruby']
```

`Arr::forget()`

The `Arr::forget` method removes a given key / value pair from a deeply nested array using "dot" notation:

```
use Illuminate\Support\Arr;

$array = ['products' => ['desk' => ['price' => 100]]];

Arr::forget($array, 'products.desk');

// ['products' => []]
```

`Arr::get()`

The `Arr::get` method retrieves a value from a deeply nested array using "dot" notation:

```
use Illuminate\Support\Arr;

$array = ['products' => ['desk' => ['price' => 100]]];

$price = Arr::get($array, 'products.desk.price');

// 100
```

The `Arr::get` method also accepts a default value, which will be returned if the specific key is not found:

```
use Illuminate\Support\Arr;

$discount = Arr::get($array, 'products.desk.discount', 0);

// 0
```

`Arr::has()`

The `Arr::has` method checks whether a given item or items exists in an array using "dot" notation:

```
use Illuminate\Support\Arr;

$array = ['product' => ['name' => 'Desk', 'price' => 100]];

$contains = Arr::has($array, 'product.name');

// true

$contains = Arr::has($array, ['product.price', 'product.discount']);

// false
```

`Arr::last()`

The `Arr::last` method returns the last element of an array passing a given truth test:

```php
use Illuminate\Support\Arr;

$array = [100, 200, 300, 110];

$last = Arr::last($array, function ($value, $key) {
    return $value >= 150;
});

// 300
```

A default value may be passed as the third argument to the method. This value will be returned if no value passes the truth test:

```php
use Illuminate\Support\Arr;

$last = Arr::last($array, $callback, $default);
```

`Arr::only()`

The `Arr::only` method returns only the specified key / value pairs from the given array:

```php
use Illuminate\Support\Arr;

$array = ['name' => 'Desk', 'price' => 100, 'orders' => 10];

$slice = Arr::only($array, ['name', 'price']);

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

`Arr::pluck()`

The `Arr::pluck` method retrieves all of the values for a given key from an array:

```php
use Illuminate\Support\Arr;

$array = [
    ['developer' => ['id' => 1, 'name' => 'Taylor']],
    ['developer' => ['id' => 2, 'name' => 'Abigail']],
];

$names = Arr::pluck($array, 'developer.name');

// ['Taylor', 'Abigail']
```

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

```php
use Illuminate\Support\Arr;

$names = Arr::pluck($array, 'developer.name', 'developer.id');

// [1 => 'Taylor', 2 => 'Abigail']
```

`Arr::prepend()`

The `Arr::prepend` method will push an item onto the beginning of an array:

```php
use Illuminate\Support\Arr;

$array = ['one', 'two', 'three', 'four'];

$array = Arr::prepend($array, 'zero');

// ['zero', 'one', 'two', 'three', 'four']
```

If needed, you may specify the key that should be used for the value:

```php
use Illuminate\Support\Arr;

$array = ['price' => 100];

$array = Arr::prepend($array, 'Desk', 'name');

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

`Arr::pull()`

The `Arr::pull` method returns and removes a key / value pair from an array:

```
use Illuminate\Support\Arr;

$array = ['name' => 'Desk', 'price' => 100];

$name = Arr::pull($array, 'name');

// $name: Desk

// $array: ['price' => 100]
```

A default value may be passed as the third argument to the method. This value will be returned if the key doesn't exist:

```
use Illuminate\Support\Arr;

$value = Arr::pull($array, $key, $default);
```

`Arr::random()`

The `Arr::random` method returns a random value from an array:

```
use Illuminate\Support\Arr;

$array = [1, 2, 3, 4, 5];

$random = Arr::random($array);

// 4 - (retrieved randomly)
```

You may also specify the number of items to return as an optional second argument. Note that providing this argument will return an array, even if only one item is desired:

```
use Illuminate\Support\Arr;

$items = Arr::random($array, 2);

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

`Arr::set()`

The `Arr::set` method sets a value within a deeply nested array using "dot" notation:

```
use Illuminate\Support\Arr;

$array = ['products' => ['desk' => ['price' => 100]]];

Arr::set($array, 'products.desk.price', 200);

// ['products' => ['desk' => ['price' => 200]]]
```

`Arr::sort()`

The `Arr::sort` method sorts an array by its values:

```
use Illuminate\Support\Arr;

$array = ['Desk', 'Table', 'Chair'];

$sorted = Arr::sort($array);

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

You may also sort the array by the results of the given Closure:

```
use Illuminate\Support\Arr;

$array = [
    ['name' => 'Desk'],
    ['name' => 'Table'],
    ['name' => 'Chair'],
];

$sorted = array_values(Arr::sort($array, function ($value) {
    return $value['name'];
}));
```

```
/*
    [
        ['name' => 'Chair'],
        ['name' => 'Desk'],
        ['name' => 'Table'],
    ]
*/
```

`Arr::sortRecursive()`

The `Arr::sortRecursive` method recursively sorts an array using the `sort` function for numeric sub=arrays and `ksort` for associative sub-arrays:

```
use Illuminate\Support\Arr;

$array = [
    ['Roman', 'Taylor', 'Li'],
    ['PHP', 'Ruby', 'JavaScript'],
    ['one' => 1, 'two' => 2, 'three' => 3],
];

$sorted = Arr::sortRecursive($array);

/*
    [
        ['JavaScript', 'PHP', 'Ruby'],
        ['one' => 1, 'three' => 3, 'two' => 2],
        ['Li', 'Roman', 'Taylor'],
    ]
*/
```

`Arr::where()`

The `Arr::where` method filters an array using the given Closure:

```
use Illuminate\Support\Arr;

$array = [100, '200', 300, '400', 500];

$filtered = Arr::where($array, function ($value, $key) {
    return is_string($value);
});

// [1 => '200', 3 => '400']
```

`Arr::wrap()`

The `Arr::wrap` method wraps the given value in an array. If the given value is already an array it will not be changed:

```
use Illuminate\Support\Arr;

$string = 'Laravel';

$array = Arr::wrap($string);

// ['Laravel']
```

If the given value is null, an empty array will be returned:

```
use Illuminate\Support\Arr;

$nothing = null;

$array = Arr::wrap($nothing);

// []
```

`data_fill()`

The `data_fill` function sets a missing value within a nested array or object using "dot" notation:

```
$data = ['products' => ['desk' => ['price' => 100]]];

data_fill($data, 'products.desk.price', 200);

// ['products' => ['desk' => ['price' => 100]]]

data_fill($data, 'products.desk.discount', 10);
```

```
// ['products' => ['desk' => ['price' => 100, 'discount' => 10]]]
```

This function also accepts asterisks as wildcards and will fill the target accordingly:

```
$data = [
    'products' => [
        ['name' => 'Desk 1', 'price' => 100],
        ['name' => 'Desk 2'],
    ],
];

data_fill($data, 'products.*.price', 200);

/*
    [
        'products' => [
            ['name' => 'Desk 1', 'price' => 100],
            ['name' => 'Desk 2', 'price' => 200],
        ],
    ]
*/
```

`data_get()`

The `data_get` function retrieves a value from a nested array or object using "dot" notation:

```
$data = ['products' => ['desk' => ['price' => 100]]];

$price = data_get($data, 'products.desk.price');

// 100
```

The `data_get` function also accepts a default value, which will be returned if the specified key is not found:

```
$discount = data_get($data, 'products.desk.discount', 0);

// 0
```

The function also accepts wildcards using asterisks, which may target any key of the array or object:

```
$data = [
    'product-one' => ['name' => 'Desk 1', 'price' => 100],
    'product-two' => ['name' => 'Desk 2', 'price' => 150],
];

data_get($data, '*.name');

// ['Desk 1', 'Desk 2'];
```

`data_set()`

The `data_set` function sets a value within a nested array or object using "dot" notation:

```
$data = ['products' => ['desk' => ['price' => 100]]];

data_set($data, 'products.desk.price', 200);

// ['products' => ['desk' => ['price' => 200]]]
```

This function also accepts wildcards and will set values on the target accordingly:

```
$data = [
    'products' => [
        ['name' => 'Desk 1', 'price' => 100],
        ['name' => 'Desk 2', 'price' => 150],
    ],
];

data_set($data, 'products.*.price', 200);

/*
    [
        'products' => [
            ['name' => 'Desk 1', 'price' => 200],
```

```
            ['name' => 'Desk 2', 'price' => 200],
        ],
    ]
*/
```

By default, any existing values are overwritten. If you wish to only set a value if it doesn't exist, you may pass `false` as the fourth argument:

```
$data = ['products' => ['desk' => ['price' => 100]]];

data_set($data, 'products.desk.price', 200, false);

// ['products' => ['desk' => ['price' => 100]]]
```

`head()`

The `head` function returns the first element in the given array:

```
$array = [100, 200, 300];

$first = head($array);

// 100
```

`last()`

The `last` function returns the last element in the given array:

```
$array = [100, 200, 300];

$last = last($array);

// 300
```

Paths

`app_path()`

The `app_path` function returns the fully qualified path to the `app` directory. You may also use the `app_path` function to generate a fully qualified path to a file relative to the application directory:

```
$path = app_path();

$path = app_path('Http/Controllers/Controller.php');
```

`base_path()`

The `base_path` function returns the fully qualified path to the project root. You may also use the `base_path` function to generate a fully qualified path to a given file relative to the project root directory:

```
$path = base_path();

$path = base_path('vendor/bin');
```

`config_path()`

The `config_path` function returns the fully qualified path to the `config` directory. You may also use the `config_path` function to generate a fully qualified path to a given file within the application's configuration directory:

```
$path = config_path();

$path = config_path('app.php');
```

`database_path()`

The `database_path` function returns the fully qualified path to the `database` directory. You may also use the `database_path` function to generate a fully qualified path to a given file within the database directory:

```php
$path = database_path();

$path = database_path('factories/UserFactory.php');
```

`mix()`

The `mix` function returns the path to a versioned Mix file:

```php
$path = mix('css/app.css');
```

`public_path()`

The `public_path` function returns the fully qualified path to the `public` directory. You may also use the `public_path` function to generate a fully qualified path to a given file within the public directory:

```php
$path = public_path();

$path = public_path('css/app.css');
```

`resource_path()`

The `resource_path` function returns the fully qualified path to the `resources` directory. You may also use the `resource_path` function to generate a fully qualified path to a given file within the resources directory:

```php
$path = resource_path();

$path = resource_path('sass/app.scss');
```

`storage_path()`

The `storage_path` function returns the fully qualified path to the `storage` directory. You may also use the `storage_path` function to generate a fully qualified path to a given file within the storage directory:

```php
$path = storage_path();

$path = storage_path('app/file.txt');
```

Strings

`__()`

The `__` function translates the given translation string or translation key using your localization files:

```php
echo __('Welcome to our application');

echo __('messages.welcome');
```

If the specified translation string or key does not exist, the `__` function will return the given value. So, using the example above, the `__` function would return `messages.welcome` if that translation key does not exist.

`class_basename()`

The `class_basename` function returns the class name of the given class with the class' namespace removed:

```php
$class = class_basename('Foo\Bar\Baz');

// Baz
```

`e()`

The `e` function runs PHP's `htmlspecialchars` function with the `double_encode` option set to `true` by default:

```php
echo e('<html>foo</html>');
```

```
// &lt;html&gt;foo&lt;/html&gt;
```

`preg_replace_array()`

The `preg_replace_array` function replaces a given pattern in the string sequentially using an array:

```
$string = 'The event will take place between :start and :end';

$replaced = preg_replace_array('/:[a-z_]+/', ['8:30', '9:00'], $string);

// The event will take place between 8:30 and 9:00
```

`Str::after()`

The `Str::after` method returns everything after the given value in a string:

```
use Illuminate\Support\Str;

$slice = Str::after('This is my name', 'This is');

// ' my name'
```

`Str::before()`

The `Str::before` method returns everything before the given value in a string:

```
use Illuminate\Support\Str;

$slice = Str::before('This is my name', 'my name');

// 'This is '
```

`Str::camel()`

The `Str::camel` method converts the given string to `camelCase`:

```
use Illuminate\Support\Str;

$converted = Str::camel('foo_bar');

// fooBar
```

`Str::contains()`

The `Str::contains` method determines if the given string contains the given value (case sensitive):

```
use Illuminate\Support\Str;

$contains = Str::contains('This is my name', 'my');

// true
```

You may also pass an array of values to determine if the given string contains any of the values:

```
use Illuminate\Support\Str;

$contains = Str::contains('This is my name', ['my', 'foo']);

// true
```

`Str::containsAll()`

The `Str::containsAll` method determines if the given string contains all array values:

```
use Illuminate\Support\Str;

$containsAll = Str::containsAll('This is my name', ['my', 'name']);

// true
```

`Str::endsWith()`

The `Str::endsWith` method determines if the given string ends with the given value:

```
use Illuminate\Support\Str;

$result = Str::endsWith('This is my name', 'name');

// true
```

`Str::finish()`

The `Str::finish` method adds a single instance of the given value to a string if it does not already end with the value:

```
use Illuminate\Support\Str;

$adjusted = Str::finish('this/string', '/');

// this/string/

$adjusted = Str::finish('this/string/', '/');

// this/string/
```

`Str::is()`

The `Str::is` method determines if a given string matches a given pattern. Asterisks may be used to indicate wildcards:

```
use Illuminate\Support\Str;

$matches = Str::is('foo*', 'foobar');

// true

$matches = Str::is('baz*', 'foobar');

// false
```

`Str::kebab()`

The `Str::kebab` method converts the given string to `kebab-case`:

```
use Illuminate\Support\Str;

$converted = Str::kebab('fooBar');

// foo-bar
```

`Str::limit()`

The `Str::limit` method truncates the given string at the specified length:

```
use Illuminate\Support\Str;

$truncated = Str::limit('The quick brown fox jumps over the lazy dog', 20);

// The quick brown fox...
```

You may also pass a third argument to change the string that will be appended to the end:

```
use Illuminate\Support\Str;

$truncated = Str::limit('The quick brown fox jumps over the lazy dog', 20, ' (...)

// The quick brown fox (...)
```

`Str::orderedUuid()`

The `Str::orderedUuid` method generates a "timestamp first" UUID that may be efficiently stored in an indexed database column:

```
use Illuminate\Support\Str;
```

```
return (string) Str::orderedUuid();
```

`Str::plural()`

The `Str::plural` method converts a string to its plural form. This function currently only supports the English language:

```
use Illuminate\Support\Str;

$plural = Str::plural('car');

// cars

$plural = Str::plural('child');

// children
```

You may provide an integer as a second argument to the function to retrieve the singular or plural form of the string:

```
use Illuminate\Support\Str;

$plural = Str::plural('child', 2);

// children

$plural = Str::plural('child', 1);

// child
```

`Str::random()`

The `Str::random` method generates a random string of the specified length. This function uses PHP's `random_bytes` function:

```
use Illuminate\Support\Str;

$random = Str::random(40);
```

`Str::replaceArray()`

The `Str::replaceArray` method replaces a given value in the string sequentially using an array:

```
use Illuminate\Support\Str;

$string = 'The event will take place between ? and ?';

$replaced = Str::replaceArray('?', ['8:30', '9:00'], $string);

// The event will take place between 8:30 and 9:00
```

`Str::replaceFirst()`

The `Str::replaceFirst` method replaces the first occurrence of a given value in a string:

```
use Illuminate\Support\Str;

$replaced = Str::replaceFirst('the', 'a', 'the quick brown fox jumps over the lazy

// a quick brown fox jumps over the lazy dog
```

`Str::replaceLast()`

The `Str::replaceLast` method replaces the last occurrence of a given value in a string:

```
use Illuminate\Support\Str;

$replaced = Str::replaceLast('the', 'a', 'the quick brown fox jumps over the lazy

// the quick brown fox jumps over a lazy dog
```

`Str::singular()`

The `Str::singular` method converts a string to its singular form. This function currently only supports the English language:

```
use Illuminate\Support\Str;

$singular = Str::singular('cars');

// car

$singular = Str::singular('children');

// child
```

`Str::slug()`

The `Str::slug` method generates a URL friendly "slug" from the given string:

```
use Illuminate\Support\Str;

$slug = Str::slug('Laravel 5 Framework', '-');

// laravel-5-framework
```

`Str::snake()`

The `Str::snake` method converts the given string to `snake_case`:

```
use Illuminate\Support\Str;

$converted = Str::snake('fooBar');

// foo_bar
```

`Str::start()`

The `Str::start` method adds a single instance of the given value to a string if it does not already start with the value:

```
use Illuminate\Support\Str;

$adjusted = Str::start('this/string', '/');

// /this/string

$adjusted = Str::start('/this/string', '/');

// /this/string
```

`Str::startsWith()`

The `Str::startsWith` method determines if the given string begins with the given value:

```
use Illuminate\Support\Str;

$result = Str::startsWith('This is my name', 'This');

// true
```

`Str::studly()`

The `Str::studly` method converts the given string to `StudlyCase`:

```
use Illuminate\Support\Str;

$converted = Str::studly('foo_bar');

// FooBar
```

`Str::title()`

The `Str::title` method converts the given string to `Title Case`:

```php
use Illuminate\Support\Str;

$converted = Str::title('a nice title uses the correct case');

// A Nice Title Uses The Correct Case
```

`Str::uuid()`

The `Str::uuid` method generates a UUID (version 4):

```php
use Illuminate\Support\Str;

return (string) Str::uuid();
```

`Str::words()`

The `Str::words` method limits the number of words in a string:

```php
use Illuminate\Support\Str;

return Str::words('Perfectly balanced, as all things should be.', 3, ' >>>');

// Perfectly balanced, as >>>
```

`trans()`

The `trans` function translates the given translation key using your localization files:

```php
echo trans('messages.welcome');
```

If the specified translation key does not exist, the `trans` function will return the given key. So, using the example above, the `trans` function would return `messages.welcome` if the translation key does not exist.

`trans_choice()`

The `trans_choice` function translates the given translation key with inflection:

```php
echo trans_choice('messages.notifications', $unreadCount);
```

If the specified translation key does not exist, the `trans_choice` function will return the given key. So, using the example above, the `trans_choice` function would return `messages.notifications` if the translation key does not exist.

URLs

`action()`

The `action` function generates a URL for the given controller action. You do not need to pass the full namespace of the controller. Instead, pass the controller class name relative to the `App\Http\Controllers` namespace:

```php
$url = action('HomeController@index');

$url = action([HomeController::class, 'index']);
```

If the method accepts route parameters, you may pass them as the second argument to the method:

```php
$url = action('UserController@profile', ['id' => 1]);
```

`asset()`

The `asset` function generates a URL for an asset using the current scheme of the request (HTTP or HTTPS):

```php
$url = asset('img/photo.jpg');
```

You can configure the asset URL host by setting the `ASSET_URL` variable in your `.env`

file. This can be useful if you host your assets on an external service like Amazon S3:

```
// ASSET_URL=http://example.com/assets

$url = asset('img/photo.jpg'); // http://example.com/assets/img/photo.jpg
```

`route()`

The `route` function generates a URL for the given named route:

```
$url = route('routeName');
```

If the route accepts parameters, you may pass them as the second argument to the method:

```
$url = route('routeName', ['id' => 1]);
```

By default, the `route` function generates an absolute URL. If you wish to generate a relative URL, you may pass `false` as the third argument:

```
$url = route('routeName', ['id' => 1], false);
```

`secure_asset()`

The `secure_asset` function generates a URL for an asset using HTTPS:

```
$url = secure_asset('img/photo.jpg');
```

`secure_url()`

The `secure_url` function generates a fully qualified HTTPS URL to the given path:

```
$url = secure_url('user/profile');

$url = secure_url('user/profile', [1]);
```

`url()`

The `url` function generates a fully qualified URL to the given path:

```
$url = url('user/profile');

$url = url('user/profile', [1]);
```

If no path is provided, a `Illuminate\Routing\UrlGenerator` instance is returned:

```
$current = url()->current();

$full = url()->full();

$previous = url()->previous();
```

Miscellaneous

`abort()`

The `abort` function throws an HTTP exception which will be rendered by the exception handler:

```
abort(403);
```

You may also provide the exception's response text and custom response headers:

```
abort(403, 'Unauthorized.', $headers);
```

`abort_if()`

The `abort_if` function throws an HTTP exception if a given boolean expression evaluates to `true`:

```
abort_if(! Auth::user()->isAdmin(), 403);
```

Like the `abort` method, you may also provide the exception's response text as the third argument and an array of custom response headers as the fourth argument.

`abort_unless()`

The `abort_unless` function throws an HTTP exception if a given boolean expression evaluates to `false`:

```
abort_unless(Auth::user()->isAdmin(), 403);
```

Like the `abort` method, you may also provide the exception's response text as the third argument and an array of custom response headers as the fourth argument.

`app()`

The `app` function returns the service container instance:

```
$container = app();
```

You may pass a class or interface name to resolve it from the container:

```
$api = app('HelpSpot\API');
```

`auth()`

The `auth` function returns an authenticator instance. You may use it instead of the `Auth` facade for convenience:

```
$user = auth()->user();
```

If needed, you may specify which guard instance you would like to access:

```
$user = auth('admin')->user();
```

`back()`

The `back` function generates a redirect HTTP response to the user's previous location:

```
return back($status = 302, $headers = [], $fallback = false);

return back();
```

`bcrypt()`

The `bcrypt` function hashes the given value using Bcrypt. You may use it as an alternative to the `Hash` facade:

```
$password = bcrypt('my-secret-password');
```

`blank()`

The `blank` function returns whether the given value is "blank":

```
blank('');
blank('   ');
blank(null);
blank(collect());

// true

blank(0);
blank(true);
blank(false);
```

```
// false
```

For the inverse of `blank`, see the `filled` method.

`broadcast()`

The `broadcast` function broadcasts the given event to its listeners:

```
broadcast(new UserRegistered($user));
```

`cache()`

The `cache` function may be used to get values from the cache. If the given key does not exist in the cache, an optional default value will be returned:

```
$value = cache('key');

$value = cache('key', 'default');
```

You may add items to the cache by passing an array of key / value pairs to the function. You should also pass the number of seconds or duration the cached value should be considered valid:

```
cache(['key' => 'value'], 300);

cache(['key' => 'value'], now()->addSeconds(10));
```

`class_uses_recursive()`

The `class_uses_recursive` function returns all traits used by a class, including traits used by all of its parent classes:

```
$traits = class_uses_recursive(App\User::class);
```

`collect()`

The `collect` function creates a collection instance from the given value:

```
$collection = collect(['taylor', 'abigail']);
```

`config()`

The `config` function gets the value of a configuration variable. The configuration values may be accessed using "dot" syntax, which includes the name of the file and the option you wish to access. A default value may be specified and is returned if the configuration option does not exist:

```
$value = config('app.timezone');

$value = config('app.timezone', $default);
```

You may set configuration variables at runtime by passing an array of key / value pairs:

```
config(['app.debug' => true]);
```

`cookie()`

The `cookie` function creates a new cookie instance:

```
$cookie = cookie('name', 'value', $minutes);
```

`csrf_field()`

The `csrf_field` function generates an HTML `hidden` input field containing the value of the CSRF token. For example, using Blade syntax:

```
{{ csrf_field() }}
```

`csrf_token()`

The `csrf_token` function retrieves the value of the current CSRF token:

```
$token = csrf_token();
```

`dd()`

The `dd` function dumps the given variables and ends execution of the script:

```
dd($value);

dd($value1, $value2, $value3, ...);
```

If you do not want to halt the execution of your script, use the `dump` function instead.

`decrypt()`

The `decrypt` function decrypts the given value using Laravel's encrypter:

```
$decrypted = decrypt($encrypted_value);
```

`dispatch()`

The `dispatch` function pushes the given job onto the Laravel job queue:

```
dispatch(new App\Jobs\SendEmails);
```

`dispatch_now()`

The `dispatch_now` function runs the given job immediately and returns the value from its `handle` method:

```
$result = dispatch_now(new App\Jobs\SendEmails);
```

`dump()`

The `dump` function dumps the given variables:

```
dump($value);

dump($value1, $value2, $value3, ...);
```

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

`encrypt()`

The `encrypt` function encrypts the given value using Laravel's encrypter:

```
$encrypted = encrypt($unencrypted_value);
```

`env()`

The `env` function retrieves the value of an environment variable or returns a default value:

```
$env = env('APP_ENV');

// Returns 'production' if APP_ENV is not set...
$env = env('APP_ENV', 'production');
```

If you execute the `config:cache` command during your deployment process, you should be sure that you are only calling the `env` function from within your configuration files.

Once the configuration has been cached, the `.env` file will not be loaded and all calls to the `env` function will return `null`.

`event()`

The `event` function dispatches the given event to its listeners:

```
event(new UserRegistered($user));
```

`factory()`

The `factory` function creates a model factory builder for a given class, name, and amount. It can be used while testing or seeding:

```
$user = factory(App\User::class)->make();
```

`filled()`

The `filled` function returns whether the given value is not "blank":

```
filled(0);
filled(true);
filled(false);

// true

filled('');
filled('   ');
filled(null);
filled(collect());

// false
```

For the inverse of `filled`, see the `blank` method.

`info()`

The `info` function will write information to the log:

```
info('Some helpful information!');
```

An array of contextual data may also be passed to the function:

```
info('User login attempt failed.', ['id' => $user->id]);
```

`logger()`

The `logger` function can be used to write a `debug` level message to the log:

```
logger('Debug message');
```

An array of contextual data may also be passed to the function:

```
logger('User has logged in.', ['id' => $user->id]);
```

A logger instance will be returned if no value is passed to the function:

```
logger()->error('You are not allowed here.');
```

`method_field()`

The `method_field` function generates an HTML `hidden` input field containing the spoofed value of the form's HTTP verb. For example, using Blade syntax:

```
<form method="POST">
    {{ method_field('DELETE') }}
</form>
```

`now()`

The `now` function creates a new `Illuminate\Support\Carbon` instance for the current time:

```
$now = now();
```

`old()`

The `old` function retrieves an old input value flashed into the session:

```
$value = old('value');

$value = old('value', 'default');
```

`optional()`

The `optional` function accepts any argument and allows you to access properties or call methods on that object. If the given object is `null`, properties and methods will return `null` instead of causing an error:

```
return optional($user->address)->street;

{!! old('name', optional($user)->name) !!}
```

The `optional` function also accepts a Closure as its second argument. The Closure will be invoked if the value provided as the first argument is not null:

```
return optional(User::find($id), function ($user) {
    return new DummyUser;
});
```

`policy()`

The `policy` method retrieves a policy instance for a given class:

```
$policy = policy(App\User::class);
```

`redirect()`

The `redirect` function returns a redirect HTTP response, or returns the redirector instance if called with no arguments:

```
return redirect($to = null, $status = 302, $headers = [], $secure = null);

return redirect('/home');

return redirect()->route('route.name');
```

`report()`

The `report` function will report an exception using your exception handler's `report` method:

```
report($e);
```

`request()`

The `request` function returns the current request instance or obtains an input item:

```
$request = request();

$value = request('key', $default);
```

`rescue()`

The `rescue` function executes the given Closure and catches any exceptions that occur during its execution. All exceptions that are caught will be sent to your

exception handler's `report` method; however, the request will continue processing:

```
return rescue(function () {
    return $this->method();
});
```

You may also pass a second argument to the `rescue` function. This argument will be the "default" value that should be returned if an exception occurs while executing the Closure:

```
return rescue(function () {
    return $this->method();
}, false);

return rescue(function () {
    return $this->method();
}, function () {
    return $this->failure();
});
```

`resolve()`

The `resolve` function resolves a given class or interface name to its instance using the service container:

```
$api = resolve('HelpSpot\API');
```

`response()`

The `response` function creates a response instance or obtains an instance of the response factory:

```
return response('Hello World', 200, $headers);

return response()->json(['foo' => 'bar'], 200, $headers);
```

`retry()`

The `retry` function attempts to execute the given callback until the given maximum attempt threshold is met. If the callback does not throw an exception, its return value will be returned. If the callback throws an exception, it will automatically be retried. If the maximum attempt count is exceeded, the exception will be thrown:

```
return retry(5, function () {
    // Attempt 5 times while resting 100ms in between attempts...
}, 100);
```

`session()`

The `session` function may be used to get or set session values:

```
$value = session('key');
```

You may set values by passing an array of key / value pairs to the function:

```
session(['chairs' => 7, 'instruments' => 3]);
```

The session store will be returned if no value is passed to the function:

```
$value = session()->get('key');

session()->put('key', $value);
```

`tap()`

The `tap` function accepts two arguments: an arbitrary `$value` and a Closure. The `$value` will be passed to the Closure and then be returned by the `tap` function. The return value of the Closure is irrelevant:

```
$user = tap(User::first(), function ($user) {
    $user->name = 'taylor';
```

```
    $user->save();
});
```

If no Closure is passed to the `tap` function, you may call any method on the given `$value`. The return value of the method you call will always be `$value`, regardless of what the method actually returns in its definition. For example, the Eloquent `update` method typically returns an integer. However, we can force the method to return the model itself by chaining the `update` method call through the `tap` function:

```
$user = tap($user)->update([
    'name' => $name,
    'email' => $email,
]);
```

To add a `tap` method to a class, you may add the `Illuminate\Support\Traits\Tappable` trait to the class. The `tap` method of this trait accepts a Closure as its only argument. The object instance itself will be passed to the Closure and then be returned by the `tap` method:

```
return $user->tap(function ($user) {
    //
});
```

`throw_if()`

The `throw_if` function throws the given exception if a given boolean expression evaluates to `true`:

```
throw_if(! Auth::user()->isAdmin(), AuthorizationException::class);

throw_if(
    ! Auth::user()->isAdmin(),
    AuthorizationException::class,
    'You are not allowed to access this page'
);
```

`throw_unless()`

The `throw_unless` function throws the given exception if a given boolean expression evaluates to `false`:

```
throw_unless(Auth::user()->isAdmin(), AuthorizationException::class);

throw_unless(
    Auth::user()->isAdmin(),
    AuthorizationException::class,
    'You are not allowed to access this page'
);
```

`today()`

The `today` function creates a new `Illuminate\Support\Carbon` instance for the current date:

```
$today = today();
```

`trait_uses_recursive()`

The `trait_uses_recursive` function returns all traits used by a trait:

```
$traits = trait_uses_recursive(\Illuminate\Notifications\Notifiable::class);
```

`transform()`

The `transform` function executes a `Closure` on a given value if the value is not blank and returns the result of the `Closure`:

```
$callback = function ($value) {
    return $value * 2;
};

$result = transform(5, $callback);
```

```
// 10
```

A default value or `Closure` may also be passed as the third parameter to the method. This value will be returned if the given value is blank:

```
$result = transform(null, $callback, 'The value is blank');

// The value is blank
```

`validator()`

The `validator` function creates a new validator instance with the given arguments. You may use it instead of the `Validator` facade for convenience:

```
$validator = validator($data, $rules, $messages);
```

`value()`

The `value` function returns the value it is given. However, if you pass a `Closure` to the function, the `Closure` will be executed then its result will be returned:

```
$result = value(true);

// true

$result = value(function () {
    return false;
});

// false
```

`view()`

The `view` function retrieves a view instance:

```
return view('auth.login');
```

`with()`

The `with` function returns the value it is given. If a `Closure` is passed as the second argument to the function, the `Closure` will be executed and its result will be returned:

```
$callback = function ($value) {
    return (is_numeric($value)) ? $value * 2 : 0;
};

$result = with(5, $callback);

// 10

$result = with(null, $callback);

// 0

$result = with(5, null);

// 5
```

Become a Laravel Partner

Laravel Partners are elite shops providing top-notch Laravel development and consulting. Each of our partners can help you craft a beautiful, well-architected project.

Our Partners

Laravel

Highlights

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

Resources

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

Partners

- Vehikl
- Tighten Co.
- Kirschbaum
- Byte 5
- 64Robots
- Cubet
- DevSquad
- Ideil
- Cyber-Duck
- ABOUT YOU
- Become A Partner

Ecosystem

- Vapor
- Forge
- Envoyer
- Horizon
- Lumen
- Nova
- Echo
- Valet
- Mix
- Spark
- Cashier
- Homestead
- Dusk
- Passport
- Scout
- Socialite

Laravel is a web application framework with expressive, elegant syntax. We believe development must be an enjoyable and creative experience to be truly fulfilling. Laravel attempts to take the pain out of development by easing common tasks used in most web projects.