Lumen

lumen.laravel.com

DOCUMENTATION

5.3

Laravel Lumen Documentation - 5.3

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: Wednesday, 08-May-19 12:29:51 CEST

Contents

Prologue

Release Notes

Lumen 5.2.0

Lumen 5.2.0 upgrades the framework to use the 5.2 family of Laravel components, as well as introduces some significant changes to Lumen's underlying philosophy and purpose.

Only Stateless APIs

Lumen 5.2 represents a shift on slimming Lumen to focus solely on serving stateless, JSON APIs. As such, sessions and views are no longer included with the framework. If you need access to these features, you should use the full Laravel framework. Upgrading your Lumen application to the full Laravel framework mainly involves copying your routes and classes over into a fresh installation of Laravel. Since Laravel and Lumen share many of the same components, your classes should not require any modification.

Authentication

Because sessions are no longer included with Lumen, authentication must be done statelessly using API tokens or headers. You have complete control over the authentication process in the new `AuthServiceProvider`. Please review the authentication documentation for more information.

Testing Helpers

Since sessions and views are no longer included with Lumen, all of the form interaction testing helpers have been removed. The testing helpers for JSON APIs remain, so be sure to review the testing documentation.

Lumen 5.1.0

Lumen 5.1.0 upgrades the framework to use the 5.1 family of Laravel components. Features such as event broadcasting, middleware parameters, and testing improvements are now available in Lumen. For the full Laravel 5.1 release notes, consult the Laravel documentation.

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.

Prologue

Upgrade Guide

Upgrading To 5.3.0 From 5.2

Lumen 5.3 does not change the structure of the framework. Instead, it serves as a maintenance release to upgrade the underlying Laravel packages to the 5.3 release series. Before upgrading your application to Lumen 5.3, you should review the Laravel 5.3 upgrade guide and make any applicable changes to your application according to which Laravel components you are using.

Once you have made the necessary adjustments to your application, you may upgrade your Lumen framework dependency in your `composer.json` file and run the `composer update` command:

```
"laravel/lumen-framework": "5.3.*"
```

Upgrading To 5.2.0 From 5.1

Lumen 5.2 represents a more decided shift towards focusing on stateless APIs. Therefore, sessions have been removed from the framework. If you would like to use these features, you should upgrade your Lumen 5.1 application to Laravel 5.2.

Upgrading your Lumen application to the full Laravel framework mainly involves copying your routes and classes over into a fresh installation of Laravel. Since Laravel and Lumen share many of the same components, your classes should not require any modification.

Updating Dependencies

Update your `composer.json` file to point to `laravel/lumen-framework 5.2.*` and `vlucas/phpdotenv ~2.2`.

Bootstrap

In the `bootstrap/app.php` file you need to modify the `Dotenv::load(...)` method call to the following:

```
try {
    (new Dotenv\Dotenv(__DIR__.'/../'))->load();
} catch (Dotenv\Exception\InvalidPathException $e) {
    //
}
```

Application

Lumen no longer implements the `Illuminate\Contracts\Foundation\Application` contract. Any `Application` contract type-hints should be updated to reference the `Laravel\Lumen\Application` class

directly.

Authentication

Since sessions are no longer support in Lumen, authentication is totally based on stateless authentication via API tokens or headers. You should review the full authentication documentation for more information on how to use the authentication system.

Collections

Eloquent Base Collections

The Eloquent collection instance now returns a base Collection (`Illuminate\Support\Collection`) for the following methods: `pluck`, `keys`, `zip`, `collapse`, `flatten`, `flip`.

Key Preservation

The `slice`, `chunk`, and `reverse` methods now preserve keys on the collection. If you do not want these methods to preserve keys, use the `values` method on the `Collection` instance.

Database

MySQL Dates

Starting with MySQL 5.7, `0000-00-00 00:00:00` is no longer considered a valid date, since `strict` mode is enabled by default. All timestamp columns should receive a valid default value when you insert records into your database. You may use the `useCurrent` method in your migrations to default the timestamp columns to the current timestamps, or you may make the timestamps `nullable` to allow `null` values:

```
$table->timestamp('foo')->nullable();

$table->timestamp('foo')->useCurrent();

$table->nullableTimestamps();
```

MySQL JSON Column Type

The `json` column type now creates actual JSON columns when used by the MySQL driver. If you are not running MySQL 5.7 or above, this column type will not be available to you. Instead, use the `text` column type in your migration.

Eloquent

Date Casts

Any attributes that have been added to your `$casts` property as `date` or `datetime` will now be converted to a string when `toArray` is called on the model or collection of models. This makes the date casting conversion consistent with dates specified in your `$dates` array.

Global Scopes

The global scopes implementation has been re-written to be much easier to use. Your global scopes no longer need a `remove` method, so it may be removed from any global scopes you have written.

If we were calling `getQuery` on an Eloquent query builder to access the underlying query builder instance, you should now call `toBase`.

If you were calling the `remove` method directly for any reason, you should change this call to `$eloquentBuilder->withoutGlobalScope($scope)`.

New methods `withoutGlobalScope` and `withoutGlobalScopes` have been added to the Eloquent query builder. Any calls to `$model->removeGlobalScopes($builder)` may be changed to simply `$builder->withoutGlobalScopes()`.

Primary keys

By default, Eloquent assumes your primary keys are integers and will automatically cast them to integers. For any primary key that is not an integer you should override the `$incrementing` property on your Eloquent model to `false`:

```
/**
 * Indicates if the IDs are auto-incrementing.
 *
 * @var bool
 */
public $incrementing = true;
```

Exception Handling

Your `App\Exceptions\Handler` class' `$dontReport` property should be updated to include at least the following exception types:

```
use Illuminate\Validation\ValidationException;
use Illuminate\Auth\Access\AuthorizationException;
use Illuminate\Database\Eloquent\ModelNotFoundException;
use Symfony\Component\HttpKernel\Exception\HttpException;

/**
 * A list of the exception types that should not be reported.
 *
 * @var array
 */
protected $dontReport = [
    AuthorizationException::class,
    HttpException::class,
    ModelNotFoundException::class,
    ValidationException::class,
];
```

IronMQ

The IronMQ queue driver has been moved into its own package and is no longer shipped with the core framework.

http://github.com/LaravelCollective/iron-queue

Storage

If you made use of Laravel's Flysystem integration, you will need to register the `filesystem` binding. Add the following code to your `bootstrap/app.php`:

```
$app->singleton('filesystem', function ($app) {
    return $app->loadComponent(
        'filesystems',
        Illuminate\Filesystem\FilesystemServiceProvider::class,
        'filesystem'
    );
});
```

Validation

The `ValidatesRequests` trait has been merged into the `ProvidesConvenienceMethods` trait used by Lumen's base controller.

If you previously used the `ValidatesRequests` trait outside of the BaseController, you may copy it from the 5.1 branch or use the full `ProvidesConvenienceMethods` trait.

Testing

The `DatabaseMigrations` and `DatabaseTransactions` traits have moved from `Illuminate\Foundation\Testing\DatabaseMigrations` and `Illuminate\Foundation\Testing\DatabaseTransactions` to a new location. Update your tests to import the new namespace:

```
<?php

use Laravel\Lumen\Testing\DatabaseMigrations;
use Laravel\Lumen\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
    use DatabaseMigrations;
}
```

Getting Started

Installation

Installation

Server Requirements

The Lumen framework has a few system requirements. Of course, all of these requirements are satisfied by the Laravel Homestead virtual machine, so it's highly recommended that you use Homestead as your local Lumen development environment.

However, if you are not using Homestead, you will need to make sure your server meets the following requirements:

- PHP >= 5.6.4
- OpenSSL PHP Extension
- PDO PHP Extension
- Mbstring PHP Extension

Installing Lumen

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

Via Lumen Installer

First, download the Lumen installer using Composer:

```
composer global require "laravel/lumen-installer"
```

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

```
lumen new blog
```

Via Composer Create-Project

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

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

Serving Your Application

To serve your project locally, you may use the Laravel Homestead virtual machine, Laravel Valet, or the built-in PHP development server:

```
php -S localhost:8000 -t public
```

Configuration

All of the configuration options for the Lumen framework are stored in the `.env` file. Once Lumen is installed, you should also configure your local environment.

Application Key

The next thing you should do after installing Lumen is set your application key to a random string. Typically, this string should be 32 characters long. The key can be set in the `.env` environment file. If you have not renamed the `.env.example` file to `.env`, you should do that now. If the application key is not set, your user encrypted data will not be secure!

Getting Started

Configuration

Introduction

All of the configuration options for the Lumen framework are stored in the `.env` file.

Accessing Configuration Values

You may easily access your configuration values using the global `config` helper function from anywhere in your application. The configuration values may be accessed using "dot" syntax, which includes the name of the file and option you wish to access. A default value may also be specified and will be returned if the configuration option does not exist:

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

To set configuration values at runtime, pass an array to the `config` helper:

```
config(['app.locale' => 'en']);
```

Environment Configuration

It is often helpful to have different configuration values based on the environment the application is running in. For example, you may wish to use a different cache driver locally than you do on your production server. It's easy using environment based configuration.

To make this a cinch, Lumen utilizes the DotEnv PHP library by Vance Lucas. In a fresh Lumen installation, the root directory of your application will contain a `.env.example` file. You should rename the `.env.example` file to `.env` when creating your application.

All of the variables listed in this file will be loaded into the `$_ENV` PHP super-global when your application receives a request. The `env` function may be used to retrieve the values of your environment variables:

```
$debug = env('APP_DEBUG', true);
```

The second value passed to the `env` function is the "default value". This value will be used if no environment variable exists for the given key.

Your `.env` file should not be committed to your application's source control, since each developer / server using your application could require a different environment configuration.

If you are developing with a team, you may wish to continue including a `.env.example` file with your application. By putting place-holder values in the example configuration file, other developers on your team can clearly see which environment variables are needed to run your application.

Determining The Current Environment

The current application environment is determined via the `APP_ENV` variable from your `.env` file. You may access this value via the `environment` method on the application instance:

```
$environment = app()->environment();
```

You may also pass arguments to the `environment` method to check if the environment matches a given value. If necessary, you may even pass multiple values to the `environment` method. If the environment matches any of the given values, the method will return `true`:

```
if (app()->environment('local')) {
    // The environment is local
}

if (app()->environment('local', 'staging')) {
    // The environment is either local OR staging...
}
```

The Basics

HTTP Routing

Basic Routing

You will define all of the routes for your application in the `routes/web.php` file. The most basic Lumen routes simply accept a URI and a `Closure`:

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

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

Available Router Methods

The router allows you to register routes that respond to any HTTP verb:

```
$app->get($uri, $callback);
$app->post($uri, $callback);
$app->put($uri, $callback);
$app->patch($uri, $callback);
$app->delete($uri, $callback);
$app->options($uri, $callback);
```

Route Parameters

Required Parameters

Of course, sometimes you will need to capture segments of the URI within your route. For example, you may need to capture a user's ID from the URL. You may do so by defining route parameters:

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

You may define as many route parameters as required by your route:

```
$app->get('posts/{post}/comments/{comment}', function ($postId, $commentId) {
    //
});
```

Route parameters are always encased within "curly" braces. The parameters will be passed into your route's `Closure` when the route is executed.

Note: Route parameters cannot contain the - character. Use an underscore (_) instead.

Named Routes

Named routes allow the convenient generation of URLs or redirects for specific routes. You may specify a name for a route using the `as` array key when defining the route:

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

You may also specify route names for controller actions:

```
$app->get('profile', [
    'as' => 'profile', 'uses' => 'UserController@showProfile'
]);
```

Generating URLs To Named Routes

Once you have assigned a name to a given route, you may use the route's name when generating URLs or redirects via the global `route` function:

```
// Generating URLs...
$url = route('profile');

// Generating Redirects...
return redirect()->route('profile');
```

If the named route defines parameters, you may pass the parameters as the second argument to the `route` function. The given parameters will automatically be inserted into the URL in their correct positions:

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

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

Route Groups

Route groups allow you to share route attributes, such as middleware or namespaces, across a large number of routes without needing to define those attributes on each individual route. Shared attributes are specified in an array format as the first parameter to the `$app->group` method.

To learn more about route groups, we'll walk through several common use-cases for the feature.

Middleware

To assign middleware to all routes within a group, you may use the `middleware` key in the group attribute array. Middleware will be executed in the order you define this array:

```
$app->group(['middleware' => 'auth'], function () use ($app) {
    $app->get('/', function ()    {
        // Uses Auth Middleware
    });

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

Namespaces

Another common use-case for route groups is assigning the same PHP namespace to a group of controllers. You may use the `namespace` parameter in your group attribute array to specify the namespace for all controllers within the group:

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

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

Route Prefixes

The `prefix` group attribute may be used to prefix each route in the group with a given URI. For example, you may want to prefix all route URIs within the group with `admin`:

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

You may also use the `prefix` parameter to specify common parameters for your grouped routes:

```
$app->group(['prefix' => 'accounts/{account_id}'], function () use ($app) {
    $app->get('detail', function ($accountId)    {
        // Matches The "/accounts/{account_id}/detail" URL
    });
});
```

The Basics

HTTP Middleware

Introduction

HTTP middleware provide a convenient mechanism for filtering HTTP requests entering your application. For example, Lumen includes a middleware that verifies the user of your application is authenticated. If the user is not authenticated, the middleware will redirect the user to the login screen. However, if the user is authenticated, the middleware will allow the request to proceed further into the application.

Of course, additional middleware can be written to perform a variety of tasks besides authentication. 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 should be stored in the `app/Http/Middleware` directory.

Defining Middleware

To create a new middleware, copy the `ExampleMiddleware` that is included with the default Lumen application. In our new middleware, we 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 or equal to `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. For example, the following middleware would perform some task before the request is handled by the application:

```
<?php

namespace App\Http\Middleware;

use Closure;

class BeforeMiddleware
{
    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
{
    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 call to the `$app->middleware()` method in your `bootstrap/app.php` file:

```
$app->middleware([
   App\Http\Middleware\OldMiddleware::class
]);
```

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 `bootstrap/app.php` file's call to the `$app->routeMiddleware()` method:

```
$app->routeMiddleware([
    'auth' => App\Http\Middleware\Authenticate::class,
]);
```

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' => 'auth', function () {
    //
}]);
```

Use an array to assign multiple middleware to the route:

```
$app->get('/', ['middleware' => ['first', 'second'], function () {
    //
}]);
```

Middleware Parameters

Middleware can also receive additional custom parameters. For example, if your application needs to verify that the authenticated user has a given "role" before performing a given action, you could create a `RoleMiddleware` that receives a role name as an additional argument.

Additional middleware parameters will be passed to the middleware after the `$next` argument:

```
<?php

namespace App\Http\Middleware;

use Closure;

class RoleMiddleware
{
    /**
     * Run the request filter.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @param  string  $role
     * @return mixed
     */
    public function handle($request, Closure $next, $role)
    {
        if (! $request->user()->hasRole($role)) {
            // Redirect...
        }

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

```
}
```

Middleware parameters may be specified when defining the route by separating the middleware name and parameters with a `:`. Multiple parameters should be delimited by commas:

```
$app->put('post/{id}', ['middleware' => 'role:editor', function ($id) {
    //
}]);
```

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 writes the session data to storage *after* the response has been sent to the browser. To accomplish this, define the middleware as "terminable" by adding a `terminate` method to the middleware:

```
<?php

namespace Illuminate\Session\Middleware;

use Closure;

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

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

The `terminate` method should receive both the request and the response. Once you have defined a terminable middleware, you should add it to the list of global middleware in your `bootstrap/app.php` file.

When calling the `terminate` method on your middleware, Lumen will resolve a fresh instance of the middleware from the service container. If you would like to use the same middleware instance when the `handle` and `terminate` methods are called, register the middleware with the container using the container's `singleton` method.

The Basics

HTTP 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 stored in the `app/Http/Controllers` directory.

Basic Controllers

Here is an example of a basic controller class. All Lumen controllers should extend the base controller class included with the default Lumen installation:

```
<?php

namespace App\Http\Controllers;

use App\User;

class UserController extends Controller
{
    /**
     * Retrieve the user for the given ID.
     *
     * @param  int  $id
     * @return Response
     */
    public function show($id)
    {
        return User::findOrFail($id);
    }
}
```

We can route to the controller action like so:

```
$app->get('user/{id}', 'UserController@show');
```

Now, when a request matches the specified route URI, the `show` method on the `UserController` class will be executed. Of course, the route parameters will also be passed to the method.

Controllers & Namespaces

It is very important to note that we did not need to specify the full controller namespace when defining the controller route. We only defined the portion of the class name that comes after the `App\Http\Controllers` namespace "root". By default, the `bootstrap/app.php` file will load the `routes.php` file within a route group containing the root controller namespace.

If you choose to nest or organize your controllers using PHP namespaces deeper into the `App\Http\Controllers` directory, simply use the specific class name relative to the `App\Http\Controllers` root namespace. So, if your full controller class is `App\Http\Controllers\Photos\AdminController`, you would register a route like so:

```
$app->get('foo', 'Photos\AdminController@method');
```

Naming Controller Routes

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

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

You may also use the `route` helper to generate a URL to a named controller route:

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

Controller Middleware

Middleware may be assigned to the controller's routes like so:

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

However, it is more convenient to specify middleware within your controller's constructor. Using the `middleware` method from your controller's constructor, you may easily assign middleware to the controller. You may even restrict the middleware to only certain methods on the controller class:

```
class UserController extends Controller
{
    /**
     * Instantiate a new UserController instance.
     *
     * @return void
     */
    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 service container is used to resolve all Lumen controllers. As a result, you are able to type-hint any dependencies your controller may need in its constructor. The dependencies will automatically be resolved and injected into the controller instance:

```
<?php

namespace App\Http\Controllers;

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 action methods. For example, let's type-hint the `Illuminate\Http\Request` instance on one of our methods:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

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. For example, if your route is defined like so:

```
$app->put('user/{id}', 'UserController@update');
```

You may still type-hint the `Illuminate\Http\Request` and access your route parameter `id` by defining your controller method like the following:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Update the specified user.
     *
     * @param  Request  $request
     * @param  string  $id
     * @return Response
     */
    public function update(Request $request, $id)
    {
        //
    }
}
```

The Basics

HTTP Requests

Accessing The Request

To obtain an instance of the current HTTP request via dependency injection, you should type-hint the `Illuminate\Http\Request` 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;

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. For example, if your route is defined like so:

```
$app->put('user/{id}', 'UserController@update');
```

You may still type-hint the `Illuminate\Http\Request` and access your route parameter `id` by defining your controller method like the following:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Update the specified user.
     *
     * @param  Request  $request
```

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

Basic Request Information

The `Illuminate\Http\Request` instance provides a variety of methods for examining the HTTP request for your application and extends the `Symfony\Component\HttpFoundation\Request` class. Here are a few more of the useful methods available on this class:

Retrieving The Request URI

The `path` method returns the request's URI. So, if the incoming request is targeted at `http://domain.com/foo/bar`, the `path` method will return `foo/bar`:

```
$uri = $request->path();
```

The `is` method allows you to verify that the incoming request URI matches a given pattern. You may use the `*` character as a wildcard when utilizing this method:

```
if ($request->is('admin/*')) {
    //
}
```

To get the full URL, not just the path info, you may use the `url` or `fullUrl` methods on the request instance:

```
// Without Query String...
$url = $request->url();

// With Query String...
$url = $request->fullUrl();
```

Retrieving The Request Method

The `method` method will return the HTTP verb for the request. You may also use the `isMethod` method to verify that the HTTP verb matches a given string:

```
$method = $request->method();

if ($request->isMethod('post')) {
    //
}
```

PSR-7 Requests

The PSR-7 standard specifies interfaces for HTTP messages, including requests and responses. If you would like to obtain an instance of a PSR-7 request, you will first need to install a few libraries. Laravel uses the Symfony HTTP Message Bridge component to convert typical Laravel requests and responses into PSR-7 compatible implementations:

```
composer require symfony/psr-http-message-bridge

composer require zendframework/zend-diactoros
```

Once you have installed these libraries, you may obtain a PSR-7 request by simply type-hinting the request type on your route or controller:

```
use Psr\Http\Message\ServerRequestInterface;

$app->get('/', function (ServerRequestInterface $request) {
    //
});
```

If you return a PSR-7 response instance from a route or controller, it will automatically be converted back to a Laravel response instance and be displayed by the framework.

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');
```

You may pass a default value as the second argument to the `input` method. This value will be returned if the requested input value is not present on the request:

```
$name = $request->input('name', 'Sally');
```

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

```
$name = $request->input('products.0.name');

$names = $request->input('products.*.name');
```

Determining If An Input Value Is Present

To determine if a value is present on the request, you may use the `has` method. The `has` method returns `true` if the value is present and is not an empty string:

```
if ($request->has('name')) {
    //
}
```

Retrieving All Input Data

You may also retrieve all of the input data as an `array` using the `all` method:

```
$input = $request->all();
```

Retrieving A Portion Of The Input Data

If you need to retrieve a sub-set of the input data, you may use the `only` and `except` methods. Both of these methods will accept a single `array` or a dynamic list of arguments:

```
$input = $request->only(['username', 'password']);

$input = $request->only('username', 'password');

$input = $request->except(['credit_card']);

$input = $request->except('credit_card');
```

Files

Retrieving Uploaded Files

You may access uploaded files that are included with the `Illuminate\Http\Request` instance using the `file` method. 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:

```
$file = $request->file('photo');
```

You may determine if a file is present on the request using the `hasFile` method:

```
if ($request->hasFile('photo')) {
    //
}
```

Validating Successful Uploads

In addition to checking if the file is present, you may verify that there were no problems uploading the file via the `isValid` method:

```
if ($request->file('photo')->isValid()) {
    //
}
```

Moving Uploaded Files

To move the uploaded file to a new location, you should use the `move` method. This method will move the file from its temporary upload location (as determined by your PHP configuration) to a more permanent destination of your choosing:

```
$request->file('photo')->move($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.

The Basics

HTTP Responses

Basic Responses

Of course, all routes and controllers should return some kind of response to be sent back to the user's browser. Lumen provides several different ways to return responses. The most basic response is simply returning a string from a route or controller:

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

The given string will automatically be converted into an HTTP response by the framework.

Response Objects

However, for most routes and controller actions, you will be returning a full `Illuminate\Http\Response` instance. 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;

$app->get('home', function () {
    return (new Response($content, $status))
                ->header('Content-Type', $value);
});
```

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

```
$app->get('home', function () {
    return response($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.

Attaching Headers To Responses

Keep in mind that most response methods are chainable, allowing for the fluent building of responses. For example, you may use the `header` method to add a series of headers to the response before sending it back to the user:

```
return response($content)
            ->header('Content-Type', $type)
            ->header('X-Header-One', 'Header Value')
            ->header('X-Header-Two', 'Header Value');
```

Or, you may use the `withHeaders` method to specify an array of headers to be added to the response:

```
return response($content)
            ->withHeaders([
                'Content-Type' => $type,
                'X-Header-One' => 'Header Value',
                'X-Header-Two' => 'Header Value',
            ]);
```

Other Response Types

The `response` helper may be used to conveniently generate other types of response instances. When the `response` helper is called without arguments, an implementation of the `Illuminate\Contracts\Routing\ResponseFactory` contract is returned. This contract provides several helpful methods for generating responses.

JSON Responses

The `json` method will automatically set the `Content-Type` header to `application/json`, as well as convert the given array into JSON using the `json_encode` PHP function:

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

You can optionally provide a status code and an array of additional headers:

```
return response()->json(['error' => 'Unauthorized'], 401, ['X-Header-One' => 'Header Value
```

If you would like to create a JSONP response, you may use the `json` method in addition to `setCallback`:

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

File Downloads

The `download` method may be used to generate a response that forces the user's browser to download the file at the given path. The `download` method accepts a file name as the second argument to the method, which will determine the file name that is seen by the user downloading the file. Finally, you may pass an array of HTTP headers as the third argument to the method:

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

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

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

Redirects

Redirect responses are instances of the `Illuminate\Http\RedirectResponse` class, and contain the proper headers needed to redirect the user to another URL. There are several ways to generate a `RedirectResponse` instance. The simplest method is to use the global `redirect` helper method:

```
$app->get('dashboard', function () {
    return redirect('home/dashboard');
});
```

Redirecting To Named Routes

When you call the `redirect` helper with no parameters, an instance of `Laravel\Lumen\Http\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');
```

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', [$user]);
```

More Features

Authentication

Introduction

Authentication in Lumen, while using the same underlying libraries as Laravel, is configured quite differently from the full Laravel framework. Since Lumen does not support session state, incoming requests that you wish to authenticate must be authenticated via a stateless mechanism such as API tokens.

Getting Started

Authentication Service Provider

> Note: Before using Lumen's authentication features, you should uncomment the call to register the `AuthServiceProvider` service provider in your `bootstrap/app.php` file.

The `AuthServiceProvider` located in your `app/Providers` directory contains a single call to `Auth::viaRequest`. The `viaRequest` method accepts a Closure which will be called when the incoming request needs to be authenticated. Within this Closure, you may resolve your `App\User` instance however you wish. If no authenticated user can be found for the request, the Closure should return `null`:

```
$this->app['auth']->viaRequest('api', function ($request) {
        // Return User or null...
});
```

Again, you may retrieve the authenticated user however you wish. You may use an API token in the request headers or query string, a bearer token on the request, or using any other approach your application requires.

Accessing The Authenticated User

Just like in the full Laravel framework, you may use the `Auth::user()` method to retrieve the current user. Alternatively, you may use the `$request->user()` method on an `Illuminate\Http\Request` instance:

```
use Illuminate\Http\Request;

$app->get('/post/{id}', ['middleware' => 'auth', function (Request $request, $id) {
        $user = Auth::user();

        $user = $request->user();

        //
}]);
```

> Note: If you would like to use `Auth::user()` to access the currently authenticated user, you should uncomment the `$app->withFacades()` method in your `bootstrap/app.php` file.

Of course, any routes you wish to authenticate should be assigned the `auth` middleware, so you should uncomment the call to `$app->routeMiddleware()` in your `bootstrap/app.php` file:

```
$app->routeMiddleware([
    'auth' => App\Http\Middleware\Authenticate::class,
]);
```

More Features

Authorization

Introduction

In addition to providing authentication services out of the box, Lumen also provides a simple way to organize authorization logic and control access to resources. There are a variety of methods and helpers to assist you in organizing your authorization logic.

In general, authorization can be used in Lumen the same way it is used in Laravel. We will cover a few differences here, but you should refer to the full Laravel documentation for additional details.

Differences From Laravel

Defining Abilities

The primary difference when using authorization in Lumen compared to Laravel is in regards to how abilities are defined. In Lumen, you may simply use the `Gate` facade in your `AuthServiceProvider` to define abilities:

```
Gate::define('update-post', function ($user, $post) {
    return $user->id === $post->user_id;
});
```

Defining Policies

Unlike Laravel, Lumen does not have a `$policies` array on its `AuthServiceProvider`. However, you may still call the `policy` method on the `Gate` facade from within the provider's `boot` method:

```
Gate::policy(Post::class, PostPolicy::class);
```

Again, to learn more about policies, you should consult the full Laravel documentation.

Checking Abilities

You may "check" abilities just as you would in the full Laravel framework. First, you may use the `Gate` facade. If you choose to use the facade, be sure to enable facades in your `bootstrap/app.php` file. Remember, we don't need to pass the `User` instance into the `allows` method since the currently authenticated user will automatically be passed to your authorization callback:

```
if (Gate::allows('update-post', $post)) {
    //
}

if (Gate::denies('update-post', $post)) {
    abort(403);
}
```

Of course, you may also check if a given `User` instance has a given ability:

```
if ($request->user()->can('update-post', $post)) {
    abort(403);
}

if ($request->user()->cannot('update-post', $post)) {
    abort(403);
}
```

More Features

Cache

Introduction

Laravel provides a unified API for various caching systems. The cache configuration is located in the `.env` file. In this file you may specify which cache driver you would like used by default throughout your application. Laravel supports popular caching backends like Memcached and Redis out of the box.

Differences From Laravel

The Lumen cache drivers utilize the exact same code as the full Laravel cache drivers. Beyond configuration, there are no differences between using the cache in Lumen and using the cache in Laravel; therefore, please consult the full Laravel documentation for usage examples.

> Note: Before using the `Cache` facade, be sure you have uncommented the `$app->withFacades()` method call in your `bootstrap/app.php` file.

Redis Support

Before using a Redis cache with Lumen, you will need to install the `predis/predis (~1.0)` and `illuminate/redis (5.3.*)` packages via Composer. Then, you should register the `Illuminate\Redis\RedisServiceProvider` in your `bootstrap/app.php` file.

If you have not called `$app->withEloquent()` in your `bootstrap/app.php` file, then you should call `$app->configure('database');` in the `bootstrap/app.php` file to ensure the Redis database configuration is properly loaded.

More Features

Database

Configuration

Lumen makes connecting with databases and running queries extremely simple. Currently Lumen 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.

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.

For example, without facades enabled, you may access a database connection via the `app` helper:

```
$results = app('db')->select("SELECT * FROM users");
```

Or, with facades enabled, you may access the database connection via the `DB` facade:

```
$results = DB::select("SELECT * FROM users");
```

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 migrations.

More Features

Encryption

Configuration

Before using Lumens's encrypter, you should set the `APP_KEY` option of your `.env` file to a 32 character, random string. If this value is not properly set, all values encrypted by Lumen will be insecure.

Basic Usage

Encrypting A Value

You may encrypt a value using the `Crypt` facade. All encrypted values are encrypted using OpenSSL and the `AES-256-CBC` cipher. Furthermore, all encrypted values are signed with a message authentication code (MAC) to detect any modifications to the encrypted string.

For example, we may use the `encrypt` method to encrypt a secret and store it on an Eloquent model:

```
<?php

namespace App\Http\Controllers;

use App\User;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Crypt;

class UserController extends Controller
{
        /**
         * Store a secret message for the user.
         *
         * @param  Request  $request
         * @param  int  $id
         * @return Response
         */
        public function storeSecret(Request $request, $id)
        {
                $user = User::findOrFail($id);

                $user->fill([
                        'secret' => Crypt::encrypt($request->secret)
                ])->save();
        }
}
```

Decrypting A Value

Of course, you may decrypt values using the `decrypt` method on the `Crypt` facade. If the value can not be properly decrypted, such as when the MAC is invalid, an

`Illuminate\Contracts\Encryption\DecryptException` will be thrown:

```
use Illuminate\Contracts\Encryption\DecryptException;

try {
        $decrypted = Crypt::decrypt($encryptedValue);
} catch (DecryptException $e) {
        //
}
```

More Features

Errors & Logging

Introduction

When you start a new Lumen project, error and exception handling is already configured for you. In addition, Lumen is integrated with the Monolog logging library, which provides support for a variety of powerful log handlers.

Configuration

Error Detail

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

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

Custom Monolog Configuration

If you would like to have complete control over how Monolog is configured for your application, you may use the application's `configureMonologUsing` method. You should place a call to this method in your `bootstrap/app.php` file:

```
$app->configureMonologUsing(function($monolog) {
    $monolog->pushHandler(...);

    return $monolog;
});

return $app;
```

The Exception Handler

All exceptions are handled by the `App\Exceptions\Handler` class. This class contains two methods: `report` and `render`. We'll examine each of these methods in detail.

The Report Method

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 class where the exception is logged. However, you are free to log exceptions however you wish.

For example, 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);
}
```

Ignoring Exceptions By Type

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.

The Render Method

The `render` method is responsible for converting a given 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:

```
/**
 * Render an exception into an HTTP response.
 *
 * @param  \Illuminate\Http\Request  $request
 * @param  \Exception  $e
 * @return \Illuminate\Http\Response
 */
public function render($request, Exception $e)
{
    if ($e instanceof CustomException) {
        return response('Custom Message');
    }

    return parent::render($request, $e);
}
```

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 generate such a response from anywhere in your application, use the following:

```
abort(404);
```

The `abort` method will immediately raise an exception which will be rendered by the exception handler. Optionally, you may provide the response text:

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

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

Logging

The Lumen logging facilities provide a simple layer on top of the powerful Monolog library. By default, Lumen is configured to create a single log file for your application which is stored in the `storage/logs` directory. You may write information to the logs using the `Log` facade:

```
<?php

namespace App\Http\Controllers;

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

class UserController extends Controller
{
    /**
     * Show the user for the given ID.
     *
     * @param  int  $id
     * @return Response
     */
    public function show($id)
    {
        Log::info('Showing user: '.$id);

        return User::findOrFail($id);
    }
}
```

> Note: Before using the `Log` facade, be sure you have uncommented the `$app->withFacades()` method call in your `bootstrap/app.php` file.

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

```
Log::emergency($error);
Log::alert($error);
Log::critical($error);
Log::error($error);
Log::warning($error);
Log::notice($error);
Log::info($error);
Log::debug($error);
```

Contextual Information

An array of contextual data may also be passed to the log methods. This contextual data will be

formatted and displayed with the log message:

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

More Features

Events

Introduction

Lumen's events provides a simple observer implementation, allowing you to subscribe and listen for events in your application. Event classes are typically stored in the `app/Events` directory, while their listeners are stored in `app/Listeners`.

Differences From Laravel

In general, events in Lumen function exactly like they do in the full-stack Laravel framework, so please review the full Laravel documentation. Event broadcasting is even supported in Lumen, which allows you to listen for your server side events in your client-side JavaScript. However, there are a few minor differences which warrant discussion.

Generators

In Lumen, there are no generator commands to generate events and listeners for you, so you should simply copy the `ExampleEvent` or `ExampleListener` classes to define your own events and listeners. These example classes provide the basic structure of every event and listener.

Registering Events / Listeners

Like the full Laravel framework, the `EventServiceProvider` included with your Lumen application provides a convenient place to register all event listeners. The `listen` property contains an array of all events (keys) and their listeners (values). Of course, you may add as many events to this array as your application requires:

```
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'App\Events\ExampleEvent' => [
        'App\Listeners\ExampleListener',
    ],
];
```

Firing Events

You may use the `event` helper function or `Event` facade to fire events throughout your Lumen application. Again, these functions behave exactly like their full Laravel framework equivalent:

```
event(new ExampleEvent);

Event::fire(new ExampleEvent);
```

More Features

Queues

Introduction

The Lumen queue service provides a unified API across a variety of different queue back-ends. Queues allow you to defer the processing of a time consuming task, such as performing a task on a remote server, until a later time which drastically speeds up web requests to your application.

Like many other parts of the framework, Lumen's queued jobs function identically to Laravel's queued jobs. So, to learn more about queuing jobs in Lumen, please review the full Laravel queue documentation.

Configuration

The queue configuration options are in the `.env` file.

If you would like to thoroughly customize the queue configuration, then you must copy the entire `vendor/laravel/lumen-framework/config/queue.php` file to the `config` directory in the root of your project and adjust the necessary configuration options as needed. If the `config` directory does not exist, then you should create it.

Driver Prerequisites

Database

In order to use the `database` queue driver, you will need database tables to hold the jobs and failures:

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

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

Other Queue Dependencies

The following dependencies are needed for the listed queue drivers:

- Amazon SQS: `aws/aws-sdk-php ~3.0`
- Beanstalkd: `pda/pheanstalk ~3.0`
- Redis: `predis/predis ~1.0`

Differences From Laravel

Like many other parts of the framework, Lumen's queued jobs function identically to Laravel's queued jobs. So, to learn more about queuing jobs in Lumen, please review the full Laravel queue documentation.

However, there are a few minor differences that we will discuss now. First, let's talk about how queued jobs are generated in Lumen.

Generators

Lumen does not include generators for automatically creating new Job classes. Instead, you should copy the `ExampleJob` class that is included with the framework. This class provides the basic structure that is shared by every Job class. The base `Job` that is used by the `ExampleJob` already includes the needed `InteractsWithQueue`, `Queueable`, and `SerializesModels` traits:

```
<?php

namespace App\Jobs;

class ExampleJob extends Job
{
    /**
     * Create a new job instance.
     *
     * @return void
     */
    public function __construct()
    {
        //
    }

    /**
     * Execute the job.
     *
     * @return void
     */
    public function handle()
    {
        //
    }
}
```

Dispatching Jobs

Again, you should consult the full Laravel documentation for complete information on dispatching queued jobs; however, just like in the Laravel framework, you may use the `dispatch` function to dispatch jobs from anywhere within your Lumen application:

```
dispatch(new ExampleJob);
```

Of course, you may also use the `Queue` facade. If you choose to use the facade, be sure to uncomment

the call to `$app->withFacades()` in your `bootstrap/app.php` file:

```
Queue::push(new ExampleJob);
```

More Features

Service Container

Introduction

The Laravel service container is a powerful tool for managing class dependencies and performing dependency injection. Dependency injection is a fancy phrase that essentially means this: class dependencies are "injected" into the class via the constructor or, in some cases, "setter" methods.

Differences From Laravel

Lumen uses the exact same service container as the Laravel framework, so you have access to all of its powerful features. For complete documentation on the container, check out the full Laravel container documentation.

Accessing The Container

The `Laravel\Lumen\Application` instance is an extension of `Illuminate\Container\Container`, so may be treated as the service container for your application.

Typically, you will register bindings into the container within your service providers. Of course, you may use the `bind`, `singleton`, `instance`, and other container methods provided by the container. Remember, all of these methods are documented in the full Laravel container documentation.

Resolving Instances

To resolve things out of the container, you may either type-hint the dependency you need on a class that is already automatically resolved by the container, such as a route Closure, controller constructor, controller method, middleware, event listener, or queued job. Or, you may use the `app` function from anywhere in your application:

```
$instance = app(Something::class);
```

More Features

Service 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, event listeners, middleware, and even routes. 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 as many service providers as your application requires.

Writing Service Providers

All service providers extend the `Illuminate\Support\ServiceProvider` class. This abstract class requires that you define at least one method on your provider: `register`. Within the `register` method, you should only bind things into the service container. You should never attempt to register any event listeners, routes, or any other piece of functionality within the `register` method.

The Register Method

As mentioned previously, within the `register` method, you should only bind things into the service container. You should never attempt to register any event listeners, routes, or any other piece of functionality within the `register` method. Otherwise, you may accidentally use a service that is provided by a service provider which has not loaded yet.

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()
    {
        $this->app->singleton(Connection::class, function ($app) {
            return new Connection(config('riak'));
        });
    }
}
```

This service provider only defines a `register` method, and uses that method to define an implementation of `Riak\Connection` in the service container. If you don't understand how the service container works, check out its documentation.

The Boot Method

So, what if we need to register a view composer within our service provider? This should be done within the `boot` method. This method is called after all other service providers have been registered, meaning you have access to all other services that have been registered by the framework:

```
<?php

namespace App\Providers;

use Queue;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    // Other Service Provider Properties...

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Queue::failing(function ($event) {

        });
    }
}
```

Registering Providers

All service providers are registered in the `bootstrap/app.php` file. This file contains a call to the `$app->register()` method. You may add as many calls to the register method as needed to register all of your providers.

More Features

Testing

Introduction

Lumen is built with 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. The framework also ships with convenient helper methods allowing you to expressively test your application's JSON responses.

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

Test Environment

Lumen automatically configures the cache to the `array` driver while testing, meaning no 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.

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 using PHPUnit. To run your tests, simply execute the `phpunit` command from your terminal:

```
<?php

class FooTest extends TestCase
{
        public function testSomethingIsTrue()
        {
                $this->assertTrue(true);
        }
}
```

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

Application Testing

Lumen provides a very fluent API for making HTTP requests to your application and examining the output.

Testing JSON APIs

Lumen also provides several helpers for testing JSON APIs and their responses. For example, the `get`, `post`, `put`, `patch`, and `delete` methods may be used to issue requests with various HTTP verbs. You may also easily pass data and headers to these methods. To get started, let's write a test to make a `POST` request to `/user` and assert that a given array was returned in JSON format:

```
<?php

class ExampleTest extends TestCase
{
    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->json('POST', '/user', ['name' => 'Sally'])
             ->seeJson([
                'created' => true,
             ]);
    }
}
```

The `seeJson` method converts the given array into JSON, and then verifies that the JSON fragment occurs anywhere within the entire JSON response returned by the application. So, if there are other properties in the JSON response, this test will still pass as long as the given fragment is present.

Verify Exact JSON Match

If you would like to verify that the given array is an exact match for the JSON returned by the application, you should use the `seeJsonEquals` method:

```
<?php

class ExampleTest extends TestCase
{
    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->post('/user', ['name' => 'Sally'])
             ->seeJsonEquals([
                'created' => true,
             ]);
    }
}
```

Authentication

The `actingAs` helper method provides a simple way to authenticate a given user as the current user:

```
<?php

class ExampleTest extends TestCase
{
    public function testApplication()
    {
        $user = factory('App\User')->create();

                $this->actingAs($user)
                     ->get('/user');
    }
}
```

Custom HTTP Requests

If you would like to make a custom HTTP request into your application and get the full `Illuminate\Http\Response` object, you may use the `call` method:

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

        $this->assertEquals(200, $response->status());
}
```

If you are making `POST`, `PUT`, or `PATCH` requests you may pass an array of input data with the request. Of course, this data will be available in your routes and controller via the Request instance:

$response = $this->call('POST', '/user', ['name' => 'Taylor']);

Working With Databases

Lumen also provides a variety of helpful tools to make it easier to test your database driven applications. First, you may use the `seeInDatabase` helper to assert that data exists in the database matching a given set of criteria. For example, if we would like to verify that there is a record in the `users` table with the `email` value of `sally@example.com`, we can do the following:

```
public function testDatabase()
{
        // Make call to application...

        $this->seeInDatabase('users', ['email' => 'sally@foo.com']);
}
```

Of course, the `seeInDatabase` method and other helpers like it are for convenience. You are free to use any of PHPUnit's built-in assertion methods to supplement your tests.

Resetting The Database After Each Test

It is often useful to reset your database after each test so that data from a previous test does not interfere with subsequent tests.

Using Migrations

One option is to rollback the database after each test and migrate it before the next test. Lumen provides a simple `DatabaseMigrations` trait that will automatically handle this for you. Simply use the trait on your test class:

```
<?php

use Laravel\Lumen\Testing\DatabaseMigrations;
use Laravel\Lumen\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
        use DatabaseMigrations;

    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->get('/foo');
    }
}
```

Using Transactions

Another option is to wrap every test case in a database transaction. Again, Lumen provides a convenient `DatabaseTransactions` trait that will automatically handle this:

```
<?php

use Laravel\Lumen\Testing\DatabaseMigrations;
use Laravel\Lumen\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
        use DatabaseTransactions;

    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->get('/foo');
    }
}
```

Model Factories

When testing, it is common to need to insert a few records into your database before executing your test. Instead of manually specifying the value of each column when you create this test data, Lumen allows you to define a default set of attributes for each of your Eloquent models using "factories". To get started, take a look at the `database/factories/ModelFactory.php` file in your application. Out of the box, this file contains one factory definition:

```
$factory->define('App\User', function ($faker) {
```

```
    return [
        'name' => $faker->name,
        'email' => $faker->email,
    ];
});
```

Within the Closure, which serves as the factory definition, you may return the default test values of all attributes on the model. The Closure will receive an instance of the Faker PHP library, which allows you to conveniently generate various kinds of random data for testing.

Of course, you are free to add your own additional factories to the `ModelFactory.php` file.

Multiple Factory Types

Sometimes you may wish to have multiple factories for the same Eloquent model class. For example, perhaps you would like to have a factory for "Administrator" users in addition to normal users. You may define these factories using the `defineAs` method:

```
$factory->defineAs('App\User', 'admin', function ($faker) {
    return [
        'name' => $faker->name,
        'email' => $faker->email,
        'admin' => true,
    ];
});
```

Instead of duplicating all of the attributes from your base user factory, you may use the `raw` method to retrieve the base attributes. Once you have the attributes, simply supplement them with any additional values you require:

```
$factory->defineAs('App\User', 'admin', function ($faker) use ($factory) {
        $user = $factory->raw('App\User');

        return array_merge($user, ['admin' => true]);
});
```

Using Factories In Tests

Once you have defined your factories, you may use them in your tests or database seed files to generate model instances using the global `factory` function. So, let's take a look at a few examples of creating models. First, we'll use the `make` method, which creates models but does not save them to the database:

```
public function testDatabase()
{
        $user = factory('App\User')->make();

        // Use model in tests...
}
```

If you would like to override some of the default values of your models, you may pass an array of values to the `make` method. Only the specified values will be replaced while the rest of the values remain set to their default values as specified by the factory:

```
$user = factory('App\User')->make([
        'name' => 'Abigail',
```

]);

You may also create a Collection of many models or create models of a given type:

```
// Create three App\User instances...
$users = factory('App\User', 3)->make();

// Create an App\User "admin" instance...
$user = factory('App\User', 'admin')->make();

// Create three App\User "admin" instances...
$users = factory('App\User', 'admin', 3)->make();
```

Persisting Factory Models

The `create` method not only creates the model instances, but also saves them to the database using Eloquent's `save` method:

```
public function testDatabase()
{
        $user = factory('App\User')->create();

        // Use model in tests...
}
```

Again, you may override attributes on the model by passing an array to the `create` method:

```
$user = factory('App\User')->create([
        'name' => 'Abigail',
```

]);

Adding Relations To Models

You may even persist multiple models to the database. In this example, we'll even attach a relation to the created models. When using the `create` method to create multiple models, an Eloquent collection instance is returned, allowing you to use any of the convenient functions provided by the collection, such as `each`:

```
$users = factory('App\User', 3)
           ->create()
           ->each(function($u) {
                                $u->posts()->save(factory('App\Post')->make());
                       });
```

Mocking

Mocking Events

If you are making heavy use of Lumen's event system, you may wish to silence or mock certain events while testing. For example, if you are testing user registration, you probably do not want all of a `UserRegistered` event's handlers firing, since these may send "welcome" e-mails, etc.

Lumen provides a convenient `expectsEvents` method that verifies the expected events are fired, but prevents any handlers for those events from running:

```
<?php

class ExampleTest extends TestCase
{
    public function testUserRegistration()
    {
        $this->expectsEvents('App\Events\UserRegistered');

        // Test user registration code...
    }
}
```

If you would like to prevent all event handlers from running, you may use the `withoutEvents` method:

```
<?php

class ExampleTest extends TestCase
{
    public function testUserRegistration()
    {
        $this->withoutEvents();

        // Test user registration code...
    }
}
```

Mocking Jobs

Sometimes, you may wish to simply test that specific jobs are dispatched by your controllers when making requests to your application. This allows you to test your routes / controllers in isolation - set apart from your job's logic. Of course, you can then test the job itself in a separate test class.

Lumen provides a convenient `expectsJobs` method that will verify that the expected jobs are dispatched, but the job itself will not be executed:

```
<?php

class ExampleTest extends TestCase
{
    public function testPurchasePodcast()
    {
        $this->expectsJobs('App\Jobs\PurchasePodcast');

        // Test purchase podcast code...
    }
}
```

> Note: This method only detects jobs that are dispatched via the `dispatch` global helper function or the `$this->dispatch` method from a route or controller. It does not detect jobs that are sent directly to `Queue::push`.

Mocking Facades

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

```
<?php

namespace App\Http\Controllers;
```

```
use Cache;

class UserController extends Controller
{
        /**
         * Show a list of all users of the application.
         *
         * @return Response
         */
        public function index()
        {
                $value = Cache::get('key');

                //
        }
}
```

We can mock the call to the `Cache` facade by using the `shouldReceive` method, which will return an instance of a Mockery mock. Since facades are actually resolved and managed by the Lumen service container, they have much more testability than a typical static class. For example, let's mock our call to the `Cache` facade:

```
<?php

class FooTest extends TestCase
{
        public function testGetIndex()
        {
                Cache::shouldReceive('get')
                                    ->once()
                                    ->with('key')
                                    ->andReturn('value');

                $this->get('/users');
        }
}
```

> Note: You should not mock the `Request` facade. Instead, pass the input you desire into the HTTP helper methods such as `call` and `post` when running your test.

More Features

Validation

Introduction

Lumen provides several different approaches to validate your application's incoming data. By default, Lumen's base controller class uses a `ProvidesConvenienceMethods` trait which provides a convenient method to validate incoming HTTP request with a variety of powerful validation rules.

In general, validation in Lumen works exactly like validation in Laravel, so you should consult the full Laravel validation documentation; however, there are a few important differences.

Differences From Laravel

Form Requests

Form requests are not supported by Lumen. If you would like to use form requests, you should use the full Laravel framework.

The `$this->validate` Method

The `$this->validate` helper which is available in Lumen will always return a JSON response with the relevant error messages. This is in contrast to the Laravel version of the method which will return a redirect response if the request is not an AJAX request. Since Lumen is stateless and does not support sessions, flashing errors to the session is not a possibility. If you would like to use redirects and flashed error data, you should use the full Laravel framework.

Unlike Laravel, Lumen provides access to the `validate` method from within Route closures:

```
use Illuminate\Http\Request;

$app->post('/user', function (Request $request) {
        $this->validate($request, [
                'name' => 'required',
                'email' => 'required|email|unique:users'
        ]);

        // Store User...
});
```

Of course, you are free to manually create validator instances using the `Validator::make` facade method just as you would in Laravel.

The `$errors` View Variable

Lumen does not support sessions out of the box, so the `$errors` view variable that is available in every view in Laravel is not available in Lumen. Should validation fail, the `$this->validate` helper will throw `Illuminate\Validation\ValidationException` with embedded JSON response that

includes all relevant error messages. If you are not building a stateless API that solely sends JSON responses, you should use the full Laravel framework.