Lumen

lumen.laravel.com

DOCUMENTATION

5.0

Laravel Lumen Documentation - 5.0

https://lumen.laravel.com/docs/

eBook compiled from the source

https://github.com/laravel/lumen-docs/

by david@mundosaparte.com

Get the latest version at https://github.com/driade/laravel-lumen-book

Date: Thursday, 09-May-19 16:15:28 CEST

Contents

Prologue

- Introduction
- Release Notes

Getting Started

- Installation
- Configuration

The Basics

- Routing
- Middleware
- Controllers
- Requests
- Responses
- Views

Architecture Foundations

- Service Providers
- Service Container

Core Features

- Cache
- Database
- Encryption
- Errors & Logging
- Events
- Helpers
- Queues
- Unit Testing
- Validation

Full-Stack Features

- Authentication
- Filesystem / Cloud Storage
- Hashing
- Mail
- Pagination
- Session
- Templates

Prologue

Introduction

- What Is Lumen?
- When Should I Use Lumen?
- Which Laravel Features Does Lumen Include?

What Is Lumen?

Lumen is a "micro-framework" built on top of Laravel's components, and is the official micro-framework of Laravel. Lumen is built for speed, and is one of the fastest PHP micro-frameworks available - even significantly faster than similar frameworks such as Silex.

However, unlike many other micro-frameworks, Lumen lets you tap into the full power of Laravel's features, such as routing, dependency injection, the Eloquent ORM, migrations, queued jobs, and even scheduled commands.

Laravel is already fast and powerful, but Lumen strips away many of the configuration and customization options that Laravel provides in order to shave every millisecond possible off of your service's load time.

The stunningly fast speed of Lumen, combined with the convenience of Laravel's features gives you a "best of both worlds" micro-framework that is truly a joy to work with.

When Should I Use Lumen?

Lumen is designed to build blazing fast micro-services and APIs. For example, if there is one aspect of your Laravel application that receives drastically more traffic than the rest of the application, you may choose to build that aspect of the application as a small, separate Lumen application.

By reducing the load on your primary Laravel application, you can cut server costs since applications built on Lumen do not require as much server power as a full Laravel application.

Of course, Lumen applications can queue jobs for your main Laravel application to process. Laravel and Lumen are designed to make a perfect team, and, when used together, allow you to build powerful, micro-service driven applications.

Lumen is also a great fit for building fast JSON APIs since these applications do not typically require many "full-stack" features such as HTTP sessions, cookies, and templating.

Lumen Limitations

Lumen is not as configurable as the Laravel framework. For example, it is not possible to override any framework "bootstrappers" to drastically alter how the framework is constructed. Also, unlike Laravel, Lumen is not intended to be used with additional Laravel "packages" such as debug bars, CMS systems, etc.

In addition, Lumen does not use Symfony's Routing component. Instead, `nikic/fast-route` is used for greater performance. If you need Symfony Routing features such as sub-domain routing or optional parameters, you should use the full Laravel framework.

If you do choose to use the full-stack Laravel framework, do not worry that your application will suffer from poor performance. The full-stack Laravel framework powers many very large, enterprise level applications receiving up to 15,000,000 requests per day.

Lumen Features

Lumen includes many of the same features as the full-stack Laravel framework:

- Blade Templating
- Caching
- Command Scheduler
- Controllers
- Eloquent ORM
- Error Handling
- Database Abstraction
- Dependency Injection
- Logging
- Queued Jobs

By utilizing a unique bootstrapping process, Lumen is able to provide a robust feature set while still delivering extremely high performance, making it the perfect solution for PHP micro-services.

Of course, you may explore the documentation for each of these features (and others) by browsing this documentation.

Prologue

Release Notes

- 5.0.4
- 5.0 (Based On Laravel 5.0.x)

Lumen 5.0.4

When upgrading to Lumen 5.0.4, you should update your `bootstrap/app.php` file's creation of the Lumen application class to the following:

```
$app = new Laravel\Lumen\Application(
        realpath(__DIR__.'/../')
);
```

> **Note:** This is not a **required** change; however, it should prevent some bugs when using the Artisan CLI and PHP's built-in web server.

Lumen 5.0

Lumen 5.0 is the initial release of the Lumen framework, and is based on the Laravel 5.x series of PHP components.

Getting Started

Installation

- Install Composer
- Install Lumen
- Server Requirements

Install Composer

Lumen utilizes Composer to manage its dependencies. So, before using Lumen, you will need to make sure you have Composer installed on your machine.

Install Lumen

Via Lumen Installer

First, download the Lumen installer using Composer.

```
composer global require "laravel/lumen-installer=~1.0"
```

Make sure to place the `~/.composer/vendor/bin` directory in your PATH so the `lumen` executable can be located by your system.

Once installed, the simple `lumen new` command will create a fresh Lumen installation in the directory you specify. For instance, `lumen new service` would create a directory named `service` containing a fresh Lumen installation with all dependencies installed. This method of installation is much faster than installing via Composer:

```
lumen new service
```

Via Composer Create-Project

You may also install Lumen by issuing the Composer `create-project` command in your terminal:

```
composer create-project laravel/lumen --prefer-dist
```

Server Requirements

The Lumen framework has a few system requirements:

- PHP >= 5.4
- Mcrypt PHP Extension
- OpenSSL PHP Extension
- Mbstring PHP Extension
- Tokenizer PHP Extension

Configuration

Lumen needs almost no other configuration out of the box. You are free to get started developing!

You may also want to configure a few additional components of Lumen, such as:

- Cache
- Database
- Queue
- Session

Permissions

Lumen may require some permissions to be configured: folders within `storage` directory need to be writable.

Pretty URLs

Apache

The framework ships with a `public/.htaccess` file that is used to allow URLs without `index.php`. If you use Apache to serve your Lumen application, be sure to enable the `mod_rewrite` module.

If the `.htaccess` file that ships with Lumen does not work with your Apache installation, try this one:

```
Options +FollowSymLinks
RewriteEngine On

RewriteCond %{REQUEST_FILENAME} !-d
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^ index.php [L]
```

Nginx

On Nginx, the following directive in your site configuration will allow "pretty" URLs:

```
location / {
        try_files $uri $uri/ /index.php?$query_string;
}
```

Of course, when using Homestead, pretty URLs will be configured automatically.

Getting Started

Configuration

- Introduction
- After Installation
- Configuration Files
- Pretty URLs

Introduction

Unlike Laravel, Lumen only uses a single `.env` configuration file which can be used to configure the various aspects of the framework. The `.env.example` that ships with the framework can be used as a starting-point for your Lumen configuration.

> **Note:** If you would like to use the `vlucas/phpdotenv` library to load your environment variables into the `$_ENV` PHP super-global, you should uncomment the call to `Dotenv::load` in your `bootstrap/app.php` file.

After Installation

Lumen needs very little configuration out of the box. However, you should set your `APP_KEY` configuration option in the `.env` file. This value should be a 32 character, random string.

However, you may also want to configure a few additional components of Laravel, such as:

- Cache
- Database
- Queue
- Session

> **Note:** You should never have the `APP_DEBUG` configuration option set to `true` for a production application.

Permissions

Laravel may require some permissions to be configured: folders within `storage` and the `bootstrap/cache` directory require write access by the web server.

Configuration Files

By default, Lumen uses a single `.env` file to configure your application. However, you may use full, "Laravel style" configuration files if you wish. The default configuration files are stored in `vendor/laravel/lumen-framework/config` directory. Lumen will use your copy of the configuration file if you copy and paste one of the files into a `config` directory within your project root.

Using full configuration files will give you more control over some aspects of Lumen's configuration, such as configuring multiple storage "disks" or read / write database connections.

Custom Configuration Files

You may also create your own custom configuration files and load them using the `$app->configure()` method. For example, if your configuration file is located in `config/options.php`, you can load the file like so:

```
$app->configure('options');
```

Pretty URLs

Apache

The framework ships with a `public/.htaccess` file that is used to allow URLs without `index.php`. If you use Apache to serve your Laravel application, be sure to enable the `mod_rewrite` module.

If the `.htaccess` file that ships with Laravel does not work with your Apache installation, try this one:

```
Options +FollowSymLinks
RewriteEngine On

RewriteCond %{REQUEST_FILENAME} !-d
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^ index.php [L]
```

If your web host doesn't allow the `FollowSymlinks` option, try replacing it with `Options +SymLinksIfOwnerMatch`.

Nginx

On Nginx, the following directive in your site configuration will allow "pretty" URLs:

```
location / {
        try_files $uri $uri/ /index.php?$query_string;
}
```

Of course, when using Homestead, pretty URLs will be configured automatically.

The Basics

HTTP Routing

- Basic Routing
- Route Parameters
- Named Routes
- Route Groups
- Route Prefixing
- CSRF Protection
- Method Spoofing
- Throwing 404 Errors

Basic Routing

You will define most of the routes for your application in the `app/Http/routes.php` file, which is loaded by the `bootstrap/app.php` file. Like Laravel, the most basic Lumen routes simply accept a URI and a `Closure`:

Basic GET Route

```
$app->get('/', function() {
        return 'Hello World';
});
```

Other Basic Routes

```
$app->post('foo/bar', function() {
        return 'Hello World';
});

$app->patch('foo/bar', function() {
        //
});

$app->put('foo/bar', function() {
        //
});

$app->delete('foo/bar', function() {
        //
});
```

Often, you will need to generate URLs to your routes, you may do so using the `url` helper:

```
$url = url('foo');
```

Routing Requests To Controllers

If you are interested in routing requests to classes, check out the documentation on controllers.

Route Parameters

Of course, you can capture segments of the request URI within your route:

Basic Route Parameter

```
$app->get('user/{id}', function($id) {
        return 'User '.$id;
});
```

Regular Expression Parameter Constraints

> **Note:** This is the only portion of Lumen that is not directly portable to the full Laravel framework. If you choose to upgrade your Lumen application to Laravel, your regular expression constraints must be moved to a `where` method call on the route.

```
$app->get('user/{name:[A-Za-z]+}', function($name) {
        //
});
```

Named Routes

Named routes allow you to conveniently generate URLs or redirects for a specific route. You may specify a name for a route with the `as` array key:

```
$app->get('user/profile', ['as' => 'profile', function() {
        //
}]);
```

You may also specify route names for controller actions:

```
$app->get('user/profile', [
        'as' => 'profile', 'uses' => 'App\Http\Controllers\UserController@showProfile'
]);
```

Now, you may use the route's name when generating URLs or redirects:

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

$redirect = redirect()->route('profile');
```

Route Groups

Sometimes you may need to apply middleware to a group of routes. Instead of specifying the middleware on each route, you may use a route group.

Shared attributes are specified in an array format as the first parameter to the `$app->group()` method.

Middleware

Middleware is applied to all routes within the group by defining the list of middleware with the `middleware` parameter on the group attribute array. Middleware will be executed in the order you define this array:

```
$app->group(['middleware' => 'foo|bar'], function($app)
{
        $app->get('/', function() {
                // Uses Foo & Bar Middleware
        });

        $app->get('user/profile', function() {
                // Uses Foo & Bar Middleware
        });
});
```

Namespaces

You may use the `namespace` parameter in your group attribute array to specify the namespace for all controllers within the group:

```
$app->group(['namespace' => 'App\Http\Controllers\Admin'], function($app) {
        // Controllers Within The "App\Http\Controllers\Admin" Namespace
});
```

Route Prefixing

A group of routes may be prefixed by using the `prefix` option in the attributes array of a group:

```
$app->group(['prefix' => 'admin'], function($app)
{
        $app->get('users', function()
        {
                // Matches The "/admin/users" URL
        });
});
```

You can also utilize the `prefix` parameter to pass common parameters to your routes:

URL Parameter In Prefix

```
$app->group(['prefix' => 'accounts/{account_id}'], function($app)
{
        $app->get('detail', function($account_id)
        {
                //
        });
});
```

CSRF Protection

Note: You must enable sessions to utilize this feature of Lumen.

Lumen, like Laravel, makes it easy to protect your application from cross-site request forgeries. Cross-site request forgeries are a type of malicious exploit whereby unauthorized commands are performed on behalf of the authenticated user.

Lumen automatically generates a CSRF "token" for each active user session managed by the application. This token is used to verify that the authenticated user is the one actually making the requests to the application.

Insert The CSRF Token Into A Form

```
<input type="hidden" name="_token" value="<?php echo csrf_token(); ?>">
```

Of course, using the Blade templating engine:

```
<input type="hidden" name="_token" value="{{ csrf_token() }}">
```

You do not need to manually verify the CSRF token on POST, PUT, or DELETE requests. If it is enabled in the `bootstrap/app.php` file, the `Laravel\Lumen\Http\Middleware\VerifyCsrfToken` HTTP middleware will verify that the token in the request input matches the token stored in the session.

X-CSRF-TOKEN

In addition to looking for the CSRF token as a "POST" parameter, the middleware will also check for the `X-CSRF-TOKEN` request header. You could, for example, store the token in a "meta" tag and instruct jQuery to add it to all request headers:

```
<meta name="csrf-token" content="{{ csrf_token() }}" />

$.ajaxSetup({
        headers: {
                'X-CSRF-TOKEN': $('meta[name="csrf-token"]').attr('content')
        }
});
```

Now all AJAX requests will automatically include the CSRF token:

```
$.ajax({
   url: "/foo/bar",
})
```

X-XSRF-TOKEN

Lumen also stores the CSRF token in a `XSRF-TOKEN` cookie. You can use the cookie value to set the `X-XSRF-TOKEN` request header. Some Javascript frameworks, like Angular, do this automatically for you.

> Note: The difference between the `X-CSRF-TOKEN` and `X-XSRF-TOKEN` is that the first uses a plain text value and the latter uses an encrypted value, because cookies in Lumen are always encrypted when the global middleware in the `bootstrap/app.php` file are enabled.

Method Spoofing

HTML forms do not support `PUT`, `PATCH` or `DELETE` actions. So, when defining `PUT`, `PATCH` or `DELETE` routes that are called from an HTML form, you will need to add a hidden `_method` field to the form.

The value sent with the `_method` field will be used as the HTTP request method. For example:

```
<form action="/foo/bar" method="POST">
        <input type="hidden" name="_method" value="PUT">
        <input type="hidden" name="_token" value="{{ csrf_token() }}">
</form>
```

Throwing 404 Errors

There are two ways to manually trigger a 404 error from a route. First, you may use the `abort` helper:

```
abort(404);
```

The `abort` helper simply throws a `Symfony\Component\HttpKernel\Exception\HttpException` with the specified status code.

Secondly, you may manually throw an instance of `Symfony\Component\HttpKernel\Exception\NotFoundHttpException`.

More information on handling 404 exceptions and using custom responses for these errors may be found in the errors section of the documentation.

The Basics

HTTP Middleware

- Introduction
- Defining Middleware
- Registering Middleware
- Terminable Middleware

Introduction

HTTP middleware provide a convenient mechanism for filtering HTTP requests entering your application. For example, Lumen includes a middleware that verifies the CSRF token of your application.

Of course, middleware can be written to perform a variety of tasks besides CSRF validation. A CORS middleware might be responsible for adding the proper headers to all responses leaving your application. A logging middleware might log all incoming requests to your application.

All middleware are typically located in the `app/Http/Middleware` directory.

Defining Middleware

To create a new middleware, simply create a class with a `handle` method like the following:

```
public function handle($request, $next)
{
        return $next($request);
}
```

For example, let's create a middleware that will only allow access to the route if the supplied `age` is greater than 200. Otherwise, we will redirect the users back to the "home" URI.

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

use Closure;

class OldMiddleware {

        /**
         * Run the request filter.
         *
         * @param  \Illuminate\Http\Request  $request
         * @param  \Closure  $next
         * @return mixed
         */
        public function handle($request, Closure $next)
        {
                if ($request->input('age') < 200) {
                        return redirect('home');
                }

                return $next($request);
        }

}
```

As you can see, if the given `age` is less than `200`, the middleware will return an HTTP redirect to the client; otherwise, the request will be passed further into the application. To pass the request deeper into the application (allowing the middleware to "pass"), simply call the `$next` callback with the `$request`.

It's best to envision middleware as a series of "layers" HTTP requests must pass through before they hit your application. Each layer can examine the request and even reject it entirely.

Before / *After* Middleware

Whether a middleware runs before or after a request depends on the middleware itself. This middleware would perform some task **before** the request is handled by the application:

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

use Closure;

class BeforeMiddleware implements Middleware {

        public function handle($request, Closure $next)
```

```
        {
                // Perform action

                return $next($request);
        }
}
```

However, this middleware would perform its task **after** the request is handled by the application:

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

use Closure;

class AfterMiddleware implements Middleware {

        public function handle($request, Closure $next)
        {
                $response = $next($request);

                // Perform action

                return $response;
        }
}
```

Registering Middleware

Global Middleware

If you want a middleware to be run during every HTTP request to your application, simply list the middleware class in the `$app->middleware()` call of your `bootstrap/app.php` file.

Assigning Middleware To Routes

If you would like to assign middleware to specific routes, you should first assign the middleware a short-hand key in your `bootstrap/app.php` file. By default, the `$app->routeMiddleware()` method call of this file contains the entries for the route middleware defined by your application. To add your own, simply append it to this list and assign it a key of your choosing. For example:

```
$app->routeMiddleware([
    'old' => 'App\Http\Middleware\OldMiddleware',
]);
```

Once the middleware has been defined in the HTTP kernel, you may use the `middleware` key in the route options array:

```
$app->get('admin/profile', ['middleware' => 'old', function() {
        //
}]);
```

Terminable Middleware

Sometimes a middleware may need to do some work after the HTTP response has already been sent to the browser. For example, the "session" middleware included with Laravel and Lumen writes the session data to storage *after* the response has been sent to the browser. To accomplish this, define the middleware as "terminable" by implementing the `Illuminate\Contracts\Routing\TerminableMiddleware` contract:

```
use Closure;
use Illuminate\Contracts\Routing\TerminableMiddleware;

class StartSession implements TerminableMiddleware {

        public function handle($request, Closure $next)
        {
                return $next($request);
        }

        public function terminate($request, $response)
        {
                // Store the session data...
        }

}
```

As you can see, in addition to defining a `handle` method, the `TerminableMiddleware` contract requires a `terminate` method. This method receives both the request and the response. Once you have defined a terminable middleware, you should add it to the list of global middlewares in your HTTP kernel.

The Basics

HTTP Controllers

- Introduction
- Basic Controllers
- Controller Middleware
- Dependency Injection & Controllers

Introduction

Instead of defining all of your request handling logic in a single `routes.php` file, you may wish to organize this behavior using Controller classes. Controllers can group related HTTP request handling logic into a class. Controllers are typically stored in the `app/Http/Controllers` directory.

Basic Controllers

Here is an example of a basic controller class:

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

use App\User;
use App\Http\Controllers\Controller;

class UserController extends Controller {

	/**
	 * Show the profile for the given user.
	 *
	 * @param  int  $id
	 * @return Response
	 */
	public function showProfile($id)
	{
		return view('user.profile', ['user' => User::findOrFail($id)]);
	}

}
```

We can route to the controller action like so:

```
$app->get('user/{id}', 'App\Http\Controllers\UserController@showProfile');
```

> **Note:** All controllers should extend the base `App\Http\Controllers\Controller` class.

Naming Controller Routes

Like Closure routes, you may specify names on controller routes:

```
$app->get('foo', ['uses' => 'App\Http\Controllers\FooController@method', 'as' => 'name']);
```

These names can be used to generate URLs to the controller actions:

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

If the route has parameters, you may specify them like so:

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

Controller Middleware

Middleware may be specified on controller routes like so:

```
$app->get('profile', [
	'middleware' => 'auth',
	'uses' => 'App\Http\Controllers\UserController@showProfile'
]);
```

Additionally, you may specify middleware within your controller's constructor:

```
class UserController extends Controller {

	/**
	 * Instantiate a new UserController instance.
	 */
```

```
        public function __construct()
        {
                $this->middleware('auth');

                $this->middleware('log', ['only' => ['fooAction', 'barAction']]);

                $this->middleware('subscribed', ['except' => ['fooAction', 'barAction']]);
        }

}
```

Dependency Injection & Controllers

Constructor Injection

The Lumen / Laravel service container is used to resolve all controllers. As a result, you are able to type-hint any dependencies your controller may need in its constructor:

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

use App\Http\Controllers\Controller;
use App\Repositories\UserRepository;

class UserController extends Controller {

        /**
         * The user repository instance.
         */
        protected $users;

        /**
         * Create a new controller instance.
         *
         * @param  UserRepository  $users
         * @return void
         */
        public function __construct(UserRepository $users)
        {
                $this->users = $users;
        }

}
```

Method Injection

In addition to constructor injection, you may also type-hint dependencies on your controller's methods. For example, let's type-hint the `Request` instance on one of our methods:

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

use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class UserController extends Controller {

        /**
         * Store a new user.
         *
         * @param  Request  $request
         * @return Response
         */
        public function store(Request $request)
        {
                $name = $request->input('name');

                //
        }

}
```

If your controller method is also expecting input from a route parameter, simply list your route arguments after your other dependencies:

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

use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class UserController extends Controller {

        /**
         * Update the specified user.
```

```
         *
         * @param  Request  $request
         * @param  int  $id
         * @return Response
         */
        public function update(Request $request, $id)
        {
                //
        }

}
```

The Basics

HTTP Requests

- Obtaining A Request Instance
- Retrieving Input
- Old Input
- Cookies
- Files
- Other Request Information

Obtaining A Request Instance

Via Facade

The `Request` facade will grant you access to the current request that is bound in the container. For example:

```
$name = Request::input('name');
```

Remember, if you are in a namespace, you will have to import the `Request` facade using a `use Request;` statement at the top of your class file.

Via Dependency Injection

To obtain an instance of the current HTTP request via dependency injection, you should type-hint the class on your controller constructor or method. The current request instance will automatically be injected by the service container:

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

use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class UserController extends Controller {

	/**
	 * Store a new user.
	 *
	 * @param  Request  $request
	 * @return Response
	 */
	public function store(Request $request)
	{
		$name = $request->input('name');

		//
	}

}
```

If your controller method is also expecting input from a route parameter, simply list your route arguments after your other dependencies:

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

use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class UserController extends Controller {

	/**
	 * Update the specified user.
	 *
	 * @param  Request  $request
	 * @param  int  $id
	 * @return Response
	 */
	public function update(Request $request, $id)
	{
		//
	}

}
```

Retrieving Input

Retrieving An Input Value

Using a few simple methods, you may access all user input from your `Illuminate\Http\Request` instance. You do not need to worry about the HTTP verb used for the request, as input is accessed in the same way for all verbs.

```
$name = Request::input('name');
```

Retrieving A Default Value If The Input Value Is Absent

```
$name = Request::input('name', 'Sally');
```

Determining If An Input Value Is Present

```
if (Request::has('name')) {
        //
}
```

Getting All Input For The Request

```
$input = Request::all();
```

Getting Only Some Of The Request Input

```
$input = Request::only('username', 'password');

$input = Request::except('credit_card');
```

When working on forms with "array" inputs, you may use dot notation to access the arrays:

```
$input = Request::input('products.0.name');
```

Old Input

> **Note:** Before utilizing this feature of Lumen, you must enable sessions.

Lumen, like Laravel, also allows you to keep input from one request during the next request. For example, you may need to re-populate a form after checking it for validation errors.

Flashing Input To The Session

The `flash` method will flash the current input to the session so that it is available during the user's next request to the application:

```
Request::flash();
```

Flashing Only Some Input To The Session

```
Request::flashOnly('username', 'email');

Request::flashExcept('password');
```

Flash & Redirect

Since you often will want to flash input in association with a redirect to the previous page, you may easily chain input flashing onto a redirect.

```
return redirect('form')->withInput();

return redirect('form')->withInput(Request::except('password'));
```

Retrieving Old Data

To retrieve flashed input from the previous request, use the `old` method on the `Request` instance.

```
$username = Request::old('username');
```

If you are displaying old input within a Blade template, it is more convenient to use the `old` helper:

```
{{ old('username') }}
```

Cookies

To force all cookies to be encrypted and signed, you will need to uncomment the `EncryptCookies` middleware in your `bootstrap/app.php` file. All signed cookies created by the Lumen and Laravel frameworks are encrypted and signed with an

authentication code, meaning they will be considered invalid if they have been changed by the client.

Retrieving A Cookie Value

```
$value = Request::cookie('name');
```

Attaching A New Cookie To A Response

The `cookie` helper serves as a simple factory for generating new `Symfony\Component\HttpFoundation\Cookie` instances. The cookies may be attached to a `Response` instance using the `withCookie` method:

```
$response = new Illuminate\Http\Response('Hello World');

$response->withCookie(cookie('name', 'value', $minutes));
```

Creating A Cookie That Lasts Forever*

By "forever", we really mean five years.

```
$response->withCookie(cookie()->forever('name', 'value'));
```

Queueing Cookies

> **Note:** In order to utilize this feature of Lumen, you must uncomment the `AddQueuedCookiesToResponse` middleware in your `bootstrap/app.php` file.

You may also "queue" a cookie to be added to the outgoing response, even before that response has been created:

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

use Cookie;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
        /**
         * Update a resource
         *
         * @return Response
         */
         public function update()
         {
                Cookie::queue('name', 'value');

                return response('Hello World');
         }
}
```

Files

Retrieving An Uploaded File

```
$file = Request::file('photo');
```

Determining If A File Was Uploaded

```
if (Request::hasFile('photo')) {
        //
}
```

The object returned by the `file` method is an instance of the `Symfony\Component\HttpFoundation\File\UploadedFile` class, which extends the PHP `SplFileInfo` class and provides a variety of methods for interacting with the file.

Determining If An Uploaded File Is Valid

```
if (Request::file('photo')->isValid()) {
        //
}
```

Moving An Uploaded File

```
Request::file('photo')->move($destinationPath);

Request::file('photo')->move($destinationPath, $fileName);
```

Other File Methods

There are a variety of other methods available on `UploadedFile` instances. Check out the API documentation for the class for more information regarding these methods.

Other Request Information

The `Request` class provides many methods for examining the HTTP request for your application and extends the `Symfony\Component\HttpFoundation\Request` class. Here are some of the highlights.

Retrieving The Request URI

```
$uri = Request::path();
```

Retrieving The Request Method

```
$method = Request::method();

if (Request::isMethod('post')) {
        //
}
```

Determining If The Request Path Matches A Pattern

```
if (Request::is('admin/*')) {
        //
}
```

Get The Current Request URL

```
$url = Request::url();
```

The Basics

HTTP Responses

- Basic Responses
- Redirects
- Other Responses

Basic Responses

Returning Strings From Routes

The most basic response from a Lumen route is a string:

```
$app->get('/', function() {
        return 'Hello World';
});
```

Creating Custom Responses

However, for most routes and controller actions, you will be returning a full `Illuminate\Http\Response` instance or a view. Returning a full `Response` instance allows you to customize the response's HTTP status code and headers. A `Response` instance inherits from the `Symfony\Component\HttpFoundation\Response` class, providing a variety of methods for building HTTP responses:

```
use Illuminate\Http\Response;

return (new Response($content, $status))
              ->header('Content-Type', $value);
```

For convenience, you may also use the `response` helper:

```
return response($content, $status)
              ->header('Content-Type', $value);
```

> **Note:** For a full list of available `Response` methods, check out its API documentation and the Symfony API documentation.

Redirects

Redirect responses are typically instances of the `Illuminate\Http\RedirectResponse` class, and contain the proper headers needed to redirect the user to another URL.

Returning A Redirect

There are several ways to generate a `RedirectResponse` instance. The simplest method is to use the `redirect` helper method. When testing, it is not common to mock the creation of a redirect response, so using the helper method is almost always acceptable:

```
return redirect('user/login');
```

Returning A Redirect With Flash Data

> **Note:** Before using flash data, you must enable sessions.

Redirecting to a new URL and flashing data to the session are typically done at the same time. So, for convenience, you may create a `RedirectResponse` instance **and** flash data to the session in a single method chain:

```
return redirect('user/login')->with('message', 'Login Failed');
```

Redirecting To The Previous URL

You may wish to redirect the user to their previous location, for example, after a form submission. You can do so by using the `back` method:

```
return redirect()->back();

return redirect()->back()->withInput();
```

Returning A Redirect To A Named Route

When you call the `redirect` helper with no parameters, an instance of `Illuminate\Routing\Redirector` is returned, allowing you to call any method on the `Redirector` instance. For example, to generate a `RedirectResponse` to a named route, you may use the `route` method:

```
return redirect()->route('login');
```

Returning A Redirect To A Named Route With Parameters

If your route has parameters, you may pass them as the second argument to the `route` method.

```
// For a route with the following URI: profile/{id}

return redirect()->route('profile', ['id' => 1]);
```

If you are redirecting to a route with an "ID" parameter that is being populated from an Eloquent model, you may simply pass the model itself. The ID will be extracted automatically:

```
return redirect()->route('profile', ['id' => $user]);
```

Returning A Redirect To A Named Route Using Named Parameters

```
// For a route with the following URI: profile/{user}

return redirect()->route('profile', ['user' => 1]);
```

Other Responses

The `response` helper may be used to conveniently generate other types of response instances.

Creating A JSON Response

The `json` method will automatically set the `Content-Type` header to `application/json`:

```
return response()->json(['name' => 'Abigail', 'state' => 'CA']);
```

Creating A JSONP Response

```
return response()->json(['name' => 'Abigail', 'state' => 'CA'])
                 ->setCallback($request->input('callback'));
```

Creating A File Download Response

```
return response()->download($pathToFile);

return response()->download($pathToFile, $name, $headers);

return response()->download($pathToFile)->deleteFileAfterSend(true);
```

> **Note:** Symfony HttpFoundation, which manages file downloads, requires the file being downloaded to have an ASCII file name.

The Basics

Views

- Basic Usage

Basic Usage

Views contain the HTML served by your application, and serve as a convenient method of separating your controller and domain logic from your presentation logic. Views are stored in the `resources/views` directory.

A simple view looks like this:

```
<!-- View stored in resources/views/greeting.php -->

<!doctype html>
<html>
        <head>
                <title>Welcome!</title>
        </head>
        <body>
                <h1>Hello, <?php echo $name; ?></h1>
        </body>
</html>
```

The view may be returned to the browser like so:

```
$app->get('/', function() {
        return view('greeting', ['name' => 'James']);
});
```

As you can see, the first argument passed to the `view` helper corresponds to the name of the view file in the `resources/views` directory. The second argument passed to helper is an array of data that should be made available to the view.

Of course, views may also be nested within sub-directories of the `resources/views` directory. For example, if your view is stored at `resources/views/admin/profile.php`, it should be returned like so:

```
return view('admin.profile', $data);
```

Passing Data To Views

```
// Using conventional approach
$view = view('greeting')->with('name', 'Victoria');

// Using Magic Methods
$view = view('greeting')->withName('Victoria');
```

In the example above, the variable `$name` is made accessible to the view and contains `Victoria`.

If you wish, you may pass an array of data as the second parameter to the `view` helper:

```
$view = view('greetings', $data);
```

When passing information in this manner, `$data` should be an array with key/value pairs. Inside your view, you can then access each value using it's corresponding key, like `{{ $key }}` (assuming `$data['$key']` exists).

Determining If A View Exists

If you need to determine if a view exists, you may use the `exists` method:

```
if (view()->exists('emails.customer')) {
        //
}
```

Returning A View From A File Path

If you wish, you may generate a view from a fully-qualified file path:

```
return view()->file($pathToFile, $data);
```

Architecture Foundations

Service Providers

- Introduction
- Basic Provider Example
- Registering Providers

Introduction

Service providers are the central place of all Lumen application bootstrapping. Your own application, as well as all of Lumen's core services are bootstrapped via service providers.

But, what do we mean by "bootstrapped"? In general, we mean **registering** things, including registering service container bindings and event listeners. Service providers are the central place to configure your application.

If you open the `bootstrap/app.php` file included with Lumen, you will see a call to `$app->register()`. You may add additional calls to this method to register additional service providers.

In this overview you will learn how to write your own service providers and register them with your Lumen application.

Basic Provider Example

All service providers extend the `Illuminate\Support\ServiceProvider` class. This abstract class requires that you define at least one method on your provider: `register`.

The Register Method

Now, let's take a look at a basic service provider:

```
<?php namespace App\Providers;

use Riak\Connection;
use Illuminate\Support\ServiceProvider;

class RiakServiceProvider extends ServiceProvider {

	/**
	 * Register bindings in the container.
	 *
	 * @return void
	 */
	public function register()
	{
		$this->app->singleton('Riak\Contracts\Connection', function($app) {
			return new Connection($app['config']['riak']);
		});
	}

}
```

This service provider only defines a `register` method, and uses that method to define an implementation of `Riak\Contracts\Connection` in the service container. If you don't understand how the service container works, don't worry, we'll cover that soon.

This class is namespaced under `App\Providers` since that is the default location for service providers in Laravel. However, you are free to change this as you wish. Your service providers may be placed anywhere that Composer can autoload them.

Registering Providers

All service providers are registered in the `bootstrap/app.php` bootstrap file. This file contains a sample call to `$app->register()`.

To register your provider, simply add another call to this method:

```
$app->register('App\Providers\YourServiceProvider');
```

Architecture Foundations

Service Container

- Introduction
- Basic Usage
- Binding Interfaces To Implementations
- Contextual Binding
- Tagging
- Container Events

Introduction

Lumen utilizes the powerful Laravel service container, which is an amazing tool for managing class dependencies. Dependency injection is a fancy word that essentially means this: class dependencies are "injected" into the class via the constructor or, in some cases, "setter" methods.

Basic Usage

> **Note:** To better organize your container bindings, consider placing them in service providers.

Registering A Basic Resolver

There are several ways the service container can register dependencies, including Closure callbacks and binding interfaces to implementations. First, we'll explore Closure callbacks. A Closure resolver is registered in the container with a key (typically the class name) and a Closure that returns some value:

```
$app->bind('FooBar', function($app) {
        return new FooBar($app['SomethingElse']);
});
```

Registering A Singleton

Sometimes, you may wish to bind something into the container that should only be resolved once, and the same instance should be returned on subsequent calls into the container:

```
$app->singleton('FooBar', function($app) {
        return new FooBar($app['SomethingElse']);
});
```

Binding An Existing Instance Into The Container

You may also bind an existing object instance into the container using the `instance` method. The given instance will always be returned on subsequent calls into the container:

```
$fooBar = new FooBar(new SomethingElse);

$app->instance('FooBar', $fooBar);
```

Resolving

There are several ways to resolve something out of the container. First, you may use the `make` method:

```
$fooBar = $app->make('FooBar');
```

Automatic Resolution

Secondly, but importantly, you may simply "type-hint" the dependency in the constructor of a class that is resolved by the container, including controllers, event listeners, queue jobs, and more. The container will automatically inject the dependencies:

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

use App\Http\Controllers\Controller;
use App\Users\Repository as UserRepository;

class UserController extends Controller {

        /**
         * The user repository instance.
         */
        protected $users;
```

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

        /**
         * Show the user with the given ID.
         *
         * @param  int  $id
         * @return Response
         */
        public function show($id)
        {
                //
        }

}
```

Binding Interfaces To Implementations

A very powerful feature of the service container is its ability to bind an interface to a given implementation. For example, perhaps our application integrates with the Pusher web service for sending and receiving real-time events. If we are using Pusher's PHP SDK, we could inject an instance of the Pusher client into a class. First, let's register a binding for the SDK that binds it to an interface:

```
$app->bind('App\Contracts\EventPusher', 'App\Services\PusherEventPusher');
```

This tells the container that it should inject the `PusherEventPusher` when a class needs an implementation of `EventPusher`. Now we can type-hint the `EventPusher` interface in our constructor:

```
        /**
         * Create a new order handler instance.
         *
         * @param  EventPusher  $pusher
         * @return void
         */
        public function __construct(EventPusher $pusher)
        {
                $this->pusher = $pusher;
        }
```

Contextual Binding

Sometimes you may have two classes that utilize the same interface, but you wish to inject different implementations into each class. For example, when our system receives a new Order, we may want to send an event via PubNub rather than Pusher. Lumen provides a simple, fluent interface for defining this behavior:

```
$app->when('App\Service\CreateOrder')
          ->needs('App\Contracts\EventPusher')
          ->give('App\Services\PubNubEventPusher');
```

Tagging

Occasionally, you may need to resolve all of a certain "category" of binding. For example, perhaps you are building a report aggregator that receives an array of many different `Report` interface implementations. After registering the `Report` implementations, you can assign them a tag using the `tag` method:

```
$app->bind('SpeedReport', function() {
        //
});

$app->bind('MemoryReport', function() {
        //
});

$app->tag(['SpeedReport', 'MemoryReport'], 'reports');
```

Once the services have been tagged, you may easily resolve them all via the `tagged` method:

```
$app->bind('ReportAggregator', function($app) {
        return new ReportAggregator($app->tagged('reports'));
});
```

Container Events

Registering A Resolving Listener

The container fires an event each time it resolves an object. You may listen to this event using the `resolving` method:

```
$app->resolving(function($object, $app) {
        // Called when container resolves object of any type...
});

$app->resolving(function(FooBar $fooBar, $app) {
        // Called when container resolves objects of type "FooBar"...
});
```

The object being resolved will be passed to the callback.

Core Features

Cache

- Configuration
- Basic Usage

Configuration

The `CACHE_DRIVER` option in your `.env` file determines the cache "driver" to be used for the application. Of course, Lumen supports the same drivers as the full-stack Laravel framework, including Memcached and Redis:

- `array`
- `file`
- `memcached`
- `redis`
- `database`

Note: If you are using the `.env` file to configure your application, don't forget to uncomment the `Dotenv::load()` method in your `bootstrap/app.php` file.

Memcached

If you are using the Memcached driver, you may also set the `MEMCACHED_HOST` and `MEMCACHED_PORT` options in your `.env` configuration file.

Redis

Before using a Redis cache with Lumen, you will need to install the `predis/predis` (~1.0) and the `illuminate/redis` (~5.0) packages via Composer.

Database

When using the database cache driver, you will need to setup a table to contain the cache items. You'll find an example Schema declaration for the table below:

```
Schema::create('cache', function($table) {
    $table->string('key')->unique();
    $table->text('value');
    $table->integer('expiration');
});
```

Basic Usage

Note: If you intend to use the `Cache` facade, be sure to uncomment the `$app->withFacades()` call in your `bootstrap/app.php` file.

Storing An Item In The Cache

```
Cache::put('key', 'value', $minutes);
```

Using Carbon Objects To Set Expire Time

```
$expiresAt = Carbon::now()->addMinutes(10);

Cache::put('key', 'value', $expiresAt);
```

Storing An Item In The Cache If It Doesn't Exist

```
Cache::add('key', 'value', $minutes);
```

The `add` method will return `true` if the item is actually **added** to the cache. Otherwise, the method will return `false`.

Checking For Existence In Cache

```
if (Cache::has('key')) {
        //
}
```

Retrieving An Item From The Cache

```
$value = Cache::get('key');
```

Retrieving An Item Or Returning A Default Value

```
$value = Cache::get('key', 'default');

$value = Cache::get('key', function() { return 'default'; });
```

Storing An Item In The Cache Permanently

```
Cache::forever('key', 'value');
```

Sometimes you may wish to retrieve an item from the cache, but also store a default value if the requested item doesn't exist. You may do this using the `Cache::remember` method:

```
$value = Cache::remember('users', $minutes, function() {
        return DB::table('users')->get();
});
```

You may also combine the `remember` and `forever` methods:

```
$value = Cache::rememberForever('users', function() {
        return DB::table('users')->get();
});
```

Note that all items stored in the cache are serialized, so you are free to store any type of data.

Pulling An Item From The Cache

If you need to retrieve an item from the cache and then delete it, you may use the `pull` method:

```
$value = Cache::pull('key');
```

Removing An Item From The Cache

```
Cache::forget('key');
```

Core Features

Database

- Configuration
- Basic Usage
- Migrations

Configuration

Lumen makes connecting with databases and running queries extremely simple. Currently Laravel supports four database systems: MySQL, Postgres, SQLite, and SQL Server.

You may use the `DB_*` configuration options in your `.env` configuration file to configure your database settings, such as the driver, host, username, and password.

> **Note:** In order for your configuration values to be loaded, you will need to uncomment the `Dotenv::load()` method call in your `bootstrap/app.php` file.

Basic Usage

> **Note:** If you would like to use the `DB` facade, you should uncomment the `$app->withFacades()` call in your `bootstrap/app.php` file.

Basic Queries

To learn how to execute basic, raw SQL queries via the database component, you may consult the full Laravel documentation.

Query Builder

Lumen may also utilize the Laravel fluent query builder. To learn more about this feature, consult the full Laravel documentation.

Eloquent ORM

If you would like to use the Eloquent ORM, you should uncomment the `$app->withEloquent()` call in your `bootstrap/app.php` file.

Of course, you may easily use the full Eloquent ORM with Lumen. To learn how to use Eloquent, check out the full Laravel documentation.

Migrations

For further information on how to create database tables and run migrations, check out the Laravel documentation on the schema builder and migrator.

Core Features

Encryption

- Introduction
- Basic Usage

Introduction

Lumen provides facilities for strong AES encryption via the Mcrypt PHP extension.

Basic Usage

Encrypting A Value

```
$encrypted = Crypt::encrypt('secret');
```

> **Note:** Be sure to set a 32 character random string in the `APP_KEY` option of the `.env` file. Otherwise, encrypted values will not be secure.

Decrypting A Value

```
$decrypted = Crypt::decrypt($encryptedValue);
```

Setting The Cipher & Mode

You may also set the cipher and mode used by the encrypter:

```
Crypt::setMode('ctr');

Crypt::setCipher($cipher);
```

Core Features

Errors & Logging

- Configuration
- Handling Errors
- HTTP Exceptions
- Logging

Configuration

Lumen is pre-configured with Monolog, a PSR-3 compatible logger.

By default, the logger is configured to use a single log file that is stored in the `storage/logs` directory; however, you may customize this behavior as needed. Since Lumen uses the popular Monolog logging library, you can take advantage of the variety of handlers that Monolog offers.

Error Detail

The amount of error detail your application displays through the browser is controlled by the `APP_DEBUG` configuration option in your `.env` configuration file.

> **Note:** For local development, you should set the `APP_DEBUG` environment variable to `true`. **In your production environment, this value should always be `false`.**

Handling Errors

All exceptions are handled by the `App\Exceptions\Handler` class. This class contains two methods: `report` and `render`.

The `report` method is used to log exceptions or send them to an external service like BugSnag. By default, the `report` method simply passes the exception to the base implementation on the parent class where the exception is logged. However, you are free to log exceptions however you wish. If you need to report different types of exceptions in different ways, you may use the PHP `instanceof` comparison operator:

```
/**
 * Report or log an exception.
 *
 * This is a great spot to send exceptions to Sentry, Bugsnag, etc.
 *
 * @param  \Exception  $e
 * @return void
 */
public function report(Exception $e)
{
        if ($e instanceof CustomException) {
                //
        }

        return parent::report($e);
}
```

The `render` method is responsible for converting the exception into an HTTP response that should be sent back to the browser. By default, the exception is passed to the base class which generates a response for you. However, you are free to check the exception type or return your own custom response.

The `$dontReport` property of the exception handler contains an array of exception types that will not be logged. By default, exceptions resulting from 404 errors are not written to your log files. You may add other exception types to this array as needed.

HTTP Exceptions

Some exceptions describe HTTP error codes from the server. For example, this may be a "page not found" error (404), an "unauthorized error" (401) or even a developer generated 500 error. In order to return such a response, use the following:

```
abort(404);
```

Optionally, you may provide a response:

```
abort(403, 'Unauthorized action.');
```

This method may be used at any time during the request's lifecycle.

Logging

> **Note:** If you intend to use the `Log` facade, be sure to uncomment the `$app->withFacades()` call in your `bootstrap/app.php` file.

The Lumen and Laravel logging facilities provide a simple layer on top of the powerful Monolog library. By default, Lumen is configured to create daily log files for your application which are stored in the `storage/logs` directory. You may write information to the log like so:

```
Log::info('This is some useful information.');

Log::warning('Something could be going wrong.');

Log::error('Something is really going wrong.');
```

The logger provides the seven logging levels defined in RFC 5424: **debug**, **info**, **notice**, **warning**, **error**, **critical**, and **alert**.

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

```
Log::info('Log message', ['context' => 'Other helpful information']);
```

Resolving The Logger From The Container

If you would like to resolve an instance of the logger from the service container, you may resolve it like so:

```
$logger = app('Psr\Log\LoggerInterface');
```

Of course, you may type-hint this dependency on a route Closure or controller to have the container automatically inject the dependency.

Core Features

Events

- Basic Usage
- Queued Event Handlers
- Event Subscribers

Basic Usage

The Lumen and Laravel event facilities provides a simple observer implementation, allowing you to subscribe and listen for events in your application.

Subscribing To An Event

> **Note:** If you intend to use the `Event` facade, be sure to uncomment the `$app->withFacades()` call in your `bootstrap/app.php` file.

To subscribe to an event, you may use the `Event::listen` method:

```
Event::listen(
        'PodcastWasPurchased', 'EmailPurchaseConfirmation@handle'
);
```

> **Note:** You may place these event registrations in a service provider.

When the event is fired, the event object will be passed to the `handle` method of the listener:

```
class EmailPurchaseConfirmation {

        public function handle(PodcastWasPurchased $event)
        {
                //
        }

}
```

Of course, you are free to place your event and listener classes wherever you want in your application, such as an `app/Events` directory.

Firing An Event

Now we are ready to fire our event using the `Event` facade:

```
$response = Event::fire(new PodcastWasPurchased($podcast));
```

The `fire` method returns an array of responses that you can use to control what happens next in your application.

You may also use the `event` helper to fire an event:

```
event(new PodcastWasPurchased($podcast));
```

Closure Listeners

You can even listen to events without creating a separate handler class at all. For example, in the `register` method of a service provider, you could do the following:

```
Event::listen('App\Events\PodcastWasPurchased', function($event) {
        // Handle the event...
});
```

Stopping The Propagation Of An Event

Sometimes, you may wish to stop the propagation of an event to other listeners. You may do so using by returning `false` from your handler:

```
Event::listen('App\Events\PodcastWasPurchased', function($event) {
        // Handle the event...

        return false;
});
```

Queued Event Handlers

If you would like your event listener to be queued, you may mark it with the `Illuminate\Contracts\Queue\ShouldBeQueued` interface:

```
use Illuminate\Contracts\Queue\ShouldBeQueued;

class SendPurchaseConfirmation implements ShouldBeQueued {

        public function handle(PurchasePodcast $event)
        {
                //
        }

}
```

That's it! Now when this listener is called for an event, it will be queued automatically by the event dispatcher.

> **Note:** Of course, you will need to configure your queue settings before using this feature.

If no exceptions are thrown when the handler is executed by the queue, the queued job will be deleted automatically after it has processed. If you need to access the queued job's `delete` and `release` methods manually, you may do so. The `Illuminate\Queue\InteractsWithQueue` trait, which is included by default on queued handlers, gives you access to these methods:

```
public function handle(PodcastWasPurchased $event)
{
        if (true) {
                $this->release(30);
        }
}
```

Core Features

Helper Functions

- Arrays
- Paths
- Strings
- URLs
- Miscellaneous

Arrays

array_add

The `array_add` function adds a given key / value pair to the array if the given key doesn't already exist in the array.

```
$array = ['foo' => 'bar'];

$array = array_add($array, 'key', 'value');
```

array_divide

The `array_divide` function returns two arrays, one containing the keys, and the other containing the values of the original array.

```
$array = ['foo' => 'bar'];

list($keys, $values) = array_divide($array);
```

array_dot

The `array_dot` function flattens a multi-dimensional array into a single level array that uses "dot" notation to indicate depth.

```
$array = ['foo' => ['bar' => 'baz']];

$array = array_dot($array);

// ['foo.bar' => 'baz'];
```

array_except

The `array_except` method removes the given key / value pairs from the array.

```
$array = array_except($array, ['keys', 'to', 'remove']);
```

array_fetch

The `array_fetch` method returns a flattened array containing the selected nested element.

```
$array = [
        ['developer' => ['name' => 'Taylor']],
        ['developer' => ['name' => 'Dayle']]
];

$array = array_fetch($array, 'developer.name');

// ['Taylor', 'Dayle'];
```

array_first

The `array_first` method returns the first element of an array passing a given truth test.

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

$value = array_first($array, function($key, $value) {
        return $value >= 150;
});
```

A default value may also be passed as the third parameter:

```
$value = array_first($array, $callback, $default);
```

array_last

The `array_last` method returns the last element of an array passing a given truth test.

```
$array = [350, 400, 500, 300, 200, 100];

$value = array_last($array, function($key, $value) {
        return $value > 350;
});

// 500
```

A default value may also be passed as the third parameter:

```
$value = array_last($array, $callback, $default);
```

array_flatten

The `array_flatten` method will flatten a multi-dimensional array into a single level.

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

$array = array_flatten($array);

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

array_forget

The `array_forget` method will remove a given key / value pair from a deeply nested array using "dot" notation.

```
$array = ['names' => ['joe' => ['programmer']]];

array_forget($array, 'names.joe');
```

array_get

The `array_get` method will retrieve a given value from a deeply nested array using "dot" notation.

```
$array = ['names' => ['joe' => ['programmer']]];

$value = array_get($array, 'names.joe');

$value = array_get($array, 'names.john', 'default');
```

Note: Want something like `array_get` but for objects instead? Use `object_get`.

array_only

The `array_only` method will return only the specified key / value pairs from the array.

```
$array = ['name' => 'Joe', 'age' => 27, 'votes' => 1];

$array = array_only($array, ['name', 'votes']);
```

array_pluck

The `array_pluck` method will pluck a list of the given key / value pairs from the array.

```
$array = [['name' => 'Taylor'], ['name' => 'Dayle']];

$array = array_pluck($array, 'name');

// ['Taylor', 'Dayle'];
```

array_pull

The `array_pull` method will return a given key / value pair from the array, as well as remove it.

```
$array = ['name' => 'Taylor', 'age' => 27];

$name = array_pull($array, 'name');
```

array_set

The `array_set` method will set a value within a deeply nested array using "dot" notation.

```
$array = ['names' => ['programmer' => 'Joe']];

array_set($array, 'names.editor', 'Taylor');
```

array_sort

The `array_sort` method sorts the array by the results of the given Closure.

```
$array = [
        ['name' => 'Jill'],
        ['name' => 'Barry']
];

$array = array_values(array_sort($array, function($value) {
        return $value['name'];
}));
```

array_where

Filter the array using the given Closure.

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

$array = array_where($array, function($key, $value) {
        return is_string($value);
});

// Array ( [1] => 200 [3] => 400 )
```

head

Return the first element in the array.

```
$first = head($this->returnsArray('foo'));
```

last

Return the last element in the array. Useful for method chaining.

```
$last = last($this->returnsArray('foo'));
```

Paths

base_path

Get the fully qualified path to the root of the application install.

storage_path

Get the fully qualified path to the `storage` directory.

Strings

camel_case

Convert the given string to `camelCase`.

```
$camel = camel_case('foo_bar');

// fooBar
```

class_basename

Get the class name of the given class, without any namespace names.

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

// Baz
```

e

Run `htmlentities` over the given string, with UTF-8 support.

```
$entities = e('<html>foo</html>');
```

ends_with

Determine if the given haystack ends with a given needle.

```
$value = ends_with('This is my name', 'name');
```

snake_case

Convert the given string to `snake_case`.

```
$snake = snake_case('fooBar');

// foo_bar
```

str_limit

Limit the number of characters in a string.

```
str_limit($value, $limit = 100, $end = '...')
```

Example:

```
$value = str_limit('The PHP framework for web artisans.', 7);

// The PHP...
```

starts_with

Determine if the given haystack begins with the given needle.

```
$value = starts_with('This is my name', 'This');
```

str_contains

Determine if the given haystack contains the given needle.

```
$value = str_contains('This is my name', 'my');
```

str_finish

Add a single instance of the given needle to the haystack. Remove any extra instances.

```
$string = str_finish('this/string', '/');

// this/string/
```

str_is

Determine if a given string matches a given pattern. Asterisks may be used to indicate wildcards.

```
$value = str_is('foo*', 'foobar');
```

str_plural

Convert a string to its plural form (English only).

```
$plural = str_plural('car');
```

str_random

Generate a random string of the given length.

```
$string = str_random(40);
```

str_singular

Convert a string to its singular form (English only).

```
$singular = str_singular('cars');
```

str_slug

Generate a URL friendly "slug" from a given string.

```
str_slug($title, $separator);
```

Example:

```
$title = str_slug("Laravel 5 Framework", "-");

// laravel-5-framework
```

studly_case

Convert the given string to `StudlyCase`.

```
$value = studly_case('foo_bar');

// FooBar
```

trans

Translate a given language line. Alias of `Lang::get`.

```
$value = trans('validation.required'):
```

trans_choice

Translate a given language line with inflection. Alias of `Lang::choice`.

```
$value = trans_choice('foo.bar', $count);
```

URLs

route

Generate a URL for a given named route.

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

url

Generate a fully qualified URL to the given path.

```
echo url('foo/bar', $parameters = [], $secure = null);
```

Miscellaneous

csrf_token

Get the value of the current CSRF token.

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

dd

Dump the given variable and end execution of the script.

```
dd($value);
```

env

Gets the value of an environment variable or return a default value.

```
env('APP_ENV', 'production')
```

event

Fire an event.

```
event('my.event');
```

value

If the given value is a `Closure`, return the value returned by the `Closure`. Otherwise, return the value.

```
$value = value(function() { return 'bar'; });
```

view

Get a View instance for the given view path.

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

Core Features

Queues

- Configuration
- Basic Usage
- More Dispatch Methods
- Queueing Closures
- Running The Queue Listener
- Daemon Queue Worker
- Failed Jobs

Configuration

Lumen utilizes Laravel's queue component to provide a unified API across a variety of different queue services. Queues allow you to defer the processing of a time consuming task, such as sending an e-mail, until a later time, thus drastically speeding up the web requests to your application.

Lumen and Laravel provide support for database, Beanstalkd, IronMQ, Amazon SQS, Redis, null, and synchronous (for local use) queue drivers. The `null` queue driver simply discards queued jobs so they are never run.

The `QUEUE_DRIVER` option in your `.env` file determines the queue "driver" that will be used by your application.

Queue Database Table

In order to use the `database` queue driver, you will need a database table to hold the jobs. The table schema should look like the following:

```
Schema::create('jobs', function(Blueprint $table)
{
        $table->bigIncrements('id');
        $table->string('queue');
        $table->text('payload');
        $table->tinyInteger('attempts')->unsigned();
        $table->tinyInteger('reserved')->unsigned();
        $table->unsignedInteger('reserved_at')->nullable();
        $table->unsignedInteger('available_at');
        $table->unsignedInteger('created_at');
});
```

Other Queue Dependencies

The following dependencies are needed for the listed queue drivers:

- Amazon SQS: `aws/aws-sdk-php`
- Beanstalkd: `pda/pheanstalk ~3.0`
- IronMQ: `iron-io/iron_mq ~1.5`
- Redis: `predis/predis ~1.0`

Basic Usage

Pushing A Job Onto The Queue

All of the queueable jobs for your application are stored in the `App\Jobs` directory. The base `App\Job` class may serve as a base class for the rest of your jobs.

> **Note:** If you intend to use the `Queue` facade, be sure to uncomment the `$app->withFacades()` call in your `bootstrap/app.php` file.

To push a new job onto the queue, use the `Queue::push` method:

```
Queue::push(new SendEmail($message));
```

You may also use the `dispatch` method from a Closure route or a controller:

```
$this->dispatch(new SendEmail($message));
```

The job's `handle` method will be called when the job is executed by the queue. You may type-hint any dependencies you need on the `handle` method and the service container will automatically inject them:

```
public function handle(UserRepository $users)
```

```
{
        //
}
```

Specifying The Queue / Tube For A Job

You may also specify the queue / tube a job should be sent to:

```
Queue::pushOn('emails', new SendEmail($message));
```

Passing The Same Payload To Multiple Jobs

If you need to pass the same data to several queue jobs, you may use the `Queue::bulk` method:

```
Queue::bulk([new SendEmail($message), new AnotherCommand]);
```

Delaying The Execution Of A Job

Sometimes you may wish to delay the execution of a queued job. For instance, you may wish to queue a job that sends a customer an e-mail 15 minutes after sign-up. You can accomplish this using the `Queue::later` method:

```
$date = Carbon::now()->addMinutes(15);

Queue::later($date, new SendEmail($message));
```

In this example, we're using the Carbon date library to specify the delay we wish to assign to the job. Alternatively, you may pass the number of seconds you wish to delay as an integer.

> **Note:** The Amazon SQS service has a delay limit of 900 seconds (15 minutes).

Queues And Eloquent Models

If your queued job accepts an Eloquent model in its constructor, only the identifier for the model will be serialized onto the queue. When the job is actually handled, the queue system will automatically re-retrieve the full model instance from the database. It's all totally transparent to your application and prevents issues that can arise from serializing full Eloquent model instances.

Deleting A Processed Job

Once you have processed a job, it must be deleted from the queue. If no exception is thrown during the execution of your job, this will be done automatically.

If you would like to `delete` or `release` the job manually, the `Illuminate\Queue\InteractsWithQueue` trait provides access to the queue job `release` and `delete` methods. The `release` method accepts a single value: the number of seconds you wish to wait until the job is made available again.

```
public function handle(SendEmail $job)
{
        if (true) {
                $this->release(30);
        }
}
```

Releasing A Job Back Onto The Queue

IF an exception is thrown while the job is being processed, it will automatically be released back onto the queue so it may be attempted again. The job will continue to be released until it has been attempted the maximum number of times allowed by your application. The number of maximum attempts is defined by the `--tries` switch used on the `queue:listen` or `queue:work` Artisan jobs.

Checking The Number Of Run Attempts

If an exception occurs while the job is being processed, it will automatically be released back onto the queue. You may check the number of attempts that have been made to run the job using the `attempts` method:

```
if ($this->attempts() > 3) {
        //
}
```

> **Note:** Your job / handler must use the `Illuminate\Queue\InteractsWithQueue` trait in order to call this method.

More Dispatch Methods

Mapping Command Properties From Requests

It is very common to map HTTP request variables into jobs. So, instead of forcing you to do this manually for each request, Lumen provides some helper methods to make it a cinch. Let's take a look at the `dispatchFrom` method available from Closure routes and controller methods:

```
$this->dispatchFrom('Command\Class\Name', $request);
```

This method will examine the constructor of the job class it is given, and then extract variables from the HTTP request (or any other `ArrayAccess` object) to fill the needed constructor parameters of the job. So, if our job class accepts a `firstName` variable in its constructor, the job bus will attempt to pull the `firstName` parameter from the HTTP request.

You may also pass an array as the third argument to the `dispatchFrom` method. This array will be used to fill any constructor parameters that are not available on the request:

```
$this->dispatchFrom('Command\Class\Name', $request, [
        'firstName' => 'Taylor',
]);
```

Queueing Closures

> **Note:** Before queueing Closures, you will need to add the `jeremeamia/superclosure` (~2.0) dependency to your `composer.json` file.

You may also push a Closure onto the queue. This is very convenient for quick, simple tasks that need to be queued:

Pushing A Closure Onto The Queue

```
Queue::push(function($job) use ($id) {
        Account::delete($id);

        $job->delete();
});
```

> **Note:** Instead of making objects available to queued Closures via the `use` directive, consider passing primary keys and re-pulling the associated models from within your queue job. This often avoids unexpected serialization behavior.

When using Iron.io push queues, you should take extra precaution queueing Closures. The end-point that receives your queue messages should check for a token to verify that the request is actually from Iron.io. For example, your push queue end-point should be something like: `https://yourapp.com/queue/receive?token=SecretToken`. You may then check the value of the secret token in your application before marshalling the queue request.

Running The Queue Listener

Lumen, like Laravel, includes an Artisan task that will run new jobs as they are pushed onto the queue. You may run this task using the `queue:listen` job:

Starting The Queue Listener

```
php artisan queue:listen
```

You may also specify which queue connection the listener should utilize:

```
php artisan queue:listen connection
```

Note that once this task has started, it will continue to run until it is manually stopped. You may use a process monitor such as Supervisor to ensure that the queue listener does not stop running.

You may pass a comma-delimited list of queue connections to the `listen` job to set queue priorities:

```
php artisan queue:listen --queue=high,low
```

In this example, jobs on the `high-connection` will always be processed before moving onto jobs from the `low-connection`.

Specifying The Job Timeout Parameter

You may also set the length of time (in seconds) each job should be allowed to run:

```
php artisan queue:listen --timeout=60
```

Specifying Queue Sleep Duration

In addition, you may specify the number of seconds to wait before polling for new jobs:

```
php artisan queue:listen --sleep=5
```

Note that the queue only "sleeps" if no jobs are on the queue. If more jobs are available, the queue will continue to work them without sleeping.

Processing The First Job On The Queue

To process only the first job on the queue, you may use the `queue:work` job:

```
php artisan queue:work
```

Daemon Queue Worker

The `queue:work` also includes a `--daemon` option for forcing the queue worker to continue processing jobs without ever re-booting the framework. This results in a significant reduction of CPU usage when compared to the `queue:listen` job.

To start a queue worker in daemon mode, use the `--daemon` flag:

```
php artisan queue:work connection --daemon

php artisan queue:work connection --daemon --sleep=3

php artisan queue:work connection --daemon --sleep=3 --tries=3
```

As you can see, the `queue:work` job supports most of the same options available to `queue:listen`. You may use the `php artisan help queue:work` job to view all of the available options.

Deploying With Daemon Queue Workers

The simplest way to deploy an application using daemon queue workers is to put the application in maintenance mode at the beginning of your deployment. This can be done using the `php artisan down` job. Once the application is in maintenance mode, Lumen and Laravel will not accept any new jobs off of the queue, but will continue to process existing jobs.

The easiest way to restart your workers is to include the following job in your deployment script:

```
php artisan queue:restart
```

This job will instruct all queue workers to restart after they finish processing their current job.

> **Note:** This job relies on the cache system to schedule the restart. By default, APCu does not work for CLI jobs. If you are using APCu, add `apc.enable_cli=1` to your APCu configuration.

Coding For Daemon Queue Workers

Daemon queue workers do not restart the framework before processing each job. Therefore, you should be careful to free any heavy resources before your job finishes. For example, if you are doing image manipulation with the GD library, you should free the memory with `imagedestroy` when you are done.

Similarly, your database connection may disconnect when being used by long-running daemon. You may use the `DB::reconnect` method to ensure you have a fresh connection.

Failed Jobs

Since things don't always go as planned, sometimes your queued jobs will fail. Don't worry, it happens to the best of us! Lumen and Laravel include a convenient way to specify the maximum number of times a job should be attempted. After a job has exceeded this amount of attempts, it will be inserted into a `failed_jobs` table.

The `failed_jobs` table should have a schema like the following:

```
Schema::create('failed_jobs', function(Blueprint $table)
{
        $table->increments('id');
        $table->text('connection');
        $table->text('queue');
        $table->text('payload');
        $table->timestamp('failed_at');
});
```

You can specify the maximum number of times a job should be attempted using the `--tries` switch on the `queue:listen` job:

```
php artisan queue:listen connection-name --tries=3
```

If you would like to register an event that will be called when a queue job fails, you may use the `Queue::failing` method. This event is a great opportunity to notify your team via e-mail or HipChat.

```
Queue::failing(function($connection, $job, $data) {
        //
});
```

You may also define a `failed` method directly on a queue job class, allowing you to perform job specific actions when a failure occurs:

```
public function failed()
{
        // Called when the job is failing...
}
```

Retrying Failed Jobs

To view all of your failed jobs, you may use the `queue:failed` Artisan job:

```
php artisan queue:failed
```

The `queue:failed` job will list the job ID, connection, queue, and failure time. The job ID may be used to retry the failed job. For instance, to retry a failed job that has an ID of 5, the following job should be issued:

```
php artisan queue:retry 5
```

If you would like to delete a failed job, you may use the `queue:forget` job:

```
php artisan queue:forget 5
```

To delete all of your failed jobs, you may use the `queue:flush` job:

```
php artisan queue:flush
```

Core Features

Testing

- Introduction
- Defining & Running Tests
- Test Environment
- Calling Routes From Tests
- Mocking Facades
- Framework Assertions
- Helper Methods
- Refreshing The Application

Introduction

Lumen, like Laravel, is built with unit testing in mind. In fact, support for testing with PHPUnit is included out of the box, and a `phpunit.xml` file is already setup for your application.

An example test file is provided in the `tests` directory. After installing a new Lumen application, simply run `phpunit` on the command line to run your tests.

Defining & Running Tests

To create a test case, simply create a new test file in the `tests` directory. The test class should extend `TestCase`. You may then define test methods as you normally would when using PHPUnit.

An Example Test Class

```
class FooTest extends TestCase {

        public function testSomethingIsTrue()
        {
                $this->assertTrue(true);
        }

}
```

You may run all of the tests for your application by executing the `phpunit` command from your terminal.

Note: If you define your own `setUp` method, be sure to call `parent::setUp`.

Test Environment

Lumen automatically sets the cache and session drivers to `array` while in the test environment, meaning no session or cache data will be persisted while testing. You are free to create other testing environment configurations as necessary.

The `testing` environment variables may be configured in the `phpunit.xml` file.

Calling Routes From Tests

Calling A Route From A Test

You may easily call one of your routes for a test using the `call` method:

```
$response = $this->call('GET', '/user/profile');

$response = $this->call(
        $method, $uri, $parameters, $cookies, $files, $server, $content
);
```

You may then inspect the `Illuminate\Http\Response` object:

```
$this->assertEquals('Hello World', $response->getContent());
```

The `getContent` method will return the evaluated string contents of the response. If your route returns a `View`, you may access it using the `original` property:

```
$view = $response->original;

$this->assertEquals('John', $view['name']);
```

To call a HTTPS route, you may use the `callSecure` method:

```
$response = $this->callSecure('GET', '/foo/bar');
```

Mocking Facades

When testing, you may often want to mock a call to a static facade. For example, consider the following controller action:

```
public function getIndex()
{
        Event::fire('foo', ['name' => 'Dayle']);

        return 'All done!';
}
```

We can mock the call to the `Event` class by using the `shouldReceive` method on the facade, which will return an instance of a Mockery mock.

> **Note:** Lumen does not install Mockery by default. It can be installed with `composer require mockery/mockery --dev`

Mocking A Facade

```
public function testGetIndex()
{
        Event::shouldReceive('fire')->once()->with('foo', ['name' => 'Dayle']);

        $this->call('GET', '/');
}
```

> **Note:** You should not mock the `Request` facade. Instead, pass the input you desire into the `call` method when running your test.

Framework Assertions

Lumen, like Laravel, ships with several `assert` methods to make testing a little easier:

Asserting Responses Are OK

```
public function testMethod()
{
        $this->call('GET', '/');

        $this->assertResponseOk();
}
```

Asserting Response Statuses

```
$this->assertResponseStatus(403);
```

Asserting Responses Are Redirects

```
$this->assertRedirectedTo('foo');

$this->assertRedirectedToRoute('route.name');

$this->assertRedirectedToAction('Controller@method');
```

Asserting A View Has Some Data

```
public function testMethod()
{
        $this->call('GET', '/');

        $this->assertViewHas('name');
        $this->assertViewHas('age', $value);
}
```

Asserting The Session Has Some Data

```
public function testMethod()
{
        $this->call('GET', '/');

        $this->assertSessionHas('name');
        $this->assertSessionHas('age', $value);
```

```
}
```

Asserting The Session Has Errors

```
public function testMethod()
{
        $this->call('GET', '/');

        $this->assertSessionHasErrors();

        // Asserting the session has errors for a given key...
        $this->assertSessionHasErrors('name');

        // Asserting the session has errors for several keys...
```

$this->assertSessionHasErrors(['name', 'age']);

```
}
```

Asserting Old Input Has Some Data

```
public function testMethod()
{
        $this->call('GET', '/');

        $this->assertHasOldInput();
}
```

Helper Methods

The `TestCase` class contains several helper methods to make testing your application easier.

Setting And Flushing Sessions From Tests

```
$this->session(['foo' => 'bar']);

$this->flushSession();
```

Setting The Currently Authenticated User

You may set the currently authenticated user using the `be` method:

```
$user = new User(['name' => 'John']);

$this->be($user);
```

You may re-seed your database from a test using the `seed` method:

Re-Seeding Database From Tests

```
$this->seed();

$this->seed('DatabaseSeeder');
```

Refreshing The Application

As you may already know, you can access your Application (service container) via `$this->app` from any test method. This service container instance is refreshed for each test class. If you wish to manually force the Application to be refreshed for a given method, you may use the `refreshApplication` method from your test method. This will reset any extra bindings, such as mocks, that have been placed in the service container since the test case started running.

Core Features

Validation

- Basic Usage
- Route / Controller Validation
- Working With Error Messages
- Error Messages & Views
- Available Validation Rules
- Conditionally Adding Rules
- Custom Error Messages
- Custom Validation Rules

Basic Usage

Lumen, like Laravel, ships with a simple, convenient facility for validating data and retrieving validation error messages via the `Validation` facade.

Basic Validation Example

```
$validator = Validator::make(
        ['name' => 'Dayle'],
        ['name' => 'required|min:5']
);
```

The first argument passed to the `make` method is the data under validation. The second argument is the validation rules that should be applied to the data.

Using Arrays To Specify Rules

Multiple rules may be delimited using either a "pipe" character, or as separate elements of an array.

```
$validator = Validator::make(
        ['name' => 'Dayle'],
        ['name' => ['required', 'min:5']]
);
```

Validating Multiple Fields

```
$validator = Validator::make(
        [
                'name' => 'Dayle',
                'password' => 'lamepassword',
                'email' => 'email@example.com'
        ],
        [
                'name' => 'required',
                'password' => 'required|min:8',
                'email' => 'required|email|unique:users'
        ]
);
```

Once a `Validator` instance has been created, the `fails` (or `passes`) method may be used to perform the validation.

```
if ($validator->fails())
{
        // The given data did not pass validation
}
```

If validation has failed, you may retrieve the error messages from the validator.

```
$messages = $validator->messages();
```

You may also access an array of the failed validation rules, without messages. To do so, use the `failed` method:

```
$failed = $validator->failed();
```

Validating Files

The `Validator` class provides several rules for validating files, such as `size`, `mimes`, and others. When validating files, you may simply pass them into the validator with your other data.

After Validation Hook

The validator also allows you to attach callbacks to be run after validation is completed. This allows you to easily perform further validation, and even add more error messages to the message collection. To get started, use the `after` method on a validator instance:

```
$validator = Validator::make(...);

$validator->after(function($validator) {
        if ($this->somethingElseIsInvalid()) {
                $validator->errors()->add('field', 'Something is wrong with this field!');
        }
});

if ($validator->fails()) {
        //
}
```

You may add as many `after` callbacks to a validator as needed.

Route / Controller Validation

Of course, manually creating and checking a `Validator` instance each time you do validation is a headache. Don't worry, you have other options! The base `Laravel\Lumen\Routing\Controller` class included with Lumen uses a `ValidatesRequests` trait. This trait provides a single, convenient method for validating incoming HTTP requests. Here's what it looks like:

```
/**
 * Store the incoming blog post.
 *
 * @param  Request  $request
 * @return Response
 */
public function store(Request $request)
{
        $this->validate($request, [
                'title' => 'required|unique|max:255',
                'body' => 'required',
        ]);

        //
}
```

You may even call the `validate` method from a route Closure:

```
use Illuminate\Http\Request;

$app->post('comment', function(Request $request) {

        $this->validate($request, [
                'title' => 'required|unique|max:255',
                'body' => 'required',
        ]);

        //
});
```

If validation passes, your code will keep executing normally. However, if validation fails, an `Illuminate\Contracts\Validation\ValidationException` will be thrown. This exception is automatically caught and a redirect is generated to the user's previous location. The validation errors are even automatically flashed to the session!

If the incoming request was an AJAX request, no redirect will be generated. Instead, an HTTP response with a 422 status code will be returned to the browser containing a JSON representation of the validation errors.

For example, here is the equivalent code written manually:

```
/**
 * Store the incoming blog post.
 *
 * @param  Request  $request
 * @return Response
 */
public function store(Request $request)
{
        $v = Validator::make($request->all(), [
                'title' => 'required|unique|max:255',
                'body' => 'required',
        ]);

        if ($v->fails()) {
                return redirect()->back()->withErrors($v->errors());
        }

        //
}
```

Customizing The Flashed Error Format

If you wish to customize the format of the validation errors that are flashed to the session when validation fails, override the `formatValidationErrors` on your base controller. Don't forget to import the `Illuminate\Validation\Validator` class at the top of the file:

```
/**
 * {@inheritdoc}
 */
protected function formatValidationErrors(Validator $validator)
{
        return $validator->errors()->all();
}
```

If you would like to customize the format of the validation errors when using the `validate` method from route Closures, you may do so by calling the `Laravel\Lumen\Routing\Closure` class:

```
use Laravel\Lumen\Routing\Closure;

Closure::formatErrorsUsing(function($validator) {
        return $validator->errors()->all();
});
```

Likewise, you may customize how the entire HTTP response for route Closure validation errors is rendered:

```
use Laravel\Lumen\Routing\Closure;

Closure::buildResponseUsing(function($validator, $errors) {
        // Return Illuminate\Http\Response Instance...
});
```

Working With Error Messages

After calling the `messages` method on a `Validator` instance, you will receive a `MessageBag` instance, which has a variety of convenient methods for working with error messages.

Retrieving The First Error Message For A Field

```
echo $messages->first('email');
```

Retrieving All Error Messages For A Field

```
foreach ($messages->get('email') as $message) {
        //
}
```

Retrieving All Error Messages For All Fields

```
foreach ($messages->all() as $message) {
        //
}
```

Determining If Messages Exist For A Field

```
if ($messages->has('email')) {
        //
}
```

Retrieving An Error Message With A Format

```
echo $messages->first('email', '<p>:message</p>');
```

Retrieving All Error Messages With A Format

```
foreach ($messages->all('<li>:message</li>') as $message) {
        //
}
```

Error Messages & Views

Note: Before using this feature of Lumen, you will need to enable sessions.

Once you have performed validation, you will need an easy way to get the error messages back to your views. This is conveniently handled by Lumen. Consider the following routes as an example:

```
$app->get('register', function() {
        return view('user.register');
});

$app->post('register', function() {
        $rules = [...];

        $validator = Validator::make(Input::all(), $rules);

        if ($validator->fails()) {
                return redirect('register')->withErrors($validator);
        }
});
```

Note that when validation fails, we pass the `Validator` instance to the Redirect using the `withErrors` method. This method will flash the error messages to the session so that they are available on the next request.

However, notice that we do not have to explicitly bind the error messages to the view in our GET route. This is because Laravel will always check for errors in the session data, and automatically bind them to the view if they are available. **So, it is important to note that an `$errors` variable will always be available in all of your views, on every request**, allowing you to conveniently assume the `$errors` variable is always defined and can be safely used. The `$errors` variable will be an instance of `MessageBag`.

So, after redirection, you may utilize the automatically bound `$errors` variable in your view:

```
<?php echo $errors->first('email'); ?>
```

Named Error Bags

If you have multiple forms on a single page, you may wish to name the `MessageBag` of errors. This will allow you to retrieve the error messages for a specific form. Simply pass a name as the second argument to `withErrors`:

```
return redirect('register')->withErrors($validator, 'login');
```

You may then access the named `MessageBag` instance from the `$errors` variable:

```
<?php echo $errors->login->first('email'); ?>
```

Available Validation Rules

Below is a list of all available validation rules and their function:

- Accepted
- Active URL
- After (Date)
- Alpha
- Alpha Dash
- Alpha Numeric
- Array
- Before (Date)
- Between
- Boolean
- Confirmed
- Date
- Date Format
- Different
- Digits
- Digits Between
- E-Mail
- Exists (Database)
- Image (File)
- In
- Integer
- IP Address
- Max
- MIME Types
- Min
- Not In
- Numeric
- Regular Expression
- Required
- Required If
- Required With
- Required With All

- Required Without
- Required Without All
- Same
- Size
- String
- Timezone
- Unique (Database)
- URL

accepted

The field under validation must be *yes*, *on*, *1*, or *true*. This is useful for validating "Terms of Service" acceptance.

active_url

The field under validation must be a valid URL according to the `checkdnsrr` PHP function.

after:*date*

The field under validation must be a value after a given date. The dates will be passed into the PHP `strtotime` function.

alpha

The field under validation must be entirely alphabetic characters.

alpha_dash

The field under validation may have alpha-numeric characters, as well as dashes and underscores.

alpha_num

The field under validation must be entirely alpha-numeric characters.

array

The field under validation must be of type array.

before:*date*

The field under validation must be a value preceding the given date. The dates will be passed into the PHP `strtotime` function.

between:*min*,*max*

The field under validation must have a size between the given *min* and *max*. Strings, numerics, and files are evaluated in the same fashion as the `size` rule.

boolean

The field under validation must be able to be cast as a boolean. Accepted input are `true`, `false`, `1`, `0`, `"1"` and `"0"`.

confirmed

The field under validation must have a matching field of `foo_confirmation`. For example, if the field under validation is `password`, a matching `password_confirmation` field must be present in the input.

date

The field under validation must be a valid date according to the `strtotime` PHP function.

date_format:*format*

The field under validation must match the *format* defined according to the `date_parse_from_format` PHP function.

different:*field*

The given *field* must be different than the field under validation.

digits:*value*

The field under validation must be *numeric* and must have an exact length of *value*.

digits_between:*min,max*

The field under validation must have a length between the given *min* and *max*.

email

The field under validation must be formatted as an e-mail address.

exists:*table,column*

The field under validation must exist on a given database table.

Basic Usage Of Exists Rule

```
'state' => 'exists:states'
```

Specifying A Custom Column Name

```
'state' => 'exists:states,abbreviation'
```

You may also specify more conditions that will be added as "where" clauses to the query:

```
'email' => 'exists:staff,email,account_id,1'
```

Passing `NULL` as a "where" clause value will add a check for a `NULL` database value:

```
'email' => 'exists:staff,email,deleted_at,NULL'
```

image

The file under validation must be an image (jpeg, png, bmp, gif, or svg)

in:*foo,bar,...*

The field under validation must be included in the given list of values.

integer

The field under validation must have an integer value.

ip

The field under validation must be formatted as an IP address.

max:*value*

The field under validation must be less than or equal to a maximum *value*. Strings, numerics, and files are evaluated in the same fashion as the `size` rule.

mimes:*foo,bar,...*

The file under validation must have a MIME type corresponding to one of the listed extensions.

Basic Usage Of MIME Rule

```
'photo' => 'mimes:jpeg,bmp,png'
```

min:*value*

The field under validation must have a minimum *value*. Strings, numerics, and files are evaluated in the same fashion as the `size` rule.

not_in:*foo,bar,...*

The field under validation must not be included in the given list of values.

numeric

The field under validation must have a numeric value.

regex:*pattern*

The field under validation must match the given regular expression.

Note: When using the `regex` pattern, it may be necessary to specify rules in an array instead of using pipe delimiters, especially if the regular expression contains a pipe character.

required

The field under validation must be present in the input data.

required_if:*field,value,...*

The field under validation must be present if the *field* field is equal to any *value*.

required_with:*foo,bar,...*

The field under validation must be present *only if* any of the other specified fields are present.

required_with_all:*foo,bar,...*

The field under validation must be present *only if* all of the other specified fields are present.

required_without:*foo,bar,...*

The field under validation must be present *only when* any of the other specified fields are not present.

required_without_all:*foo,bar,...*

The field under validation must be present *only when* all of the other specified fields are not present.

same:*field*

The given *field* must match the field under validation.

size:*value*

The field under validation must have a size matching the given *value*. For string data, *value* corresponds to the number of characters. For numeric data, *value* corresponds to a given integer value. For files, *size* corresponds to the file size in kilobytes.

string:*value*

The field under validation must be a string type.

timezone

The field under validation must be a valid timezone identifier according to the `timezone_identifiers_list` PHP function.

unique:*table,column,except,idColumn*

The field under validation must be unique on a given database table. If the `column` option is not specified, the field name will be used.

Basic Usage Of Unique Rule

```
'email' => 'unique:users'
```

Specifying A Custom Column Name

```
'email' => 'unique:users,email_address'
```

Forcing A Unique Rule To Ignore A Given ID

```
'email' => 'unique:users,email_address,10'
```

Adding Additional Where Clauses

You may also specify more conditions that will be added as "where" clauses to the query:

```
'email' => 'unique:users,email_address,NULL,id,account_id,1'
```

In the rule above, only rows with an `account_id` of `1` would be included in the unique check.

url

The field under validation must be formatted as an URL.

> **Note:** This function uses PHP's `filter_var` method.

Conditionally Adding Rules

In some situations, you may wish to run validation checks against a field **only** if that field is present in the input array. To quickly accomplish this, add the `sometimes` rule to your rule list:

```
$v = Validator::make($data, [
        'email' => 'sometimes|required|email',
]);
```

In the example above, the `email` field will only be validated if it is present in the `$data` array.

Complex Conditional Validation

Sometimes you may wish to require a given field only if another field has a greater value than 100. Or you may need two fields to have a given value only when another field is present. Adding these validation rules doesn't have to be a pain. First, create a `Validator` instance with your *static rules* that never change:

```
$v = Validator::make($data, [
        'email' => 'required|email',
        'games' => 'required|numeric',
]);
```

Let's assume our web application is for game collectors. If a game collector registers with our application and they own more than 100 games, we want them to explain why they own so many games. For example, perhaps they run a game re-sell shop, or maybe they just enjoy collecting. To conditionally add this requirement, we can use the `sometimes` method on the `Validator` instance.

```
$v->sometimes('reason', 'required|max:500', function($input) {
        return $input->games >= 100;
});
```

The first argument passed to the `sometimes` method is the name of the field we are conditionally validating. The second argument is the rules we want to add. If the `Closure` passed as the third argument returns `true`, the rules will be added. This method makes it a breeze to build complex conditional validations. You may even add conditional validations for several fields at once:

```
$v->sometimes(['reason', 'cost'], 'required', function($input) {
        return $input->games >= 100;
});
```

> **Note:** The `$input` parameter passed to your `Closure` will be an instance of `Illuminate\Support\Fluent` and may be used as an object to access your input and files.

Custom Error Messages

If needed, you may use custom error messages for validation instead of the defaults. There are several ways to specify custom messages.

Passing Custom Messages Into Validator

```
$messages = [
        'required' => 'The :attribute field is required.',
```

```
];

$validator = Validator::make($input, $rules, $messages);
```

Note: The `:attribute` place-holder will be replaced by the actual name of the field under validation. You may also utilize other place-holders in validation messages.

Other Validation Place-Holders

```
$messages = [
        'same'    => 'The :attribute and :other must match.',
        'size'    => 'The :attribute must be exactly :size.',
        'between' => 'The :attribute must be between :min - :max.',
        'in'      => 'The :attribute must be one of the following types: :values',
];
```

Specifying A Custom Message For A Given Attribute

Sometimes you may wish to specify a custom error messages only for a specific field:

```
$messages = [
        'email.required' => 'We need to know your e-mail address!',
];
```

Specifying Custom Messages In Language Files

In some cases, you may wish to specify your custom messages in a language file instead of passing them directly to the `Validator`. To do so, add your messages to `custom` array in the `resources/lang/xx/validation.php` language file.

```
'custom' => [
        'email' => [
                'required' => 'We need to know your e-mail address!',
        ],
],
```

Custom Validation Rules

Registering A Custom Validation Rule

Lumen provides a variety of helpful validation rules; however, you may wish to specify some of your own. One method of registering custom validation rules is using the `Validator::extend` method:

```
Validator::extend('foo', function($attribute, $value, $parameters) {
        return $value == 'foo';
});
```

Note: Validator extensions should be placed in service providers.

The custom validator Closure receives three arguments: the name of the `$attribute` being validated, the `$value` of the attribute, and an array of `$parameters` passed to the rule.

You may also pass a class and method to the `extend` method instead of a Closure:

```
Validator::extend('foo', 'FooValidator@validate');
```

Note that you will also need to define an error message for your custom rules. You can do so either using an inline custom message array or by adding an entry in the validation language file.

Extending The Validator Class

Instead of using Closure callbacks to extend the Validator, you may also extend the Validator class itself. To do so, write a Validator class that extends `Illuminate\Validation\Validator`. You may add validation methods to the class by prefixing them with `validate`:

```
<?php

class CustomValidator extends Illuminate\Validation\Validator {

        public function validateFoo($attribute, $value, $parameters)
        {
                return $value == 'foo';
        }

}
```

Registering A Custom Validator Resolver

Next, you need to register your custom Validator extension:

```
Validator::resolver(function($translator, $data, $rules, $messages) {
        return new CustomValidator($translator, $data, $rules, $messages);
});
```

When creating a custom validation rule, you may sometimes need to define custom place-holder replacements for error messages. You may do so by creating a custom Validator as described above, and adding a `replaceXXX` function to the validator.

```
protected function replaceFoo($message, $attribute, $rule, $parameters)
{
        return str_replace(':foo', $parameters[0], $message);
}
```

If you would like to add a custom message "replacer" without extending the `Validator` class, you may use the `Validator::replacer` method:

```
Validator::replacer('rule', function($message, $attribute, $rule, $parameters) {
        //
});
```

Full-Stack Features

Authentication

- Introduction
- Configuration
- Basic Usage

Introduction

Lumen is primarily designed for building fast micro-services and APIs; however, if you wish, you may use Laravel's authentication system to authenticate users of your Lumen application.

Configuration

> **Note:** Using the authentication system will require enabling sessions. You can do so by uncommenting the middleware listed in the default call to `$app->middleware` in your `bootstrap/app.php` file.

The authentication system has several configuration options you can set in your `.env` file:

- `AUTH_DRIVER`
- `AUTH_MODEL`
- `AUTH_TABLE`

The `AUTH_DRIVER` value specifies the authentication driver used by the framework. If `eloquent` is specified as the driver, the Eloquent ORM driver will be utilized, while `database` will specify that the plain "database" driver should be used.

The `AUTH_MODEL` option specifies the name of the Eloquent model to be used for authentication. This model must implement the `Illuminate\Contracts\Auth\Authenticatable` contract. For an example model, check out the `App\User` model included in the full-stack Laravel framework.

The `AUTH_TABLE` option specifies which database table contains the "users" of your application. Of course, this option only applies when using the `database` authentication driver.

Basic Usage

Unlike Laravel, Lumen does not include any scaffolding for authentication, so you will need to use the authentication libraries manually.

> **Note:** If you intend to use the `Auth` facade, be sure to uncomment the `$app->withFacades()` call in your `bootstrap/app.php` file.

First, let's check out the `attempt` method:

```
use Illuminate\Http\Request;

$app->post('auth/login', function(Request $request) {

        if (Auth::attempt($request->only('email', 'password'))) {
                return redirect('dashboard');
        }

});
```

The `attempt` method accepts an array of key / value pairs as its first argument. The `password` value will be hashed. The other values in the array will be used to find the user in your database table. So, in the example above, the user will be retrieved by the value of the `email` column. If the user is found, the hashed password stored in the database will be compared with the hashed `password` value passed to the method via the array. If the two hashed passwords match, a new authenticated session will be started for the user.

The `attempt` method will return `true` if authentication was successful. Otherwise, `false` will be returned.

> **Note:** In this example, `email` is not a required option, it is merely used as an example. You should use whatever column name corresponds to a "username" in your database.

Authenticating A User With Conditions

You also may add extra conditions to the authentication query:

```
if (Auth::attempt(['email' => $email, 'password' => $password, 'active' => 1])) {
```

```
        // The user is active, not suspended, and exists.
}
```

Determining If A User Is Authenticated

To determine if the user is already logged into your application, you may use the `check` method:

```
if (Auth::check()) {
        // The user is logged in...
}
```

Authenticating A User And "Remembering" Them

If you would like to provide "remember me" functionality in your application, you may pass a boolean value as the second argument to the `attempt` method, which will keep the user authenticated indefinitely, or until they manually logout. Of course, your `users` table must include the string `remember_token` column, which will be used to store the "remember me" token.

```
if (Auth::attempt(['email' => $email, 'password' => $password], $remember)) {
        // The user is being remembered...
}
```

If you are "remembering" users, you may use the `viaRemember` method to determine if the user was authenticated using the "remember me" cookie:

```
if (Auth::viaRemember()) {
        //
}
```

Authenticating Users By ID

To log a user into the application by their ID, use the `loginUsingId` method:

```
Auth::loginUsingId(1);
```

Validating User Credentials Without Login

The `validate` method allows you to validate a user's credentials without actually logging them into the application:

```
if (Auth::validate($credentials)) {
        //
}
```

Logging A User In For A Single Request

You may also use the `once` method to log a user into the application for a single request. No sessions or cookies will be utilized:

```
if (Auth::once($credentials)) {
        //
}
```

Manually Logging In A User

If you need to log an existing user instance into your application, you may call the `login` method with the user instance:

```
Auth::login($user);
```

This is equivalent to logging in a user via credentials using the `attempt` method.

Logging A User Out Of The Application

```
Auth::logout();
```

Full-Stack Features

Filesystem / Cloud Storage

- Introduction
- Configuration
- Basic Usage

Introduction

Lumen provides a wonderful filesystem abstraction thanks to the Flysystem PHP package by Frank de Jonge. The Flysystem integration provides simple to use drivers for working with local filesystems, Amazon S3, and Rackspace Cloud Storage. Even better, it's amazingly simple to switch between these storage options as the API remains the same for each system!

Configuration

The filesystem configuration options are located in your `.env` configuration file. You may look at the `.env.example` configuration file for an example of using these options.

Before using the S3 or Rackspace drivers, you will need to install the appropriate package via Composer:

- Amazon S3: `league/flysystem-aws-s3-v2 ~1.0`
- Rackspace: `league/flysystem-rackspace ~1.0`

When using the `local` driver, note that all file operations are relative to the `storage/app` directory. Therefore, the following method would store a file in `storage/app/file.txt`:

```
Storage::disk('local')->put('file.txt', 'Contents');
```

Basic Usage

> **Note:** If you intend to use the `Storage` facade, be sure to uncomment the `$app->withFacades()` call in your `bootstrap/app.php` file.

The `Storage` facade may be used to interact with any of your configured disks. Alternatively, you may type-hint the `Illuminate\Contracts\Filesystem\Factory` contract on any class that is resolved via the Laravel service container.

Retrieving A Particular Disk

```
$disk = Storage::disk('s3');

$disk = Storage::disk('local');
```

Determining If A File Exists

```
$exists = Storage::disk('s3')->exists('file.jpg');
```

Calling Methods On The Default Disk

```
if (Storage::exists('file.jpg')) {
        //
}
```

Retrieving A File's Contents

```
$contents = Storage::get('file.jpg');
```

Setting A File's Contents

```
Storage::put('file.jpg', $contents);
```

Prepend To A File

```
Storage::prepend('file.log', 'Prepended Text');
```

Append To A File

```
Storage::append('file.log', 'Appended Text');
```

Delete A File

```
Storage::delete('file.jpg');

Storage::delete(['file1.jpg', 'file2.jpg']);
```

Copy A File To A New Location

```
Storage::copy('old/file1.jpg', 'new/file1.jpg');
```

Move A File To A New Location

```
Storage::move('old/file1.jpg', 'new/file1.jpg');
```

Get File Size

```
$size = Storage::size('file1.jpg');
```

Get The Last Modification Time (UNIX)

```
$time = Storage::lastModified('file1.jpg');
```

Get All Files Within A Directory

```
$files = Storage::files($directory);

// Recursive...
$files = Storage::allFiles($directory);
```

Get All Directories Within A Directory

```
$directories = Storage::directories($directory);

// Recursive...
$directories = Storage::allDirectories($directory);
```

Create A Directory

```
Storage::makeDirectory($directory);
```

Delete A Directory

```
Storage::deleteDirectory($directory);
```

Full-Stack Features

Hashing

- Introduction
- Basic Usage

Introduction

The Lumen `Hash` facade provides secure Bcrypt hashing for storing user passwords.

Basic Usage

> **Note:** If you intend to use the `Hash` facade, be sure to uncomment the `$app->withFacades()` call in your `bootstrap/app.php` file.

Hashing A Password Using Bcrypt

```
$password = Hash::make('secret');
```

You may also use the `bcrypt` helper function:

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

Verifying A Password Against A Hash

```
if (Hash::check('secret', $hashedPassword))     {
        // The passwords match...
}
```

Checking If A Password Needs To Be Rehashed

```
if (Hash::needsRehash($hashed)) {
        $hashed = Hash::make('secret');
}
```

Full-Stack Features

Mail

- Configuration
- Basic Usage
- Embedding Inline Attachments
- Queueing Mail

Configuration

> **Note:** By default, the `illuminate/mail` package is not included with Lumen, so you will need to add the `illuminate/mail` dependency in your `composer.json` file.

Lumen utilizes Laravel's mail libraries which provides a clean, simple API over the popular SwiftMailer library.

The `MAIL_*` options in your `.env` file are used to configure your mail settings. By default, a sample SMTP configuration is provided. However, you may use any SMTP server you wish.

If you wish to use the PHP `mail` function to send mail, you may change the `MAIL_DRIVER` to `mail` in the configuration file. A `sendmail` driver is also available.

API Drivers

Lumen also includes drivers for the Mailgun and Mandrill HTTP APIs. These APIs are often simpler and quicker than the SMTP servers. Both of these drivers require that the Guzzle 4 HTTP library be installed into your application. You can add Guzzle 4 to your project by adding the following line to your `composer.json` file:

```
"guzzlehttp/guzzle": "~4.0"
```

Mailgun Driver

To use the Mailgun driver, set the `MAIL_DRIVER` option to `mailgun`. Next, create an `config/services.php` configuration file if one does not already exist for your project. Verify that it contains the following options:

```
'mailgun' => [
        'domain' => 'your-mailgun-domain',
        'secret' => 'your-mailgun-key',
],
```

Mandrill Driver

To use the Mandrill driver, set the `MAIL_DRIVER` option to `mandrill`. Next, create an `config/services.php` configuration file if one does not already exist for your project. Verify that it contains the following options:

```
'mandrill' => [
        'secret' => 'your-mandrill-key',
],
```

Log Driver

If the `MAIL_DRIVER` option of your configuration file is set to `log`, all e-mails will be written to your log files, and will not actually be sent to any of the recipients. This is primarily useful for quick, local debugging and content verification.

Basic Usage

> **Note:** If you intend to use the `Mail` facade, be sure to uncomment the `$app->withFacades()` call in your `bootstrap/app.php` file.

The `Mail::send` method may be used to send an e-mail message:

```
Mail::send('emails.welcome', ['key' => 'value'], function($message)     {
        $message->to('foo@example.com', 'John Smith')->subject('Welcome!');
});
```

The first argument passed to the `send` method is the name of the view that should be used as the e-mail body. The second is the data to be passed to the view, often as an associative array where the data items are available to the view by `$key`. The third is a Closure allowing you to specify various options on the e-mail message.

> **Note:** A `$message` variable is always passed to e-mail views, and allows the inline embedding of attachments. So, it is best to

avoid passing a `message` variable in your view payload.

You may also specify a plain text view to use in addition to an HTML view:

```
Mail::send(['html.view', 'text.view'], $data, $callback);
```

Or, you may specify only one type of view using the `html` or `text` keys:

```
Mail::send(['text' => 'view'], $data, $callback);
```

You may specify other options on the e-mail message such as any carbon copies or attachments as well:

```
Mail::send('emails.welcome', $data, function($message) {
        $message->from('us@example.com', 'Laravel');

        $message->to('foo@example.com')->cc('bar@example.com');

        $message->attach($pathToFile);
});
```

When attaching files to a message, you may also specify a MIME type and / or a display name:

```
$message->attach($pathToFile, ['as' => $display, 'mime' => $mime]);
```

If you just need to e-mail a simple string instead of an entire view, use the `raw` method:

```
Mail::raw('Text to e-mail', function($message) {
        $message->from('us@example.com', 'Laravel');

        $message->to('foo@example.com')->cc('bar@example.com');
});
```

Note: The message instance passed to a `Mail::send` Closure extends the SwiftMailer message class, allowing you to call any method on that class to build your e-mail messages.

Embedding Inline Attachments

Embedding inline images into your e-mails is typically cumbersome; however, Laravel provides a convenient way to attach images to your e-mails and retrieving the appropriate CID.

Embedding An Image In An E-Mail View

```
<body>
        Here is an image:

        <img src="<?php echo $message->embed($pathToFile); ?>">
</body>
```

Embedding Raw Data In An E-Mail View

```
<body>
        Here is an image from raw data:

        <img src="<?php echo $message->embedData($data, $name); ?>">
</body>
```

Note that the `$message` variable is always passed to e-mail views by the `Mail` facade.

Queueing Mail

Queueing A Mail Message

Note: Before queueing mail messages, you will need to add the `jeremeamia/superclosure` (~2.0) dependency to your `composer.json` file.

Since sending e-mail messages can drastically lengthen the response time of your application, many developers choose to queue e-mail messages for background sending. Lumen and Laravel makes this easy using its built-in unified queue API. To queue a mail message, simply use the `queue` method on the `Mail` facade:

```
Mail::queue('emails.welcome', $data, function($message) {
        $message->to('foo@example.com', 'John Smith')->subject('Welcome!');
});
```

You may also specify the number of seconds you wish to delay the sending of the mail message using the `later` method:

```
Mail::later(5, 'emails.welcome', $data, function($message) {
```

```
        $message->to('foo@example.com', 'John Smith')->subject('Welcome!');
});
```

If you wish to specify a specific queue or "tube" on which to push the message, you may do so using the `queueOn` and `laterOn` methods:

```
Mail::queueOn('queue-name', 'emails.welcome', $data, function($message) {
        $message->to('foo@example.com', 'John Smith')->subject('Welcome!');
});
```

Full-Stack Features

Pagination

- Configuration
- Usage
- Appending To Pagination URLs
- Converting To JSON

Configuration

In other frameworks, pagination can be very painful. Lumen and Laravel makes it a breeze. Lumen can generate an intelligent "range" of links based on the current page. The generated HTML is compatible with the Bootstrap CSS framework.

Of course, if you are building a JSON API, the paginator will generate a useful JSON response, including URLs to the previous and next "page".

Usage

There are several ways to paginate items. The simplest is by using the `paginate` method on the query builder or an Eloquent model.

Paginating Database Results

```
$users = DB::table('users')->paginate(15);
```

> **Note:** Currently, pagination operations that use a `groupBy` statement cannot be executed efficiently by Lumen and Laravel. If you need to use a `groupBy` with a paginated result set, it is recommended that you query the database and create a paginator manually.

Creating A Paginator Manually

Sometimes you may wish to create a pagination instance manually, passing it an array of items. You may do so by creating either an `Illuminate\Pagination\Paginator` or `Illuminate\Pagination\LengthAwarePaginator` instance, depending on your needs.

Paginating An Eloquent Model

You may also paginate Eloquent models:

```
$allUsers = User::paginate(15);

$someUsers = User::where('votes', '>', 100)->paginate(15);
```

The argument passed to the `paginate` method is the number of items you wish to display per page. Once you have retrieved the results, you may simply return them from your route / controller. The results will automatically be converted to JSON.

You may also display the results in your view. Create the pagination links using the `render` method:

```
<div class="container">
        @foreach ($users as $user)
                {{ $user->name }}
        @endforeach
</div>

{!! $users->render() !!}
```

You may also access additional pagination information via the following methods on the paginator:

- `currentPage`
- `lastPage`
- `perPage`
- `hasMorePages`
- `url`
- `nextPageUrl`
- `total`
- `count`

"Simple Pagination"

If you do not need to inform the user how many "total" items you are paginating, you have the option of using the `simplePaginate` method to perform a more efficient query. This is useful for larger datasets:

```
$someUsers = User::where('votes', '>', 100)->simplePaginate(15);
```

Customizing The Paginator URI

You may also customize the URI used by the paginator via the `setPath` method:

```
$users = User::paginate();

$users->setPath('custom/url');
```

The example above will create URLs like the following: http://example.com/custom/url?page=2

Appending To Pagination URLs

You can add to the query string of pagination links using the `appends` method on the Paginator:

```
$users->appends(['sort' => 'votes']);
```

This will generate URLs that look something like this:

```
http://example.com/something?page=2&sort=votes
```

If you wish to append a "hash fragment" to the paginator's URLs, you may use the `fragment` method:

```
$users->fragment('foo');
```

This method call will generate URLs that look something like this:

```
http://example.com/something?page=2#foo
```

Converting To JSON

The `Paginator` class implements the `Illuminate\Contracts\Support\JsonableInterface` contract and exposes the `toJson` method. You may also convert a `Paginator` instance to JSON by returning it from a route. The JSON'd form of the instance will include some "meta" information such as `total`, `current_page`, and `last_page`. The instance's data will be available via the `data` key in the JSON array.

Full-Stack Features

Session

- Configuration
- Session Usage
- Flash Data
- Database Sessions
- Session Drivers

Configuration

Since HTTP driven applications are stateless, sessions provide a way to store information about the user across requests. Lumen, like Laravel, ships with a variety of session back-ends available for use through a clean, unified API. Support for popular back-ends such as Memcached, Redis, and databases is included out of the box.

The session driver is controlled by the `SESSION_DRIVER` configuration option in your `.env` file. By default, Lumen is configured to use the `memcached` session driver, which will work well for the majority of applications.

> **Note:** If you are using the `.env` file to configure your application, don't forget to uncomment the `Dotenv::load()` method in your `bootstrap/app.php` file.

Before using Redis sessions with Lumen, you will need to install the `predis/predis` package (~1.0) and `illuminate/redis` package (~5.0) via Composer.

Reserved Keys

The Lumen framework uses the `flash` session key internally, so you should not add an item to the session by that name.

Session Usage

Enabling The Session

> **Note:** Before using sessions, you must uncomment the middleware within the `$app->middleware()` method call in your `bootstrap/app.php` file.

Accessing The Session

The session may be accessed in several ways, via the HTTP request's `session` method, the `Session` facade, or the `session` helper function. When the `session` helper is called without arguments, it will return the entire session object. For example:

```
session()->regenerate();
```

Storing An Item In The Session

```
Session::put('key', 'value');
```

```
session(['key' => 'value']);
```

Push A Value Onto An Array Session Value

```
Session::push('user.teams', 'developers');
```

Retrieving An Item From The Session

```
$value = Session::get('key');
```

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

Retrieving An Item Or Returning A Default Value

```
$value = Session::get('key', 'default');
```

```
$value = Session::get('key', function() { return 'default'; });
```

Retrieving An Item And Forgetting It

```
$value = Session::pull('key', 'default');
```

Retrieving All Data From The Session

```
$data = Session::all();
```

Determining If An Item Exists In The Session

```
if (Session::has('users')) {
        //
}
```

Removing An Item From The Session

```
Session::forget('key');
```

Removing All Items From The Session

```
Session::flush();
```

Regenerating The Session ID

```
Session::regenerate();
```

Flash Data

Sometimes you may wish to store items in the session only for the next request. You may do so using the `Session::flash` method:

```
Session::flash('key', 'value');
```

Reflashing The Current Flash Data For Another Request

```
Session::reflash();
```

Reflashing Only A Subset Of Flash Data

```
Session::keep(['username', 'email']);
```

Database Sessions

When using the `database` session driver, you will need to setup a table to contain the session items. Below is an example `Schema` declaration for the table:

```
Schema::create('sessions', function($table)
{
        $table->string('id')->unique();
        $table->text('payload');
        $table->integer('last_activity');
});
```

Session Drivers

The session "driver" defines where session data will be stored for each request. Lumen, like Laravel, ships with several great drivers out of the box:

- `file` - sessions will be stored in `storage/framework/sessions`.
- `cookie` - sessions will be stored in secure, encrypted cookies.
- `database` - sessions will be stored in a database used by your application.
- `memcached` / `redis` - sessions will be stored in one of these fast, cached based stores.
- `array` - sessions will be stored in a simple PHP array and will not be persisted across requests.

Full-Stack Features

Templates

- Blade Templating
- Other Blade Control Structures

Blade Templating

Blade is a simple, yet powerful templating engine provided with Laravel, and it's even available in Lumen. Blade is driven by *template inheritance* and *sections*. All Blade templates should use the `.blade.php` extension.

Defining A Blade Layout

```
<!-- Stored in resources/views/layouts/master.blade.php -->

<html>
        <head>
                <title>App Name - @yield('title')</title>
        </head>
        <body>
                @section('sidebar')
                        This is the master sidebar.
                @show

                <div class="container">
                        @yield('content')
                </div>
        </body>
</html>
```

Using A Blade Layout

```
@extends('layouts.master')

@section('title', 'Page Title')

@section('sidebar')
        @@parent

        <p>This is appended to the master sidebar.</p>
@stop

@section('content')
        <p>This is my body content.</p>
@stop
```

Note that views which `extend` a Blade layout simply override sections from the layout. Content of the layout can be included in a child view using the `@@parent` directive in a section, allowing you to append to the contents of a layout section such as a sidebar or footer.

Sometimes, such as when you are not sure if a section has been defined, you may wish to pass a default value to the `@yield` directive. You may pass the default value as the second argument:

```
@yield('section', 'Default Content')
```

Other Blade Control Structures

Echoing Data

```
Hello, {{ $name }}.

The current UNIX timestamp is {{ time() }}.
```

Echoing Data After Checking For Existence

Sometimes you may wish to echo a variable, but you aren't sure if the variable has been set. Basically, you want to do this:

```
{{ isset($name) ? $name : 'Default' }}
```

However, instead of writing a ternary statement, Blade allows you to use the following convenient short-cut:

```
{{ $name or 'Default' }}
```

Displaying Raw Text With Curly Braces

If you need to display a string that is wrapped in curly braces, you may escape the Blade behavior by prefixing your text with an `@` symbol:

```
@{{ This will not be processed by Blade }}
```

If you don't want the data to be escaped, you may use the following syntax:

```
Hello, {!! $name !!}.
```

> **Note:** Be very careful when echoing content that is supplied by users of your application. Always use the double curly brace syntax to escape any HTML entities in the content.

If Statements

```
@if (count($records) === 1)
        I have one record!
@elseif (count($records) > 1)
        I have multiple records!
@else
        I don't have any records!
@endif

@unless (Auth::check())
        You are not signed in.
@endunless
```

Loops

```
@for ($i = 0; $i < 10; $i++)
        The current value is {{ $i }}
@endfor

@foreach ($users as $user)
        <p>This is user {{ $user->id }}</p>
@endforeach

@forelse($users as $user)
        <li>{{ $user->name }}</li>
@empty
        <p>No users</p>
@endforelse

@while (true)
        <p>I'm looping forever.</p>
@endwhile
```

Including Sub-Views

```
@include('view.name')
```

You may also pass an array of data to the included view:

```
@include('view.name', ['some' => 'data'])
```

Overwriting Sections

To overwrite a section entirely, you may use the `overwrite` statement:

```
@extends('list.item.container')

@section('list.item.content')
        <p>This is an item of type {{ $item->type }}</p>
@overwrite
```

Displaying Language Lines

```
@lang('language.line')

@choice('language.line', 1)
```

Comments

```
{{-- This comment will not be in the rendered HTML --}}
```