Lumen

lumen.laravel.com

DOCUMENTATION

5.1

Laravel Lumen Documentation - 5.1

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 16:15:37 CEST

Contents

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:

- PHP >= 5.5.9
- OpenSSL PHP Extension
- Mbstring PHP Extension
- Tokenizer 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=~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 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 laravel/lumen blog "5.1.*"
```

Configuration

Basic Configuration

Unlike the full-stack Laravel framework which has multiple configuration files, all of the configuration options for the Lumen framework are stored in a single `.env` configuration file.

Directory Permissions

After installing Lumen, you may need to configure some permissions. Directories within the `storage` directory should be writable by your web server or Lumen will not run. If you are using the Homestead virtual machine, these permissions should already be set.

Application Key

After installing Lumen, you should set your application key to a 32 character, random string. 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 sessions and other encrypted data will not be secure!**

> **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.

Additional 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

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.

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. If you install Lumen via Composer, this file will automatically be renamed to `.env`. Otherwise, you should rename the file manually.

All of the variables listed in this file will be loaded into the `$_ENV` PHP super-global when your application receives a request. You may use the `env` helper to retrieve values from these variables. In fact, if you review the Lumen configuration files, you will notice several of the options already using this helper!

Feel free to modify your environment variables as needed for your own local server, as well as your production environment. However, 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.

Configuration Files

You may use full "Laravel style" configuration files if you wish. The default files are stored in the `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 at `config/options.php`, you can load the file like so:

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

Accessing The Current Application Environment

You may access the current application environment via the `environment` method on the `App` facade:

```
$environment = App::environment();
```

You may also pass arguments to the `environment` method to check if the environment matches a given value. You may even pass multiple values if necessary:

```
if (App::environment('local')) {
        // The environment is local
}

if (App::environment('local', 'staging')) {
        // The environment is either local OR staging...
}
```

An application instance may also be accessed via the `app` helper method:

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

Accessing Configuration Values

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

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

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

```
config(['app.timezone' => 'America/Chicago']);
```

Getting Started

Release Notes

- 5.1.0
- 5.0.4
- 5.0 (Based On Laravel 5.0.x)

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.

The Basics

HTTP Routing

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. The most basic Lumen routes simply accept a URI and a `Closure`:

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

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

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

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

Generating URLs To Routes

You may generate URLs to your application's routes using the `url` helper:

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

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.

Regular Expression Constraints

You may constrain the format of your route parameters by defining a regular expression in your route definition:

```
$app->get('user/{name:[A-Za-z]+}', function ($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 using the `as` array key when defining the route:

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

You may also specify route names for controller actions:

```
$app->get('user/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 `route` function:

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

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

If the route defines parameters, you may pass the parameters as the second argument to the `route` method. The given parameters will automatically be inserted into the URL:

```
$app->get('user/{id}/profile', ['as' => 'profile', function ($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 routes. 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:

```
$app->group(['middleware' => 'auth'], function ($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:

```
$app->group(['namespace' => 'App\Http\Controllers\Admin'], function ($app) {

        // Controllers Within The "App\Http\Controllers\Admin" Namespace

});
```

Route Prefixes

The `prefix` group array 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`:

```
$app->group(['prefix' => 'admin'], function ($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:

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

CSRF Protection

> **Note:** You must enable sessions before using this Lumen feature.

Introduction

Lumen 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. To retrieve the current CSRF token value, use the `csrf_token` helper:

```
<?php echo csrf_token(); ?>
```

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

You do not need to manually verify the CSRF token on POST, PUT, or DELETE requests. The `VerifyCsrfToken` HTTP middleware will verify token in the request input matches the token stored in the session.

X-CSRF-TOKEN

In addition to checking for the CSRF token as a POST parameter, the Lumen `VerifyCsrfToken` middleware will also check for the `X-CSRF-TOKEN` request header. You could, for example, store the token in a "meta" tag:

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

Once you have created the `meta` tag, you can instruct a library like jQuery to add the token to all request headers. This provides simple, convenient CSRF protection for your AJAX based applications:

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

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. It is unlikely that you will need to use this value manually.

Form 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:

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

```
abort(404);
```

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

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.

Defining Middleware

Middleware are typically placed in the `app/Http/Middleware` directory. To create a new middleware, define a class with a `handle` method like the following:

```
/**
 * Filter the incoming request.
 *
 * @param  \Illuminate\Http\Request  $request
 * @param  \Closure  $next
 * @return mixed
 */
public function handle($request, Closure $next)
{
    //
}
```

For example, we may define a middleware to 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
{
        /**
         * Filter the incoming request.
         *
         * @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 `$app->middleware()` call in 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 entries for the middleware included with Lumen. 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 bootstrap file, you may use the `middleware` key in the route options array:

```
$app->get('admin/profile', ['middleware' => 'auth', 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 included with 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 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 middlewares in your bootstrap file.

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 typically 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
{
        /**
         * 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}', 'UserController@showProfile');
```

Now, when a request matches the specified route URI, the `showProfile` 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']);
```

Once you have assigned a name to the controller route, you can easily generate URLs to the action. To generate a URL to a controller action, use the `action` helper method. Again, we only need to specify the part of the controller class name that comes after the base `App\Http\Controllers` namespace:

```
$url = action('FooController@method');
```

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:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

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

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  int  $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. The Lumen `Illuminate\Http\Request` 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` method on the request instance:

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

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. Lumen uses the Symfony HTTP Message Bridge component to convert typical Lumen requests and responses into PSR-7 compatible implementations:

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

composer require zendframework/zend-diactoros
```

Next, we need to add a container binding for the`ServerRequestInterface` so it can be resolved:

```
use Symfony\Bridge\PsrHttpMessage\Factory\DiactorosFactory;

$app->bind('Psr\Http\Message\ServerRequestInterface', function ($app) {
    return (new DiactorosFactory)->createRequest($app->make('request'));
});
```

Once you have installed these libraries and registered the container binding, 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 Lumen 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:

```
$input = $request->input('products.0.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 accept a single `array` as their only argument:

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

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

Old Input

> **Note:** You must enable sessions before using this feature.

Lumen allows you to keep input from one request during the next request. This feature is particularly useful for re-populating forms after detecting validation errors. However, if you are using Lumen's included validation services, it is unlikely you will need to manually use these methods, as some of Lumen's built-in validation facilities will call them automatically.

Flashing Input To The Session

The `flash` method on the `Illuminate\Http\Request` instance will flash the current input to the session so that it is available during the user's next request to the application:

```
$request->flash();
```

You may also use the `flashOnly` and `flashExcept` methods to flash a sub-set of the request data into the session:

```
$request->flashOnly('username', 'email');

$request->flashExcept('password');
```

Flash Input Into Session Then 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 using the `withInput` method:

```
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. The `old` method provides a convenient helper for pulling the flashed input data out of the session:

```
$username = $request->old('username');
```

Lumen also provides a global `old` helper function. 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 Cookies From The Request

To retrieve a cookie value from the request, you may use the `cookie` method on the `Illuminate\Http\Request` instance:

```
$value = $request->cookie('name');
```

Attaching A New Cookie To A Response

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

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

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

return $response;
```

To create a long-lived cookie, which lasts for five years, you may use the `forever` method on the cookie factory by first calling the `cookie` helper with no arguments, and then chaining the `forever` method onto the returned cookie factory:

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

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');
```

Verifying File Presence

You may also 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.

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;

$app->get('home', function () {
        return (new Response($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');
```

Attaching Cookies To Responses

The `withCookie` helper method on the response instance allows you to easily attach cookies to the response. For example, you may use the `withCookie` method to generate a cookie and attach it to the response instance:

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

The `withCookie` method accepts additional optional arguments which allow you to further customize your cookie's properties:

```
->withCookie($name, $value, $minutes, $path, $domain, $secure, $httpOnly)
```

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.

View Responses

If you need control over the response status and headers, but also need to return a view as the response content, you may use the `view` method:

```
return response(view('hello', $data))->header('Content-Type', $type);
```

Of course, if you do not need to pass a custom HTTP status code or custom headers, you may simply use the global `view` helper function.

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']);
```

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');
});
```

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

```
$app->post('user/profile', function () {
        // Validate the request...

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

Redirecting To Named Routes

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');
```

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', [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]);
```

Redirecting With Flashed Session Data

> **Note:** You must enable sessions before using this feature.

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. This is particularly convenient for storing status messages after an action:

```
$app->post('user/profile', function () {
        // Update the user's profile...

        return redirect('dashboard')->with('status', 'Profile updated!');
});
```

Of course, after the user is redirected to a new page, you may retrieve and display the flashed message from the session. For example, using Blade syntax:

```
@if (session('status'))
        <div class="alert alert-success">
                {{ session('status') }}
        </div>
@endif
```

The Basics

Views

Basic Usage

Views contain the HTML served by your application and separate your controller / application logic from your presentation logic. Views are stored in the `resources/views` directory.

A simple view might look something like this:

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

<html>
        <body>
                <h1>Hello, <?php echo $name; ?></h1>
        </body>
</html>
```

Since this view is stored at `resources/views/greeting.php`, we may return it using the global `view` helper function 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. In this case, we are passing the `name` variable, which is displayed in the view by simply executing `echo` on the variable.

Of course, views may also be nested within sub-directories of the `resources/views` directory. "Dot" notation may be used to reference nested views. For example, if your view is stored at `resources/views/admin/profile.php`, you may reference it like so:

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

Determining If A View Exists

If you need to determine if a view exists, you may use the `exists` method after calling the `view` helper with no arguments. This method will return `true` if the view exists on disk:

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

When the `view` helper is called without arguments, an instance of `Illuminate\Contracts\View\Factory` is returned, giving you access to any of the factory's methods.

View Data

Passing Data To Views

As you saw in the previous examples, you may easily pass an array of data to views:

```
return view('greetings', ['name' => 'Victoria']);
```

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, such as `<?php echo $key; ?>`. As an alternative to passing a complete array of data to the `view` helper function, you may use the `with` method to add individual pieces of data to the view:

```
$view = view('greeting')->with('name', 'Victoria');
```

Sharing Data With All Views

Occasionally, you may need to share a piece of data with all views that are rendered by your application. You may do so using the view factory's `share` method. Typically, you would place calls to `share` within a service provider's `boot` method. You are free to add them to the `AppServiceProvider` or generate a separate service provider to house them:

```
<?php
```

```
namespace App\Providers;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
        public function boot()
        {
                view()->share('key', 'value');
        }

        /**
         * Register the service provider.
         *
         * @return void
         */
        public function register()
        {
                //
        }
}
```

More Features

Cache

Configuration

Lumen provides a unified API for various caching systems. The cache configuration is located in your application's `env` file. In this file you may specify which cache driver you would like used by default throughout your application. Lumen supports popular caching backends like Memcached and Redis out of the box. For larger applications, it is recommended that you use an in-memory cache such as Memcached or APC.

Cache Prerequisites

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');
});
```

Memcached

Using the Memcached cache requires the Memcached PECL package to be installed. The default configuration uses TCP/IP based on Memcached::addServer.

Redis

Before using a Redis cache with Lumen, you will need to install the `predis/predis` package (~1.0) and `illuminate/redis` package (~5.1) via Composer.

Cache Usage

Obtaining A Cache Instance

The `Illuminate\Contracts\Cache\Factory` and `Illuminate\Contracts\Cache\Repository` contracts provide access to Lumen's cache services. The `Factory` contract provides access to all cache drivers defined for your application. The `Repository` contract is typically an implementation of the default cache driver for your application as specified by your `cache` configuration file.

However, you may also use the `Cache` facade, which is what we will use throughout this documentation. The `Cache` facade provides convenient, terse access to the underlying implementations of the Lumen cache contracts.

For example, let's import the `Cache` facade into a controller:

```
<?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');
                //
        }
}
```

Accessing Multiple Cache Stores

Using the `Cache` facade, you may access various cache stores via the `store` method. The key passed to the `store` method should correspond to one of the stores listed in the `stores` configuration array in your `cache` configuration file:

```
$value = Cache::store('file')->get('foo');

Cache::store('redis')->put('bar', 'baz', 10);
```

Retrieving Items From The Cache

The `get` method on the `Cache` facade is used to retrieve items from the cache. If the item does not exist in the cache, `null` will be returned. If you wish, you may pass a second argument to the `get` method specifying the custom default value you wish to be returned if the item doesn't exist:

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

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

You may even pass a `Closure` as the default value. The result of the `Closure` will be returned if the specified item does not exist in the cache. Passing a Closure allows you to defer the retrieval of default values from a database or other external service:

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

Checking For Item Existence

The `has` method may be used to determine if an item exists in the cache:

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

Incrementing / Decrementing Values

The `increment` and `decrement` methods may be used to adjust the value of integer items in the cache. Both of these methods optionally accept a second argument indicating the amount by which to increment or decrement the item's value:

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

Cache::increment('key', $amount);

Cache::decrement('key');

Cache::decrement('key', $amount);
```

Retrieve Or Update

Sometimes you may wish to retrieve an item from the cache, but also store a default value if the requested item doesn't exist. For example, you may wish to retrieve all users from the cache or, if they don't exist, retrieve them from the database and add them to the cache. You may do this using the `Cache::remember` method:

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

If the item does not exist in the cache, the `Closure` passed to the `remember` method will be executed and its result will be placed in the cache.

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

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

Retrieve And Delete

If you need to retrieve an item from the cache and then delete it, you may use the `pull` method. Like the `get` method, `null` will be returned if the item does not exist in the cache:

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

Storing Items In The Cache

You may use the `put` method on the `Cache` facade to store items in the cache. When you place an item in the cache, you will need to specify the number of minutes for which the value should be cached:

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

Instead of passing the number of minutes until the item expires, you may also pass a PHP `DateTime` instance representing the expiration time of the cached item:

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

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

The `add` method will only add the item to the cache if it does not already exist in the cache store. The method will return `true` if the item is actually added to the cache. Otherwise, the method will return `false`:

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

The `forever` method may be used to store an item in the cache permanently. These values must be manually removed from the cache using the `forget` method:

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

Removing Items From The Cache

You may remove items from the cache using the `forget` method on the `Cache` facade:

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

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.

> **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.

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 `bootstrap/app.php` file to a 32 character, random string. If this value is not properly set, all values encrypted by Lumens 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 Crypt;
use App\User;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

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` configuration file.

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

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()->view('errors.custom', [], 500);
        }

    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 daily log files for your application which are 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\Http\Controllers\Controller;

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

                return view('user.profile', ['user' => User::findOrFail($id)]);
        }
}
```

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

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

Registering Events / Listeners

The `EventServiceProvider` included with your Lumen application provides a convenient place to register all event listeners. The provider is not loaded by default and must be enabled by un-commenting the following line in your `bootstrap/app.php` file:

```
// $app->register(App\Providers\EventServiceProvider::class);
```

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. For example, let's add our `PodcastWasPurchased` event:

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

Defining Events

An event class is simply a data container which holds the information related to the event. For example, let's assume our generated `PodcastWasPurchased` event receives a Eloquent ORM object:

```
<?php

namespace App\Events;

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

class PodcastWasPurchased extends Event
{
    use SerializesModels;

    public $podcast;

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

As you can see, this event class contains no special logic. It is simply a container for the `Podcast` object that was purchased. The

`SerializesModels` trait used by the event will gracefully serialize any Eloquent models if the event object is serialized using PHP's `serialize` function.

Defining Listeners

Next, let's take a look at the listener for our example event. Event listeners receive the event instance in their `handle` method. Within the `handle` method, you may perform any logic necessary to respond to the event.

```
<?php

namespace App\Listeners;

use App\Events\PodcastWasPurchased;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class EmailPurchaseConfirmation
{
    /**
     * Create the event listener.
     *
     * @return void
     */
    public function __construct()
    {
        //
    }

    /**
     * Handle the event.
     *
     * @param  PodcastWasPurchased  $event
     * @return void
     */
    public function handle(PodcastWasPurchased $event)
    {
        // Access the podcast using $event->podcast...
    }
}
```

Your event listeners may also type-hint any dependencies they need on their constructors. All event listeners are resolved via the Lumen service container, so dependencies will be injected automatically.

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 listener's `handle` method.

Queued Event Listeners

Need to queue an event listener? It couldn't be any easier. Simply add the `ShouldQueue` interface to the listener class. Listeners generated by the `event:generate` Artisan command already have this interface imported into the current namespace, so you can use it immediately:

```
<?php

namespace App\Listeners;

use App\Events\PodcastWasPurchased;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class EmailPurchaseConfirmation implements ShouldQueue
{
        //
}
```

That's it! Now, when this listener is called for an event, it will be queued automatically by the event dispatcher using Lumen's queue system. If no exceptions are thrown when the listener is executed by the queue, the queued job will automatically be deleted after it has processed.

Manually Accessing The Queue

If you need to access the underlying queue job's `delete` and `release` methods manually, you may do so. The `Illuminate\Queue\InteractsWithQueue` trait, which is imported by default on generated listeners, gives you access to these methods:

```
<?php
```

```
namespace App\Listeners;

use App\Events\PodcastWasPurchased;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class EmailPurchaseConfirmation implements ShouldQueue
{
        use InteractsWithQueue;

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

Firing Events

To fire an event, you may use the `Event` facade, passing an instance of the event to the `fire` method. The `fire` method will dispatch the event to all of its registered listeners:

```
<?php

namespace App\Http\Controllers;

use Event;
use App\Podcast;
use App\Events\PodcastWasPurchased;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
        /**
         * Show the profile for the given user.
         *
         * @param  int  $userId
         * @param  int  $podcastId
         * @return Response
         */
        public function purchasePodcast($userId, $podcastId)
        {
                $podcast = Podcast::findOrFail($podcastId);

                // Purchase podcast logic...

                Event::fire(new PodcastWasPurchased($podcast));
        }
}
```

Alternatively, you may use the global `event` helper function to fire events:

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

Broadcasting Events

In many modern web applications, web sockets are used to implement real-time, live-updating user interfaces. When some data is updated on the server, a message is typically sent over a websocket connection to be handled by the client.

To assist you in building these types of applications, Lumen makes it easy to "broadcast" your events over a websocket connection. Broadcasting your Lumen events allows you to share the same event names between your server-side code and your client-side JavaScript framework.

Configuration

Lumen supports several broadcast drivers out of the box: Pusher, Redis, and a `log` driver for local development and debugging. A configuration example is included for each of these drivers. The `BROADCAST_DRIVER` configuration option may be used to set the default driver.

Broadcast Prerequisites

The following dependencies are needed for event broadcasting:

- Pusher: `pusher/pusher-php-server ~2.0`
- Redis: `predis/predis ~1.0` `illuminate/redis ~5.1`

Queue Prerequisites

Before broadcasting events, you will also need to configure and run a queue listener. All event broadcasting is done via queued jobs so that the response time of your application is not seriously affected.

Marking Events For Broadcast

To inform Lumen that a given event should be broadcast, implement the `Illuminate\Contracts\Broadcasting\ShouldBroadcast` interface on the event class. The `ShouldBroadcast` interface requires you to implement a single method: `broadcastOn`. The `broadcastOn` method should return an array of "channel" names that the event should be broadcast on:

```
<?php

namespace App\Events;

use App\User;
use App\Events\Event;
use Illuminate\Queue\SerializesModels;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;

class ServerCreated extends Event implements ShouldBroadcast
{
    use SerializesModels;

    public $user;

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

    /**
     * Get the channels the event should be broadcast on.
     *
     * @return array
     */
    public function broadcastOn()
    {
        return ['user.'.$this->user->id];
    }
}
```

Then, you only need to fire the event as you normally would. Once the event has been fired, a queued job will automatically broadcast the event over your specified broadcast driver.

Broadcast Data

When an event is broadcast, all of its `public` properties are automatically serialized and broadcast as the event's payload, allowing you to access any of its public data from your JavaScript application. So, for example, if your event has a single public `$user` property that contains an Eloquent model, the broadcast payload would be:

```
{
        "user": {
                "id": 1,
                "name": "Jonathan Banks"
                ...
        }
}
```

However, if you wish to have even more fine-grained control over your broadcast payload, you may add a `broadcastWith` method to your event. This method should return the array of data that you wish to broadcast with the event:

```
/**
 * Get the data to broadcast.
 *
 * @return array
 */
public function broadcastWith()
{
    return ['user' => $this->user->id];
}
```

Consuming Event Broadcasts

Pusher

You may conveniently consume events broadcast using the Pusher driver using Pusher's JavaScript SDK. For example, let's consume the `App\Events\ServerCreated` event from our previous examples:

```
this.pusher = new Pusher('pusher-key');

this.pusherChannel = this.pusher.subscribe('user.' + USER_ID);

this.pusherChannel.bind('App\\Events\\ServerCreated', function(message) {
        console.log(message.user);
});
```

Redis

If you are using the Redis broadcaster, you will need to write your own Redis pub/sub consumer to receive the messages and broadcast them using the websocket technology of your choice. For example, you may choose to use the popular Socket.io library which is written in Node.

Using the `socket.io` and `ioredis` Node libraries, you can quickly write an event broadcaster to publish all events that are broadcast by your Lumen application:

```
var app = require('http').createServer(handler);
var io = require('socket.io')(app);

var Redis = require('ioredis');
var redis = new Redis();

app.listen(6001, function() {
        console.log('Server is running!');
});

function handler(req, res) {
        res.writeHead(200);
        res.end('');
}

io.on('connection', function(socket) {
        //
});

redis.psubscribe('*', function(err, count) {
        //
});

redis.on('pmessage', function(subscribed, channel, message) {
        message = JSON.parse(message);
        io.emit(channel + ':' + message.event, message.data);
});
```

Event Subscribers

Event subscribers are classes that may subscribe to multiple events from within the class itself, allowing you to define several event handlers within a single class. Subscribers should define a `subscribe` method, which will be passed an event dispatcher instance:

```
<?php

namespace App\Listeners;

class UserEventListener {

        /**
         * Handle user login events.
         */
        public function onUserLogin($event) {}

        /**
         * Handle user logout events.
         */
        public function onUserLogout($event) {}

        /**
         * Register the listeners for the subscriber.
         *
         * @param  Illuminate\Events\Dispatcher  $events
         * @return array
         */
        public function subscribe($events)
        {
                $events->listen(
                        'App\Events\UserLoggedIn',
                        'App\Listeners\UserEventListener@onUserLogin'
                );
                $events->listen(
                        'App\Events\UserLoggedOut',
                        'App\Listeners\UserEventListener@onUserLogout'
                );
        }

}
```

Registering An Event Subscriber

Once the subscriber has been defined, it may be registered with the event dispatcher. You may register subscribers using the `$subscribe` property on the `EventServiceProvider`. For example, let's add the `UserEventListener`.

```
<?php

namespace App\Providers;

use Illuminate\Foundation\Support\Providers\EventServiceProvider as ServiceProvider;

class EventServiceProvider extends ServiceProvider
{
    /**
     * The event listener mappings for the application.
     *
     * @var array
     */
    protected $listen = [
        //
    ];

    /**
     * The subscriber classes to register.
     *
     * @var array
     */
    protected $subscribe = [
        'App\Listeners\UserEventListener',
    ];
}
```

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 sending an e-mail, until a later time which drastically speeds up web requests to your application.

Configuration

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

Driver Prerequisites

Database

In order to use the `database` queue driver, you will need a database table to hold the jobs. To generate a migration that creates this table, run the `queue:table` Artisan command. Once the migration is created, you may migrate your database using the `migrate` command:

```
php artisan queue:table

php artisan migrate
```

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`
- IronMQ: `iron-io/iron_mq ~2.0`
- Redis: `predis/predis ~1.0`

Writing Job Classes

Job Class Structure

By default, all of the queueable jobs for your application are stored in the `app/Jobs` directory. Job classes are very simple, normally containing only a `handle` method which is called when the job is processed by the queue. To get started, let's take a look at an example job class:

```
<?php

namespace App\Jobs;

use Mail;
use App\User;
use App\Jobs\Job;
use Illuminate\Contracts\Mail\Mailer;
use Illuminate\Queue\SerializesModels;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Bus\SelfHandling;
use Illuminate\Contracts\Queue\ShouldQueue;
```

```
class SendReminderEmail extends Job implements SelfHandling, ShouldQueue
{
    use SerializesModels;

    protected $user;

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

    /**
     * Execute the job.
     *
     * @param  Mailer  $mailer
     * @return void
     */
    public function handle(Mailer $mailer)
    {
        $mailer->send('emails.reminder', ['user' => $this->user], function ($m) {
                //
        });

        $user->reminders()->create(...);
    }
}
```

In this example, note that we were able to pass an Eloquent model directly into the queued job's constructor. Because of the `SerializesModels` trait that the job is using, Eloquent models will be gracefully serialized and unserialized when the job is processing. 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.

The `handle` method is called when the job is processed by the queue. Note that we are able to type-hint dependencies on the `handle` method of the job. The Lumen service container automatically injects these dependencies.

When Things Go Wrong

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. More information on running the queue listener can be found below.

Manually Releasing Jobs

If you would like to `release` the job manually, the `InteractsWithQueue` trait, which is already included in your generated job class, provides access to the queue job `release` method. The `release` method accepts one argument: the number of seconds you wish to wait until the job is made available again:

```
public function handle(Mailer $mailer)
{
        if (condition) {
                $this->release(10);
        }
}
```

Checking The Number Of Run Attempts

As noted above, 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:

```
public function handle(Mailer $mailer)
{
        if ($this->attempts() > 3) {
                //
        }
}
```

Pushing Jobs Onto The Queue

The default Lumen controller located in `app/Http/Controllers/Controller.php` uses a `DispatchesJob` trait. This trait provides several methods allowing you to conveniently push jobs onto the queue, such as the `dispatch` method:

```
<?php

namespace App\Http\Controllers;

use App\User;
use Illuminate\Http\Request;
use App\Jobs\SendReminderEmail;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
        /**
         * Send a reminder e-mail to a given user.
         *
         * @param  Request  $request
         * @param  int  $id
         * @return Response
         */
        public function sendReminderEmail(Request $request, $id)
        {
                $user = User::findOrFail($id);

                $this->dispatch(new SendReminderEmail($user));
        }
}
```

Specifying The Queue For A Job

You may also specify the queue a job should be sent to.

By pushing jobs to different queues, you may "categorize" your queued jobs, and even prioritize how many workers you assign to various queues. This does not push jobs to different queue "connections" as defined by your queue configuration file, but only to specific queues within a single connection. To specify the queue, use the `onQueue` method on the job instance. The `onQueue` method is provided by the base `App\Jobs\Job` class included with Lumen:

```
<?php

namespace App\Http\Controllers;

use App\User;
use Illuminate\Http\Request;
use App\Jobs\SendReminderEmail;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
        /**
         * Send a reminder e-mail to a given user.
         *
         * @param  Request  $request
         * @param  int  $id
         * @return Response
         */
        public function sendReminderEmail(Request $request, $id)
        {
                $user = User::findOrFail($id);

                $job = (new SendReminderEmail($user))->onQueue('emails');

                $this->dispatch($job);
        }
}
```

Delayed Jobs

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 a reminder e-mail 15 minutes after sign-up. You may accomplish this using the `delay` method on your job class, which is provided by the `Illuminate\Bus\Queueable` trait:

```
<?php

namespace App\Http\Controllers;

use App\User;
use Illuminate\Http\Request;
use App\Jobs\SendReminderEmail;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
        /**
```

```
         * Send a reminder e-mail to a given user.
         *
         * @param  Request  $request
         * @param  int  $id
         * @return Response
         */
        public function sendReminderEmail(Request $request, $id)
        {
                $user = User::findOrFail($id);

                $job = (new SendReminderEmail($user))->delay(60);

                $this->dispatch($job);
        }
}
```

In this example, we're specifying that the job should be delayed in the queue for 60 seconds before being made available to workers.

> **Note:** The Amazon SQS service has a maximum delay time of 15 minutes.

Dispatching Jobs 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 on the `DispatchesJobs` trait. By default, this trait is included on the base Lumen controller class:

```
<?php

namespace App\Http\Controllers;

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

class CommerceController extends Controller
{
        /**
         * Process the given order.
         *
         * @param  Request  $request
         * @param  int  $id
         * @return Response
         */
        public function processOrder(Request $request, $id)
        {
                // Process the request...

                $this->dispatchFrom('App\Jobs\ProcessOrder', $request);
        }
}
```

This method will examine the constructor of the given job class and 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 `productId` variable in its constructor, the job bus will attempt to pull the `productId` 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('App\Jobs\ProcessOrder', $request, [
        'taxPercentage' => 20,
]);
```

Running The Queue Listener

Starting The Queue Listener

Lumen includes an Artisan command that will run new jobs as they are pushed onto the queue. You may run the listener using the `queue:listen` command:

```
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.

Queue Priorities

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` queue will always be processed before moving onto jobs from the `low` queue.

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.

Supervisor Configuration

Supervisor is a process monitor for the Linux operating system, and will automatically restart your `queue:listen` or `queue:work` commands if they fail. To install Supervisor on Ubuntu, you may use the following command:

```
sudo apt-get install supervisor
```

Supervisor configuration files are typically stored in the `/etc/supervisor/conf.d` directory. Within this directory, you may create any number of configuration files that instruct supervisor how your processes should be monitored. For example, let's create a `laravel-worker.conf` file that starts and monitors a `queue:work` process:

```
[program:laravel-worker]
process_name=%(program_name)s_%(process_num)02d
command=php /home/forge/app.com/artisan queue:work sqs --sleep=3 --tries=3 --daemon
autostart=true
autorestart=true
user=forge
numprocs=8
redirect_stderr=true
stdout_logfile=/home/forge/app.com/worker.log
```

In this example, the `numprocs` directive will instruct Supervisor to run 8 `queue:work` processes and monitor all of them, automatically restarting them if they fail. Once the configuration file has been created, you may update the Supervisor configuration and start the processes using the following commands:

```
sudo supervisorctl reread

sudo supervisorctl update

sudo supervisorctl start laravel-worker
```

For more information on configuring and using Supervisor, consult the Supervisor documentation. Alternatively, you may use Lumen Forge to automatically configure and manage your Supervisor configuration from a convenient web interface.

Daemon Queue Listener

The `queue:work` Artisan command 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` command:

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.

Coding Considerations For Daemon Queue Listeners

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.

Deploying With Daemon Queue Listeners

Since daemon queue workers are long-lived processes, they will not pick up changes in your code without being restarted. So, the simplest way to deploy an application using daemon queue workers is to restart the workers during your deployment script. You may gracefully restart all of the workers by including the following command in your deployment script:

```
php artisan queue:restart
```

This command will gracefully instruct all queue workers to restart after they finish processing their current job so that no existing jobs are lost.

> **Note:** This command 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.

Dealing With 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 includes 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 name of the failed jobs can be configured via the `config/queue.php` configuration file.

To create a migration for the `failed_jobs` table, you may use the `queue:failed-table` command:

```
php artisan queue:failed-table
```

When running your queue listener, you may specify the maximum number of times a job should be attempted using the `--tries` switch on the `queue:listen` command:

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

Failed Job Events

If you would like to register an event that will be called when a queued job fails, you may use the `Queue::failing` method. This event is a great opportunity to notify your team via e-mail or HipChat. For example, we may attach a callback to this event from the `AppServiceProvider` that is included with Lumen:

```
<?php

namespace App\Providers;

use Queue;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
        public function boot()
        {
                Queue::failing(function ($connection, $job, $data) {
                        // Notify team of failing job...
                });
        }

        /**
         * Register the service provider.
         *
         * @return void
         */
        public function register()
        {
                //
        }
}
```

Failed Method On Job Classes

For more granular control, you may define a `failed` method directly on a queue job class, allowing you to perform job specific actions when a failure occurs:

```
<?php

namespace App\Jobs;

use App\Jobs\Job;
use Illuminate\Queue\SerializesModels;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Bus\SelfHandling;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendReminderEmail extends Job implements SelfHandling, ShouldQueue
{
    use InteractsWithQueue, SerializesModels;

    /**
     * Execute the job.
     *
     * @param  Mailer  $mailer
     * @return void
     */
    public function handle(Mailer $mailer)
    {
                //
    }

    /**
     * Handle a job failure.
     *
     * @return void
     */
        public function failed()
        {
                // Called when the job is failing...
        }
}
```

Retrying Failed Jobs

To view all of your failed jobs that have been inserted into your `failed_jobs` database table, you may use the `queue:failed` Artisan command:

```
php artisan queue:failed
```

The `queue:failed` command 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 command should be issued:

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

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

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

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

```
php artisan queue:flush
```

More Features

Service Container

Binding

Almost all of your service container bindings will be registered within service providers, so all of these examples will demonstrate using the container in that context. However, there is no need to bind classes into the container if they do not depend on any interfaces. The container does not need to be instructed how to build these objects, since it can automatically resolve such "concrete" objects using PHP's reflection services.

Within a service provider, you always have access to the container via the `$this->app` instance variable. We can register a binding using the `bind` method, passing the class or interface name that we wish to register along with a `Closure` that returns an instance of the class:

```
$this->app->bind('HelpSpot\API', function ($app) {
        return new HelpSpot\API($app['HttpClient']);
});
```

Notice that we receive the container itself as an argument to the resolver. We can then use the container to resolve sub-dependencies of the object we are building.

Binding A Singleton

The `singleton` method binds a class or interface into the container that should only be resolved one time, and then that same instance will be returned on subsequent calls into the container:

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

Binding Instances

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);

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

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, let's assume we have an `EventPusher` interface and a `RedisEventPusher` implementation. Once we have coded our `RedisEventPusher` implementation of this interface, we can register it with the service container like so:

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

This tells the container that it should inject the `RedisEventPusher` when a class needs an implementation of `EventPusher`. Now we can type-hint the `EventPusher` interface in a constructor, or any other location where dependencies are injected by the service container:

```
use App\Contracts\EventPusher;

/**
 * Create a new class 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:

```
$this->app->when('App\Handlers\Commands\CreateOrderHandler')
          ->needs('App\Contracts\EventPusher')
          ->give('App\Services\PubNubEventPusher');
```

You may even pass a Closure to the `give` method:

```
$this->app->when('App\Handlers\Commands\CreateOrderHandler')
          ->needs('App\Contracts\EventPusher')
          ->give(function () {
                        // Resolve dependency...
                });
```

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:

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

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

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

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

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

Resolving

There are several ways to resolve something out of the container. First, you may use the `make` method, which accepts the name of the class or interface you wish to resolve:

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

Secondly, you may access the container like an array, since it implements PHP's `ArrayAccess` interface:

```
$fooBar = $this->app['FooBar'];
```

Lastly, but most 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, middleware, and more. In practice, this is how most of your objects are resolved by the container.

The container will automatically inject dependencies for the classes it resolves. For example, you may type-hint a repository defined by your application in a controller's constructor. The repository will automatically be resolved and injected into the class:

```
<?php

namespace App\Http\Controllers;

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)
        {
                //
        }
}
```

Container Events

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

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

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

As you can see, the object being resolved will be passed to the callback, allowing you to set any additional properties on the object before it is given to its consumer.

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.

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

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 accidently 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()
        {
                $this->app->singleton('Riak\Contracts\Connection', function ($app) {
                        return new Connection(config('riak'));
                });
        }
}
```

This service provider only defines a `register` method, and uses that method to define an implementation of `Riak\Contracts\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 Illuminate\Support\ServiceProvider;

class EventServiceProvider extends ServiceProvider
{
	/**
	 * Perform post-registration booting of services.
	 *
	 * @return void
	 */
	public function boot()
	{
		view()->composer('view', function () {
			//
		});
	}

	/**
	 * Register bindings in the container.
	 *
	 * @return void
	 */
	public function register()
	{
		//
	}
}
```

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

Session

Introduction

Since HTTP driven applications are stateless, sessions provide a way to store information about the user across requests. Lumen 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.

Enabling The Session

To enable sessions, you must uncomment all of the middleware within the `$app->middleware()` method call in your `bootstrap/app.php` file.

Configuration

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. In production applications, you may consider using the `memcached` or `redis` drivers for even faster session performance.

The session driver defines where session data will be stored for each request. Lumen ships with several great drivers out of the box:

- `file` - sessions are stored in `storage/framework/sessions`.
- `cookie` - sessions are stored in secure, encrypted cookies.
- `database` - sessions are stored in a database used by your application.
- `memcached` / `redis` - sessions are stored in one of these fast, cached based stores.
- `array` - sessions are stored in a simple PHP array and will not be persisted across requests.

> **Note:** The array driver is typically used for running tests to prevent session data from persisting.

Driver Prerequisites

Database

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');
});
```

Redis

Before using Redis sessions with Lumen, you will need to install the `predis/predis` package (~1.0) and `illuminate/redis` package (~5.1) via Composer.

Other Session Considerations

The Lumen framework uses the `flash` session key internally, so you should not add an item to the session by that name.

Basic Usage

Accessing The Session

First, let's access the session. We can access the session instance via the HTTP request, which can be type-hinted on a controller method. Remember, controller method dependencies are injected via the Lumen service container:

```
<?php

namespace App\Http\Controllers;
```

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

class UserController extends Controller
{
        /**
         * Show the profile for the given user.
         *
         * @param  Request  $request
         * @param  int  $id
         * @return Response
         */
        public function showProfile(Request $request, $id)
        {
                $value = $request->session()->get('key');

                //
        }
}
```

When you retrieve a value from the session, you may also pass a default value as the second argument to the `get` method. This default value will be returned if the specified key does not exist in the session. If you pass a `Closure` as the default value to the `get` method, the `Closure` will be executed and its result returned:

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

$value = $request->session()->get('key', function() {
        return 'default';
});
```

If you would like to retrieve all data from the session, you may use the `all` method:

```
$data = $request->session()->all();
```

You may also use the global `session` PHP function to retrieve and store data in the session:

```
Route::get('home', function () {
        // Retrieve a piece of data from the session...
        $value = session('key');

        // Store a piece of data in the session...
        session(['key' => 'value']);
});
```

Determining If An Item Exists In The Session

The `has` method may be used to check if an item exists in the session. This method will return `true` if the item exists:

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

Storing Data In The Session

Once you have access to the session instance, you may call a variety of functions to interact with the underlying data. For example, the `put` method stores a new piece of data in the session:

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

Pushing To Array Session Values

The `push` method may be used to push a new value onto a session value that is an array. For example, if the `user.teams` key contains an array of team names, you may push a new value onto the array like so:

```
$request->session()->push('user.teams', 'developers');
```

Retrieving And Deleting An Item

The `pull` method will retrieve and delete an item from the session:

```
$value = $request->session()->pull('key', 'default');
```

Deleting Items From The Session

The `forget` method will remove a piece of data from the session. If you would like to remove all data from the session, you may use the `flush` method:

```
$request->session()->forget('key');
```

```
$request->session()->flush();
```

Regenerating The Session ID

If you need to regenerate the session ID, you may use the `regenerate` method:

```
$request->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 `flash` method. Method stored in the session using this method will only be available during the subsequent HTTP request, and then will be deleted. Flash data is primarily useful for short-lived status messages:

```
$request->session()->flash('status', 'Task was successful!');
```

If you need to keep your flash data around for even more requests, you may use the `reflash` method, which will keep all of the flash data around for an additional request. If you only need to keep specific flash data around, you may use the `keep` method:

```
$request->session()->reflash();
```

```
$request->session()->keep(['username', 'email']);
```

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 applications.

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 session and cache to the `array` driver while testing, 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.

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, examining the output, and even filling out forms. For example, take a look at the `ExampleTest.php` file included in your `tests` directory:

```
<?php

class ExampleTest extends TestCase
{
    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->visit('/')
             ->see('Lumen.');
    }
}
```

The `visit` method makes a `GET` request into the application. The `see` method asserts that we should see the given text in the response returned by the application. This is the most basic application test available in Lumen.

Interacting With Your Application

Of course, you can do much more than simply assert that text appears in a given response. Let's take a look at some examples of clicking links and filling out forms:

Clicking Links

In this test, we will make a request to the application, "click" a link in the returned response, and then assert that we landed on a given URI. For example, let's assume there is a link in our response that has a text value of "About Us":

```
<a href="/about-us">About Us</a>
```

Now, let's write a test that clicks the link and asserts the user lands on the correct page:

```
public function testBasicExample()
{
    $this->visit('/')
         ->click('About Us')
         ->seePageIs('/about-us');
}
```

Working With Forms

Lumen also provides several methods for testing forms. The `type`, `select`, `check`, `attach`, and `press` methods allow you to interact with all of your form's inputs. For example, let's imagine this form exists on the application's registration page:

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

        <div>
                Name: <input type="text" name="name">
        </div>

        <div>
                <input type="checkbox" value="yes" name="terms"> Accept Terms
        </div>

        <div>
                <input type="submit" value="Register">
        </div>
</form>
```

We can write a test to complete this form and inspect the result:

```
public function testNewUserRegistration()
{
    $this->visit('/register')
         ->type('Taylor', 'name')
         ->check('terms')
         ->press('Register')
         ->seePageIs('/dashboard');
}
```

Of course, if your form contains other inputs such as radio buttons or drop-down boxes, you may easily fill out those types of fields as well. Here is a list of each form manipulation method:

Method	Description
`$this->type($text, $elementName)`	"Type" text into a given field.
`$this->select($value, $elementName)`	"Select" a radio button or drop-down field.
`$this->check($elementName)`	"Check" a checkbox field.
`$this->attach($pathToFile, $elementName)`	"Attach" a file to the form.
`$this->press($buttonTextOrElementName)`	"Press" a button with the given text or name.

Working With Attachments

If your form contains `file` input types, you may attach files to the form using the `attach` method:

```
public function testPhotoCanBeUploaded()
{
    $this->visit('/upload')
```

```
         ->name('File Name', 'name')
         ->attach($absolutePathToFile, 'photo')
         ->press('Upload')
         ->see('Upload Successful!');
}
```

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->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,
             ]);
    }
}
```

Sessions / Authentication

Lumen provides several helpers for working with the session during testing. First, you may set the session data to a given array using the `withSession` method. This is useful for loading the session with data before testing a request to your application:

```
<?php

class ExampleTest extends TestCase
{
    public function testApplication()
    {
            $this->withSession(['foo' => 'bar'])
                 ->visit('/');
    }
}
```

Of course, one common use of the session is for maintaining user state, such as the authenticated user. The `actingAs` helper method provides a simple way to authenticate a given user as the current user. For example, we may use a model factory to generate and authenticate a user:

```
<?php
```

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

                $this->actingAs($user)
                         ->withSession(['foo' => 'bar'])
                     ->visit('/')
                     ->see('Hello, '.$user->name);
    }
}
```

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 Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
        use DatabaseMigrations;

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

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 Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
        use DatabaseTransactions;

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

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,
        'password' => str_random(10),
        'remember_token' => str_random(10),
    ];
});
```

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,
        'password' => str_random(10),
        'remember_token' => str_random(10),
        '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 `DispatchesJobs` trait's dispatch methods. 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->visit('/users')->see('value');
        }
}
```

> **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 `ValidatesRequests` trait which provides a convenient method to validate incoming HTTP request with a variety of powerful validation rules.

Validation Quickstart

To learn about Lumen's powerful validation features, let's look at a complete example of validating a form and displaying the error messages back to the user.

Defining The Routes

First, let's assume we have the following routes defined in our `app/Http/routes.php` file:

```
// Display a form to create a blog post...
$app->get('post/create', 'PostController@create');

// Store a new blog post...
$app->post('post', 'PostController@store');
```

Of course, the `GET` route will display a form for the user to create a new blog post, while the `POST` route will store the new blog post in the database.

Creating The Controller

Next, let's take a look at a simple controller that handles these routes. We'll leave the `store` method empty for now:

```
<?php

namespace App\Http\Controllers;

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

class PostController extends Controller
{
        /**
         * Show the form the create a new blog post.
         *
         * @return Response
         */
        public function create()
        {
                return view('post.create');
        }

        /**
         * Store a new blog post.
         *
         * @param  Request  $request
         * @return Response
         */
        public function store(Request $request)
        {
                // Validate and store the blog post...
        }
}
```

Writing The Validation Logic

Now we are ready to fill in our `store` method with the logic to validate the new blog post. If you examine your application's base controller (`Laravel\Lumen\Routing\Controller`) class, you will see that the class uses a `ValidatesRequests` trait. This trait provides a convenient `validate` method in all of your controllers.

The `validate` method accepts an incoming HTTP request and a set of validation rules. If the validation rules pass, your code will keep executing normally; however, if validation fails, an exception will be thrown and the proper error response will automatically be sent back to the user. In the case of a traditional HTTP request, a redirect response will be generated, while a JSON response will be sent for AJAX requests.

To get a better understanding of the `validate` method, let's jump back into the `store` method:

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

        // The blog post is valid, store in database...
}
```

As you can see, we simply pass the incoming HTTP request and desired validation rules into the `validate` method. Again, if the validation fails, the proper response will automatically be generated. If the validation passes, our controller will continue executing normally.

Displaying The Validation Errors

So, what if the incoming request parameters do not pass the given validation rules? As mentioned previously, Lumen will automatically redirect the user back to their previous location. In addition, all of the validation errors will automatically be flashed to the session.

Again, notice that we did not have to explicitly bind the error messages to the view in our `GET` route. This is because Lumen 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 `Illuminate\Support\MessageBag`. For more information on working with this object, check out its documentation.

So, in our example, the user will be redirected to our controller's `create` method when validation fails, allowing us to display the error messages in the view:

```
<!-- /resources/views/post/create.blade.php -->

<h1>Create Post</h1>

@if (count($errors) > 0)
```

- 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

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 `strtotime` PHP 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 a PHP `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 given *format*. The format will be evaluated using the PHP `date_parse_from_format` function.

different:*field*

The field under validation must have a different value than *field*.

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 be an integer.

ip

The field under validation must be 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 be numeric.

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:*anotherfield,value,...*

The field under validation must be present if the *anotherfield* 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

The field under validation must be a string.

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.

Specifying A Custom Column Name:

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

Custom Database Connection

Occasionally, you may need to set a custom connection for database queries made by the Validator. As seen above, setting `unique:users` as a validation rule will use the default database connection to query the database. To override this, specify the connection followed by the table name using "dot" syntax:

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

Forcing A Unique Rule To Ignore A Given ID:

Sometimes, you may wish to ignore a given ID during the unique check. For example, consider an "update profile" screen that includes the user's name, e-mail address, and location. Of course, you will want to verify that the e-mail address is unique. However, if the user only changes the name field and not the e-mail field, you do not want a validation error to be thrown because the user is already the owner of the e-mail address. You only want to throw a validation error if the user provides an e-mail address that is already used by a different user. To tell the unique rule to ignore the user's ID, you may pass the ID as the third parameter:

```
'email' => 'unique:users,email_address,'.$user->id
```

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 a valid URL according to PHP's `filter_var` function.

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 add validation rules based on more complex conditional logic. For example, 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 to access your input and files.

Custom Validation Rules

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 `extend` method on the `Validator` facade. Let's use this method within a service provider to register a custom validation rule:

```
<?php

namespace App\Providers;

use Validator;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
        public function boot()
        {
                Validator::extend('foo', function($attribute, $value, $parameters) {
                        return $value == 'foo';
                });
        }

        /**
         * Register the service provider.
         *
         * @return void
         */
        public function register()
        {
                //
        }
}
```

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');
```

Defining The Error Message

You will also need to define an error message for your custom rule. You can do so either using an inline custom message array or by adding an entry in the validation language file. This message should be placed in the first level of the array, not within the `custom` array, which is only for attribute-specific error messages:

```
"foo" => "Your input was invalid!",

"accepted" => "The :attribute must be accepted.",

// The rest of the validation error 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 then making a call to the `replacer` method on the `Validator` facade. You may do this within the `boot` method of a service provider:

```
/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
```

```
{
        Validator::extend(...);

        Validator::replacer('foo', function($message, $attribute, $rule, $parameters) {
                return str_replace(...);
        });
}
```