Configuration

* [Introduction](https://laravel.com/docs/8.x/configuration#introduction)
* [Environment Configuration](https://laravel.com/docs/8.x/configuration#environment-configuration)
  + [Environment Variable Types](https://laravel.com/docs/8.x/configuration#environment-variable-types)
  + [Retrieving Environment Configuration](https://laravel.com/docs/8.x/configuration#retrieving-environment-configuration)
  + [Determining The Current Environment](https://laravel.com/docs/8.x/configuration#determining-the-current-environment)
* [Accessing Configuration Values](https://laravel.com/docs/8.x/configuration#accessing-configuration-values)
* [Configuration Caching](https://laravel.com/docs/8.x/configuration#configuration-caching)
* [Debug Mode](https://laravel.com/docs/8.x/configuration#debug-mode)
* [Maintenance Mode](https://laravel.com/docs/8.x/configuration#maintenance-mode)

[Introduction](https://laravel.com/docs/8.x/configuration#introduction)

All of the configuration files for the Laravel framework are stored in the config directory. Each option is documented, so feel free to look through the files and get familiar with the options available to you.

These configuration files allow you to configure things like your database connection information, your mail server information, as well as various other core configuration values such as your application timezone and encryption key.

[Environment Configuration](https://laravel.com/docs/8.x/configuration#environment-configuration)

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

To make this a cinch, Laravel utilizes the [DotEnv](https://github.com/vlucas/phpdotenv) PHP library. In a fresh Laravel installation, the root directory of your application will contain a .env.example file that defines many common environment variables. During the Laravel installation process, this file will automatically be copied to .env.

Laravel's default .env file contains some common configuration values that may differ based on whether your application is running locally or on a production web server. These values are then retrieved from various Laravel configuration files within the config directory using Laravel's env function.

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

Any variable in your .env file can be overridden by external environment variables such as server-level or system-level environment variables.

[Environment File Security](https://laravel.com/docs/8.x/configuration#environment-file-security)

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. Furthermore, this would be a security risk in the event an intruder gains access to your source control repository, since any sensitive credentials would get exposed.

[Environment Variable Types](https://laravel.com/docs/8.x/configuration#environment-variable-types)

All variables in your .env files are typically parsed as strings, so some reserved values have been created to allow you to return a wider range of types from the env() function:

| .env**Value** | env()**Value** |
| --- | --- |
| true | (bool) true |
| (true) | (bool) true |
| false | (bool) false |
| (false) | (bool) false |
| empty | (string) '' |
| (empty) | (string) '' |
| null | (null) null |
| (null) | (null) null |

If you need to define an environment variable with a value that contains spaces, you may do so by enclosing the value in double quotes:

APP\_NAME="My Application"

[Retrieving Environment Configuration](https://laravel.com/docs/8.x/configuration#retrieving-environment-configuration)

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

'debug' => env('APP\_DEBUG', false),

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

[Determining The Current Environment](https://laravel.com/docs/8.x/configuration#determining-the-current-environment)

The current application environment is determined via the APP\_ENV variable from your .env file. You may access this value via the environment method on the App [facade](https://laravel.com/docs/8.x/facades):

use Illuminate\Support\Facades\App;

$environment = App::environment();

You may also pass arguments to the environment method to determine if the environment matches a given value. The method will return true if the environment matches any of the given values:

if (App::environment('local')) {

// The environment is local

}

if (App::environment(['local', 'staging'])) {

// The environment is either local OR staging...

}

The current application environment detection can be overridden by defining a server-level APP\_ENV environment variable.

[Accessing Configuration Values](https://laravel.com/docs/8.x/configuration#accessing-configuration-values)

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

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

// Retrieve a default value if the configuration value does not exist...

$value = config('app.timezone', 'Asia/Seoul');

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

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

[Configuration Caching](https://laravel.com/docs/8.x/configuration#configuration-caching)

To give your application a speed boost, you should cache all of your configuration files into a single file using the config:cache Artisan command. This will combine all of the configuration options for your application into a single file which can be quickly loaded by the framework.

You should typically run the php artisan config:cache command as part of your production deployment process. The command should not be run during local development as configuration options will frequently need to be changed during the course of your application's development.

If you execute the config:cache command during your deployment process, you should be sure that you are only calling the env function from within your configuration files. Once the configuration has been cached, the .env file will not be loaded; therefore, the env function will only return external, system level environment variables.

[Debug Mode](https://laravel.com/docs/8.x/configuration#debug-mode)

The debug option in your config/app.php configuration file determines how much information about an error is actually displayed to the user. By default, this option is set to respect the value of the APP\_DEBUG environment variable, which is stored in your .env file.

For local development, you should set the APP\_DEBUG environment variable to true. In your production environment, this value should always be false. If the variable is set to true in production, you risk exposing sensitive configuration values to your application's end users.

[Maintenance Mode](https://laravel.com/docs/8.x/configuration#maintenance-mode)

When your application is in maintenance mode, a custom view will be displayed for all requests into your application. This makes it easy to "disable" your application while it is updating or when you are performing maintenance. A maintenance mode check is included in the default middleware stack for your application. If the application is in maintenance mode, a Symfony\Component\HttpKernel\Exception\HttpException instance will be thrown with a status code of 503.

To enable maintenance mode, execute the down Artisan command:

php artisan down

If you would like the Refresh HTTP header to be sent with all maintenance mode responses, you may provide the refresh option when invoking the down command. The Refresh header will instruct the browser to automatically refresh the page after the specified number of seconds:

php artisan down --refresh=15

You may also provide a retry option to the down command, which will be set as the Retry-After HTTP header's value, although browsers generally ignore this header:

php artisan down --retry=60

[Bypassing Maintenance Mode](https://laravel.com/docs/8.x/configuration#bypassing-maintenance-mode)

Even while in maintenance mode, you may use the secret option to specify a maintenance mode bypass token:

php artisan down --secret="1630542a-246b-4b66-afa1-dd72a4c43515"

After placing the application in maintenance mode, you may navigate to the application URL matching this token and Laravel will issue a maintenance mode bypass cookie to your browser:

https://example.com/1630542a-246b-4b66-afa1-dd72a4c43515

When accessing this hidden route, you will then be redirected to the / route of the application. Once the cookie has been issued to your browser, you will be able to browse the application normally as if it was not in maintenance mode.

[Pre-Rendering The Maintenance Mode View](https://laravel.com/docs/8.x/configuration#pre-rendering-the-maintenance-mode-view)

If you utilize the php artisan down command during deployment, your users may still occasionally encounter errors if they access the application while your Composer dependencies or other infrastructure components are updating. This occurs because a significant part of the Laravel framework must boot in order to determine your application is in maintenance mode and render the maintenance mode view using the templating engine.

For this reason, Laravel allows you to pre-render a maintenance mode view that will be returned at the very beginning of the request cycle. This view is rendered before any of your application's dependencies have loaded. You may pre-render a template of your choice using the down command's render option:

php artisan down --render="errors::503"

[Redirecting Maintenance Mode Requests](https://laravel.com/docs/8.x/configuration#redirecting-maintenance-mode-requests)

While in maintenance mode, Laravel will display the maintenance mode view for all application URLs the user attempts to access. If you wish, you may instruct Laravel to redirect all requests to a specific URL. This may be accomplished using the redirect option. For example, you may wish to redirect all requests to the / URI:

php artisan down --redirect=/

[Disabling Maintenance Mode](https://laravel.com/docs/8.x/configuration#disabling-maintenance-mode)

To disable maintenance mode, use the up command:

php artisan up

You may customize the default maintenance mode template by defining your own template at resources/views/errors/503.blade.php.

[Maintenance Mode & Queues](https://laravel.com/docs/8.x/configuration#maintenance-mode-queues)

While your application is in maintenance mode, no [queued jobs](https://laravel.com/docs/8.x/queues) will be handled. The jobs will continue to be handled as normal once the application is out of maintenance mode.

[Alternatives To Maintenance Mode](https://laravel.com/docs/8.x/configuration#alternatives-to-maintenance-mode)

Since maintenance mode requires your application to have several seconds of downtime, consider alternatives like [Laravel Vapor](https://vapor.laravel.com/) and [Envoyer](https://envoyer.io/) to accomplish zero-downtime deployment with Laravel.

Directory Structure

* [Introduction](https://laravel.com/docs/8.x/structure#introduction)
* [The Root Directory](https://laravel.com/docs/8.x/structure#the-root-directory)
  + [The app Directory](https://laravel.com/docs/8.x/structure#the-root-app-directory)
  + [The bootstrap Directory](https://laravel.com/docs/8.x/structure#the-bootstrap-directory)
  + [The config Directory](https://laravel.com/docs/8.x/structure#the-config-directory)
  + [The database Directory](https://laravel.com/docs/8.x/structure#the-database-directory)
  + [The public Directory](https://laravel.com/docs/8.x/structure#the-public-directory)
  + [The resources Directory](https://laravel.com/docs/8.x/structure#the-resources-directory)
  + [The routes Directory](https://laravel.com/docs/8.x/structure#the-routes-directory)
  + [The storage Directory](https://laravel.com/docs/8.x/structure#the-storage-directory)
  + [The tests Directory](https://laravel.com/docs/8.x/structure#the-tests-directory)
  + [The vendor Directory](https://laravel.com/docs/8.x/structure#the-vendor-directory)
* [The App Directory](https://laravel.com/docs/8.x/structure#the-app-directory)
  + [The Broadcasting Directory](https://laravel.com/docs/8.x/structure#the-broadcasting-directory)
  + [The Console Directory](https://laravel.com/docs/8.x/structure#the-console-directory)
  + [The Events Directory](https://laravel.com/docs/8.x/structure#the-events-directory)
  + [The Exceptions Directory](https://laravel.com/docs/8.x/structure#the-exceptions-directory)
  + [The Http Directory](https://laravel.com/docs/8.x/structure#the-http-directory)
  + [The Jobs Directory](https://laravel.com/docs/8.x/structure#the-jobs-directory)
  + [The Listeners Directory](https://laravel.com/docs/8.x/structure#the-listeners-directory)
  + [The Mail Directory](https://laravel.com/docs/8.x/structure#the-mail-directory)
  + [The Models Directory](https://laravel.com/docs/8.x/structure#the-models-directory)
  + [The Notifications Directory](https://laravel.com/docs/8.x/structure#the-notifications-directory)
  + [The Policies Directory](https://laravel.com/docs/8.x/structure#the-policies-directory)
  + [The Providers Directory](https://laravel.com/docs/8.x/structure#the-providers-directory)
  + [The Rules Directory](https://laravel.com/docs/8.x/structure#the-rules-directory)

[Introduction](https://laravel.com/docs/8.x/structure#introduction)

The default Laravel application structure is intended to provide a great starting point for both large and small applications. But you are free to organize your application however you like. Laravel imposes almost no restrictions on where any given class is located - as long as Composer can autoload the class.

[The Root Directory](https://laravel.com/docs/8.x/structure#the-root-directory)

[The App Directory](https://laravel.com/docs/8.x/structure#the-root-app-directory)

The app directory contains the core code of your application. We'll explore this directory in more detail soon; however, almost all of the classes in your application will be in this directory.

[The Bootstrap Directory](https://laravel.com/docs/8.x/structure#the-bootstrap-directory)

The bootstrap directory contains the app.php file which bootstraps the framework. This directory also houses a cache directory which contains framework generated files for performance optimization such as the route and services cache files. You should not typically need to modify any files within this directory.

[The Config Directory](https://laravel.com/docs/8.x/structure#the-config-directory)

The config directory, as the name implies, contains all of your application's configuration files. It's a great idea to read through all of these files and familiarize yourself with all of the options available to you.

[The Database Directory](https://laravel.com/docs/8.x/structure#the-database-directory)

The database directory contains your database migrations, model factories, and seeds. If you wish, you may also use this directory to hold an SQLite database.

[The Public Directory](https://laravel.com/docs/8.x/structure#the-public-directory)

The public directory contains the index.php file, which is the entry point for all requests entering your application and configures autoloading. This directory also houses your assets such as images, JavaScript, and CSS.

[The Resources Directory](https://laravel.com/docs/8.x/structure#the-resources-directory)

The resources directory contains your [views](https://laravel.com/docs/8.x/views) as well as your raw, un-compiled assets such as CSS or JavaScript. This directory also houses all of your language files.

[The Routes Directory](https://laravel.com/docs/8.x/structure#the-routes-directory)

The routes directory contains all of the route definitions for your application. By default, several route files are included with Laravel: web.php, api.php, console.php, and channels.php.

The web.php file contains routes that the RouteServiceProvider places in the web middleware group, which provides session state, CSRF protection, and cookie encryption. If your application does not offer a stateless, RESTful API then it is likely that all of your routes will most likely be defined in the web.php file.

The api.php file contains routes that the RouteServiceProvider places in the api middleware group. These routes are intended to be stateless, so requests entering the application through these routes are intended to be authenticated [via tokens](https://laravel.com/docs/8.x/sanctum) and will not have access to session state.

The console.php file is where you may define all of your closure based console commands. Each closure is bound to a command instance allowing a simple approach to interacting with each command's IO methods. Even though this file does not define HTTP routes, it defines console based entry points (routes) into your application.

The channels.php file is where you may register all of the [event broadcasting](https://laravel.com/docs/8.x/broadcasting) channels that your application supports.

[The Storage Directory](https://laravel.com/docs/8.x/structure#the-storage-directory)

The storage directory contains your logs, compiled Blade templates, file based sessions, file caches, and other files generated by the framework. This directory is segregated into app, framework, and logs directories. The app directory may be used to store any files generated by your application. The framework directory is used to store framework generated files and caches. Finally, the logs directory contains your application's log files.

The storage/app/public directory may be used to store user-generated files, such as profile avatars, that should be publicly accessible. You should create a symbolic link at public/storage which points to this directory. You may create the link using the php artisan storage:link Artisan command.

[The Tests Directory](https://laravel.com/docs/8.x/structure#the-tests-directory)

The tests directory contains your automated tests. Example [PHPUnit](https://phpunit.de/) unit tests and feature tests are provided out of the box. Each test class should be suffixed with the word Test. You may run your tests using the phpunit or php vendor/bin/phpunit commands. Or, if you would like a more detailed and beautiful representation of your test results, you may run your tests using the php artisan test Artisan command.

[The Vendor Directory](https://laravel.com/docs/8.x/structure#the-vendor-directory)

The vendor directory contains your [Composer](https://getcomposer.org/) dependencies.

[The App Directory](https://laravel.com/docs/8.x/structure#the-app-directory)

The majority of your application is housed in the app directory. By default, this directory is namespaced under App and is autoloaded by Composer using the [PSR-4 autoloading standard](https://www.php-fig.org/psr/psr-4/).

The app directory contains a variety of additional directories such as Console, Http, and Providers. Think of the Console and Http directories as providing an API into the core of your application. The HTTP protocol and CLI are both mechanisms to interact with your application, but do not actually contain application logic. In other words, they are two ways of issuing commands to your application. The Console directory contains all of your Artisan commands, while the Http directory contains your controllers, middleware, and requests.

A variety of other directories will be generated inside the app directory as you use the make Artisan commands to generate classes. So, for example, the app/Jobs directory will not exist until you execute the make:job Artisan command to generate a job class.

Many of the classes in the app directory can be generated by Artisan via commands. To review the available commands, run the php artisan list make command in your terminal.

[The Broadcasting Directory](https://laravel.com/docs/8.x/structure#the-broadcasting-directory)

The Broadcasting directory contains all of the broadcast channel classes for your application. These classes are generated using the make:channel command. This directory does not exist by default, but will be created for you when you create your first channel. To learn more about channels, check out the documentation on [event broadcasting](https://laravel.com/docs/8.x/broadcasting).

[The Console Directory](https://laravel.com/docs/8.x/structure#the-console-directory)

The Console directory contains all of the custom Artisan commands for your application. These commands may be generated using the make:command command. This directory also houses your console kernel, which is where your custom Artisan commands are registered and your [scheduled tasks](https://laravel.com/docs/8.x/scheduling) are defined.

[The Events Directory](https://laravel.com/docs/8.x/structure#the-events-directory)

This directory does not exist by default, but will be created for you by the event:generate and make:event Artisan commands. The Events directory houses [event classes](https://laravel.com/docs/8.x/events). Events may be used to alert other parts of your application that a given action has occurred, providing a great deal of flexibility and decoupling.

[The Exceptions Directory](https://laravel.com/docs/8.x/structure#the-exceptions-directory)

The Exceptions directory contains your application's exception handler and is also a good place to place any exceptions thrown by your application. If you would like to customize how your exceptions are logged or rendered, you should modify the Handler class in this directory.

[The Http Directory](https://laravel.com/docs/8.x/structure#the-http-directory)

The Http directory contains your controllers, middleware, and form requests. Almost all of the logic to handle requests entering your application will be placed in this directory.

[The Jobs Directory](https://laravel.com/docs/8.x/structure#the-jobs-directory)

This directory does not exist by default, but will be created for you if you execute the make:job Artisan command. The Jobs directory houses the [queueable jobs](https://laravel.com/docs/8.x/queues) for your application. Jobs may be queued by your application or run synchronously within the current request lifecycle. Jobs that run synchronously during the current request are sometimes referred to as "commands" since they are an implementation of the [command pattern](https://en.wikipedia.org/wiki/Command_pattern).

[The Listeners Directory](https://laravel.com/docs/8.x/structure#the-listeners-directory)

This directory does not exist by default, but will be created for you if you execute the event:generate or make:listener Artisan commands. The Listeners directory contains the classes that handle your [events](https://laravel.com/docs/8.x/events). Event listeners receive an event instance and perform logic in response to the event being fired. For example, a UserRegistered event might be handled by a SendWelcomeEmail listener.

[The Mail Directory](https://laravel.com/docs/8.x/structure#the-mail-directory)

This directory does not exist by default, but will be created for you if you execute the make:mail Artisan command. The Mail directory contains all of your [classes that represent emails](https://laravel.com/docs/8.x/mail) sent by your application. Mail objects allow you to encapsulate all of the logic of building an email in a single, simple class that may be sent using the Mail::send method.

[The Models Directory](https://laravel.com/docs/8.x/structure#the-models-directory)

The Models directory contains all of your [Eloquent model classes](https://laravel.com/docs/8.x/eloquent). The Eloquent ORM included with Laravel provides a beautiful, simple ActiveRecord implementation for working with your database. Each database table has a corresponding "Model" which is used to interact with that table. Models allow you to query for data in your tables, as well as insert new records into the table.

[The Notifications Directory](https://laravel.com/docs/8.x/structure#the-notifications-directory)

This directory does not exist by default, but will be created for you if you execute the make:notification Artisan command. The Notifications directory contains all of the "transactional" [notifications](https://laravel.com/docs/8.x/notifications) that are sent by your application, such as simple notifications about events that happen within your application. Laravel's notification feature abstracts sending notifications over a variety of drivers such as email, Slack, SMS, or stored in a database.

[The Policies Directory](https://laravel.com/docs/8.x/structure#the-policies-directory)

This directory does not exist by default, but will be created for you if you execute the make:policy Artisan command. The Policies directory contains the [authorization policy classes](https://laravel.com/docs/8.x/authorization) for your application. Policies are used to determine if a user can perform a given action against a resource.

[The Providers Directory](https://laravel.com/docs/8.x/structure#the-providers-directory)

The Providers directory contains all of the [service providers](https://laravel.com/docs/8.x/providers) for your application. Service providers bootstrap your application by binding services in the service container, registering events, or performing any other tasks to prepare your application for incoming requests.

In a fresh Laravel application, this directory will already contain several providers. You are free to add your own providers to this directory as needed.

[The Rules Directory](https://laravel.com/docs/8.x/structure#the-rules-directory)

This directory does not exist by default, but will be created for you if you execute the make:rule Artisan command. The Rules directory contains the custom validation rule objects for your application. Rules are used to encapsulate complicated validation logic in a simple object. For more information, check out the [validation documentation](https://laravel.com/docs/8.x/validation).

Starter Kits

* [Introduction](https://laravel.com/docs/8.x/starter-kits#introduction)
* [Laravel Breeze](https://laravel.com/docs/8.x/starter-kits#laravel-breeze)
  + [Installation](https://laravel.com/docs/8.x/starter-kits#laravel-breeze-installation)
  + [Breeze & Inertia](https://laravel.com/docs/8.x/starter-kits#breeze-and-inertia)
* [Laravel Jetstream](https://laravel.com/docs/8.x/starter-kits#laravel-jetstream)

[Introduction](https://laravel.com/docs/8.x/starter-kits#introduction)

To give you a head start building your new Laravel application, we are happy to offer authentication and application starter kits. These kits automatically scaffold your application with the routes, controllers, and views you need to register and authenticate your application's users.

While you are welcome to use these starter kits, they are not required. You are free to build your own application from the ground up by simply installing a fresh copy of Laravel. Either way, we know you will build something great!

[Laravel Breeze](https://laravel.com/docs/8.x/starter-kits#laravel-breeze)

Laravel Breeze is a minimal, simple implementation of all of Laravel's [authentication features](https://laravel.com/docs/8.x/authentication), including login, registration, password reset, email verification, and password confirmation. Laravel Breeze's default view layer is made up of simple [Blade templates](https://laravel.com/docs/8.x/blade) styled with [Tailwind CSS](https://tailwindcss.com/). Breeze provides a wonderful starting point for beginning a fresh Laravel application.

[Installation](https://laravel.com/docs/8.x/starter-kits#laravel-breeze-installation)

First, you should [create a new Laravel application](https://laravel.com/docs/8.x/installation), configure your database, and run your [database migrations](https://laravel.com/docs/8.x/migrations):

curl -s https://laravel.build/example-app | bash

cd example-app

php artisan migrate

Once you have created a new Laravel application, you may install Laravel Breeze using Composer:

composer require laravel/breeze --dev

After Composer has installed the Laravel Breeze package, you may run the breeze:install Artisan command. This command publishes the authentication views, routes, controllers, and other resources to your application. Laravel Breeze publishes all of its code to your application so that you have full control and visibility over its features and implementation. After Breeze is installed, you should also compile your assets so that your application's CSS file is available:

php artisan breeze:install

npm install

npm run dev

php artisan migrate

Next, you may navigate to your application's /login or /register URLs in your web browser. All of Breeze's routes are defined within the routes/auth.php file.

To learn more about compiling your application's CSS and JavaScript, check out the [Laravel Mix documentation](https://laravel.com/docs/8.x/mix#running-mix).

[Breeze & Inertia](https://laravel.com/docs/8.x/starter-kits#breeze-and-inertia)

Laravel Breeze also offers an [Inertia.js](https://inertiajs.com/) frontend implementation powered by Vue or React. To use an Inertia stack, specify vue or react as your desired stack when executing the breeze:install Artisan command:

php artisan breeze:install vue

// Or...

php artisan breeze:install react

npm install

npm run dev

php artisan migrate

[Laravel Jetstream](https://laravel.com/docs/8.x/starter-kits#laravel-jetstream)

While Laravel Breeze provides a simple and minimal starting point for building a Laravel application, Jetstream augments that functionality with more robust features and additional frontend technology stacks. For those brand new to Laravel, we recommend learning the ropes with Laravel Breeze before graduating to Laravel Jetstream.

Jetstream provides a beautifully designed application scaffolding for Laravel and includes login, registration, email verification, two-factor authentication, session management, API support via Laravel Sanctum, and optional team management. Jetstream is designed using [Tailwind CSS](https://tailwindcss.com/) and offers your choice of [Livewire](https://laravel-livewire.com/) or [Inertia.js](https://inertiajs.com/) driven frontend scaffolding.

Complete documentation for installing Laravel Jetstream can be found within the [official Jetstream documentation](https://jetstream.laravel.com/2.x/introduction.html).

Deployment

* [Introduction](https://laravel.com/docs/8.x/deployment#introduction)
* [Server Requirements](https://laravel.com/docs/8.x/deployment#server-requirements)
* [Server Configuration](https://laravel.com/docs/8.x/deployment#server-configuration)
  + [Nginx](https://laravel.com/docs/8.x/deployment#nginx)
* [Optimization](https://laravel.com/docs/8.x/deployment#optimization)
  + [Autoloader Optimization](https://laravel.com/docs/8.x/deployment#autoloader-optimization)
  + [Optimizing Configuration Loading](https://laravel.com/docs/8.x/deployment#optimizing-configuration-loading)
  + [Optimizing Route Loading](https://laravel.com/docs/8.x/deployment#optimizing-route-loading)
  + [Optimizing View Loading](https://laravel.com/docs/8.x/deployment#optimizing-view-loading)
* [Debug Mode](https://laravel.com/docs/8.x/deployment#debug-mode)
* [Deploying With Forge / Vapor](https://laravel.com/docs/8.x/deployment#deploying-with-forge-or-vapor)

[Introduction](https://laravel.com/docs/8.x/deployment#introduction)

When you're ready to deploy your Laravel application to production, there are some important things you can do to make sure your application is running as efficiently as possible. In this document, we'll cover some great starting points for making sure your Laravel application is deployed properly.

[Server Requirements](https://laravel.com/docs/8.x/deployment#server-requirements)

The Laravel framework has a few system requirements. You should ensure that your web server has the following minimum PHP version and extensions:

* PHP >= 7.3
* BCMath PHP Extension
* Ctype PHP Extension
* Fileinfo PHP Extension
* JSON PHP Extension
* Mbstring PHP Extension
* OpenSSL PHP Extension
* PDO PHP Extension
* Tokenizer PHP Extension
* XML PHP Extension

[Server Configuration](https://laravel.com/docs/8.x/deployment#server-configuration)

[Nginx](https://laravel.com/docs/8.x/deployment#nginx)

If you are deploying your application to a server that is running Nginx, you may use the following configuration file as a starting point for configuring your web server. Most likely, this file will need to be customized depending on your server's configuration. If you would like assistance in managing your server, consider using a first-party Laravel server management and deployment service such as [Laravel Forge](https://forge.laravel.com/).

Please ensure, like the configuration below, your web server directs all requests to your application's public/index.php file. You should never attempt to move the index.php file to your project's root, as serving the application from the project root will expose many sensitive configuration files to the public Internet:

server {

listen 80;

server\_name example.com;

root /srv/example.com/public;

add\_header X-Frame-Options "SAMEORIGIN";

add\_header X-Content-Type-Options "nosniff";

index index.php;

charset utf-8;

location / {

try\_files $uri $uri/ /index.php?$query\_string;

}

location = /favicon.ico { access\_log off; log\_not\_found off; }

location = /robots.txt { access\_log off; log\_not\_found off; }

error\_page 404 /index.php;

location ~ \.php$ {

fastcgi\_pass unix:/var/run/php/php7.4-fpm.sock;

fastcgi\_param SCRIPT\_FILENAME $realpath\_root$fastcgi\_script\_name;

include fastcgi\_params;

}

location ~ /\.(?!well-known).\* {

deny all;

}

}

[Optimization](https://laravel.com/docs/8.x/deployment#optimization)

[Autoloader Optimization](https://laravel.com/docs/8.x/deployment#autoloader-optimization)

When deploying to production, make sure that you are optimizing Composer's class autoloader map so Composer can quickly find the proper file to load for a given class:

composer install --optimize-autoloader --no-dev

In addition to optimizing the autoloader, you should always be sure to include a composer.lock file in your project's source control repository. Your project's dependencies can be installed much faster when a composer.lock file is present.

[Optimizing Configuration Loading](https://laravel.com/docs/8.x/deployment#optimizing-configuration-loading)

When deploying your application to production, you should make sure that you run the config:cache Artisan command during your deployment process:

php artisan config:cache

This command will combine all of Laravel's configuration files into a single, cached file, which greatly reduces the number of trips the framework must make to the filesystem when loading your configuration values.

If you execute the config:cache command during your deployment process, you should be sure that you are only calling the env function from within your configuration files. Once the configuration has been cached, the .env file will not be loaded and all calls to the env function for .env variables will return null.

[Optimizing Route Loading](https://laravel.com/docs/8.x/deployment#optimizing-route-loading)

If you are building a large application with many routes, you should make sure that you are running the route:cache Artisan command during your deployment process:

php artisan route:cache

This command reduces all of your route registrations into a single method call within a cached file, improving the performance of route registration when registering hundreds of routes.

[Optimizing View Loading](https://laravel.com/docs/8.x/deployment#optimizing-view-loading)

When deploying your application to production, you should make sure that you run the view:cache Artisan command during your deployment process:

php artisan view:cache

This command precompiles all your Blade views so they are not compiled on demand, improving the performance of each request that returns a view.

[Debug Mode](https://laravel.com/docs/8.x/deployment#debug-mode)

The debug option in your config/app.php configuration file determines how much information about an error is actually displayed to the user. By default, this option is set to respect the value of the APP\_DEBUG environment variable, which is stored in your .env file.

In your production environment, this value should always be false. If the APP\_DEBUG variable is set to true in production, you risk exposing sensitive configuration values to your application's end users.

[Deploying With Forge / Vapor](https://laravel.com/docs/8.x/deployment#deploying-with-forge-or-vapor)

[Laravel Forge](https://laravel.com/docs/8.x/deployment#laravel-forge)

If you aren't quite ready to manage your own server configuration or aren't comfortable configuring all of the various services needed to run a robust Laravel application, [Laravel Forge](https://forge.laravel.com/) is a wonderful alternative.

Laravel Forge can create servers on various infrastructure providers such as DigitalOcean, Linode, AWS, and more. In addition, Forge installs and manages all of the tools needed to build robust Laravel applications, such as Nginx, MySQL, Redis, Memcached, Beanstalk, and more.

[Laravel Vapor](https://laravel.com/docs/8.x/deployment#laravel-vapor)

If you would like a totally serverless, auto-scaling deployment platform tuned for Laravel, check out [Laravel Vapor](https://vapor.laravel.com/). Laravel Vapor is a serverless deployment platform for Laravel, powered by AWS. Launch your Laravel infrastructure on Vapor and fall in love with the scalable simplicity of serverless. Laravel Vapor is fine-tuned by Laravel's creators to work seamlessly with the framework so you can keep writing your Laravel applications exactly like you're used to.

Deployment

* [Introduction](https://laravel.com/docs/8.x/deployment#introduction)
* [Server Requirements](https://laravel.com/docs/8.x/deployment#server-requirements)
* [Server Configuration](https://laravel.com/docs/8.x/deployment#server-configuration)
  + [Nginx](https://laravel.com/docs/8.x/deployment#nginx)
* [Optimization](https://laravel.com/docs/8.x/deployment#optimization)
  + [Autoloader Optimization](https://laravel.com/docs/8.x/deployment#autoloader-optimization)
  + [Optimizing Configuration Loading](https://laravel.com/docs/8.x/deployment#optimizing-configuration-loading)
  + [Optimizing Route Loading](https://laravel.com/docs/8.x/deployment#optimizing-route-loading)
  + [Optimizing View Loading](https://laravel.com/docs/8.x/deployment#optimizing-view-loading)
* [Debug Mode](https://laravel.com/docs/8.x/deployment#debug-mode)
* [Deploying With Forge / Vapor](https://laravel.com/docs/8.x/deployment#deploying-with-forge-or-vapor)

[Introduction](https://laravel.com/docs/8.x/deployment#introduction)

When you're ready to deploy your Laravel application to production, there are some important things you can do to make sure your application is running as efficiently as possible. In this document, we'll cover some great starting points for making sure your Laravel application is deployed properly.

[Server Requirements](https://laravel.com/docs/8.x/deployment#server-requirements)

The Laravel framework has a few system requirements. You should ensure that your web server has the following minimum PHP version and extensions:

* PHP >= 7.3
* BCMath PHP Extension
* Ctype PHP Extension
* Fileinfo PHP Extension
* JSON PHP Extension
* Mbstring PHP Extension
* OpenSSL PHP Extension
* PDO PHP Extension
* Tokenizer PHP Extension
* XML PHP Extension

[Server Configuration](https://laravel.com/docs/8.x/deployment#server-configuration)

[Nginx](https://laravel.com/docs/8.x/deployment#nginx)

If you are deploying your application to a server that is running Nginx, you may use the following configuration file as a starting point for configuring your web server. Most likely, this file will need to be customized depending on your server's configuration. If you would like assistance in managing your server, consider using a first-party Laravel server management and deployment service such as [Laravel Forge](https://forge.laravel.com/).

Please ensure, like the configuration below, your web server directs all requests to your application's public/index.php file. You should never attempt to move the index.php file to your project's root, as serving the application from the project root will expose many sensitive configuration files to the public Internet:

server {

listen 80;

server\_name example.com;

root /srv/example.com/public;

add\_header X-Frame-Options "SAMEORIGIN";

add\_header X-Content-Type-Options "nosniff";

index index.php;

charset utf-8;

location / {

try\_files $uri $uri/ /index.php?$query\_string;

}

location = /favicon.ico { access\_log off; log\_not\_found off; }

location = /robots.txt { access\_log off; log\_not\_found off; }

error\_page 404 /index.php;

location ~ \.php$ {

fastcgi\_pass unix:/var/run/php/php7.4-fpm.sock;

fastcgi\_param SCRIPT\_FILENAME $realpath\_root$fastcgi\_script\_name;

include fastcgi\_params;

}

location ~ /\.(?!well-known).\* {

deny all;

}

}

[Optimization](https://laravel.com/docs/8.x/deployment#optimization)

[Autoloader Optimization](https://laravel.com/docs/8.x/deployment#autoloader-optimization)

When deploying to production, make sure that you are optimizing Composer's class autoloader map so Composer can quickly find the proper file to load for a given class:

composer install --optimize-autoloader --no-dev

In addition to optimizing the autoloader, you should always be sure to include a composer.lock file in your project's source control repository. Your project's dependencies can be installed much faster when a composer.lock file is present.

[Optimizing Configuration Loading](https://laravel.com/docs/8.x/deployment#optimizing-configuration-loading)

When deploying your application to production, you should make sure that you run the config:cache Artisan command during your deployment process:

php artisan config:cache

This command will combine all of Laravel's configuration files into a single, cached file, which greatly reduces the number of trips the framework must make to the filesystem when loading your configuration values.

If you execute the config:cache command during your deployment process, you should be sure that you are only calling the env function from within your configuration files. Once the configuration has been cached, the .env file will not be loaded and all calls to the env function for .env variables will return null.

[Optimizing Route Loading](https://laravel.com/docs/8.x/deployment#optimizing-route-loading)

If you are building a large application with many routes, you should make sure that you are running the route:cache Artisan command during your deployment process:

php artisan route:cache

This command reduces all of your route registrations into a single method call within a cached file, improving the performance of route registration when registering hundreds of routes.

[Optimizing View Loading](https://laravel.com/docs/8.x/deployment#optimizing-view-loading)

When deploying your application to production, you should make sure that you run the view:cache Artisan command during your deployment process:

php artisan view:cache

This command precompiles all your Blade views so they are not compiled on demand, improving the performance of each request that returns a view.

[Debug Mode](https://laravel.com/docs/8.x/deployment#debug-mode)

The debug option in your config/app.php configuration file determines how much information about an error is actually displayed to the user. By default, this option is set to respect the value of the APP\_DEBUG environment variable, which is stored in your .env file.

In your production environment, this value should always be false. If the APP\_DEBUG variable is set to true in production, you risk exposing sensitive configuration values to your application's end users.

[Deploying With Forge / Vapor](https://laravel.com/docs/8.x/deployment#deploying-with-forge-or-vapor)

[Laravel Forge](https://laravel.com/docs/8.x/deployment#laravel-forge)

If you aren't quite ready to manage your own server configuration or aren't comfortable configuring all of the various services needed to run a robust Laravel application, [Laravel Forge](https://forge.laravel.com/) is a wonderful alternative.

Laravel Forge can create servers on various infrastructure providers such as DigitalOcean, Linode, AWS, and more. In addition, Forge installs and manages all of the tools needed to build robust Laravel applications, such as Nginx, MySQL, Redis, Memcached, Beanstalk, and more.

[Laravel Vapor](https://laravel.com/docs/8.x/deployment#laravel-vapor)

If you would like a totally serverless, auto-scaling deployment platform tuned for Laravel, check out [Laravel Vapor](https://vapor.laravel.com/). Laravel Vapor is a serverless deployment platform for Laravel, powered by AWS. Launch your Laravel infrastructure on Vapor and fall in love with the scalable simplicity of serverless. Laravel Vapor is fine-tuned by Laravel's creators to work seamlessly with the framework so you can keep writing your Laravel applications exactly like you're used to.

Request Lifecycle

* [Introduction](https://laravel.com/docs/8.x/lifecycle#introduction)
* [Lifecycle Overview](https://laravel.com/docs/8.x/lifecycle#lifecycle-overview)
  + [First Steps](https://laravel.com/docs/8.x/lifecycle#first-steps)
  + [HTTP / Console Kernels](https://laravel.com/docs/8.x/lifecycle#http-console-kernels)
  + [Service Providers](https://laravel.com/docs/8.x/lifecycle#service-providers)
  + [Routing](https://laravel.com/docs/8.x/lifecycle#routing)
  + [Finishing Up](https://laravel.com/docs/8.x/lifecycle#finishing-up)
* [Focus On Service Providers](https://laravel.com/docs/8.x/lifecycle#focus-on-service-providers)

[Introduction](https://laravel.com/docs/8.x/lifecycle#introduction)

When using any tool in the "real world", you feel more confident if you understand how that tool works. Application development is no different. When you understand how your development tools function, you feel more comfortable and confident using them.

The goal of this document is to give you a good, high-level overview of how the Laravel framework works. By getting to know the overall framework better, everything feels less "magical" and you will be more confident building your applications. If you don't understand all of the terms right away, don't lose heart! Just try to get a basic grasp of what is going on, and your knowledge will grow as you explore other sections of the documentation.

[Lifecycle Overview](https://laravel.com/docs/8.x/lifecycle#lifecycle-overview)

[First Steps](https://laravel.com/docs/8.x/lifecycle#first-steps)

The entry point for all requests to a Laravel application is the public/index.php file. All requests are directed to this file by your web server (Apache / Nginx) configuration. The index.php file doesn't contain much code. Rather, it is a starting point for loading the rest of the framework.

The index.php file loads the Composer generated autoloader definition, and then retrieves an instance of the Laravel application from bootstrap/app.php. The first action taken by Laravel itself is to create an instance of the application / [service container](https://laravel.com/docs/8.x/container).

[HTTP / Console Kernels](https://laravel.com/docs/8.x/lifecycle#http-console-kernels)

Next, the incoming request is sent to either the HTTP kernel or the console kernel, depending on the type of request that is entering the application. These two kernels serve as the central location that all requests flow through. For now, let's just focus on the HTTP kernel, which is located in app/Http/Kernel.php.

The HTTP kernel extends the Illuminate\Foundation\Http\Kernel class, which defines an array of bootstrappers that will be run before the request is executed. These bootstrappers configure error handling, configure logging, [detect the application environment](https://laravel.com/docs/8.x/configuration#environment-configuration), and perform other tasks that need to be done before the request is actually handled. Typically, these classes handle internal Laravel configuration that you do not need to worry about.

The HTTP kernel also defines a list of HTTP [middleware](https://laravel.com/docs/8.x/middleware) that all requests must pass through before being handled by the application. These middleware handle reading and writing the [HTTP session](https://laravel.com/docs/8.x/session), determining if the application is in maintenance mode, [verifying the CSRF token](https://laravel.com/docs/8.x/csrf), and more. We'll talk more about these soon.

The method signature for the HTTP kernel's handle method is quite simple: it receives a Request and returns a Response. Think of the kernel as being a big black box that represents your entire application. Feed it HTTP requests and it will return HTTP responses.

[Service Providers](https://laravel.com/docs/8.x/lifecycle#service-providers)

One of the most important kernel bootstrapping actions is loading the [service providers](https://laravel.com/docs/8.x/providers) for your application. All of the service providers for the application are configured in the config/app.php configuration file's providers array.

Laravel will iterate through this list of providers and instantiate each of them. After instantiating the providers, the register method will be called on all of the providers. Then, once all of the providers have been registered, the boot method will be called on each provider. This is so service providers may depend on every container binding being registered and available by the time their boot method is executed.

Service providers are responsible for bootstrapping all of the framework's various components, such as the database, queue, validation, and routing components. Essentially every major feature offered by Laravel is bootstrapped and configured by a service provider. Since they bootstrap and configure so many features offered by the framework, service providers are the most important aspect of the entire Laravel bootstrap process.

[Routing](https://laravel.com/docs/8.x/lifecycle#routing)

One of the most important service providers in your application is the App\Providers\RouteServiceProvider. This service provider loads the route files contained within your application's routes directory. Go ahead, crack open the RouteServiceProvider code and take a look at how it works!

Once the application has been bootstrapped and all service providers have been registered, the Request will be handed off to the router for dispatching. The router will dispatch the request to a route or controller, as well as run any route specific middleware.

Middleware provide a convenient mechanism for filtering or examining HTTP requests entering your application. For example, Laravel includes a middleware that verifies if 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. Some middleware are assigned to all routes within the application, like those defined in the $middleware property of your HTTP kernel, while some are only assigned to specific routes or route groups. You can learn more about middleware by reading the complete [middleware documentation](https://laravel.com/docs/8.x/middleware).

If the request passes through all of the matched route's assigned middleware, the route or controller method will be executed and the response returned by the route or controller method will be sent back through the route's chain of middleware.

[Finishing Up](https://laravel.com/docs/8.x/lifecycle#finishing-up)

Once the route or controller method returns a response, the response will travel back outward through the route's middleware, giving the application a chance to modify or examine the outgoing response.

Finally, once the response travels back through the middleware, the HTTP kernel's handle method returns the response object and the index.php file calls the send method on the returned response. The send method sends the response content to the user's web browser. We've finished our journey through the entire Laravel request lifecycle!

[Focus On Service Providers](https://laravel.com/docs/8.x/lifecycle#focus-on-service-providers)

Service providers are truly the key to bootstrapping a Laravel application. The application instance is created, the service providers are registered, and the request is handed to the bootstrapped application. It's really that simple!

Having a firm grasp of how a Laravel application is built and bootstrapped via service providers is very valuable. Your application's default service providers are stored in the app/Providers directory.

By default, the AppServiceProvider is fairly empty. This provider is a great place to add your application's own bootstrapping and service container bindings. For large applications, you may wish to create several service providers, each with more granular bootstrapping for specific services used by your application.

Service Container

* [Introduction](https://laravel.com/docs/8.x/container#introduction)
  + [Zero Configuration Resolution](https://laravel.com/docs/8.x/container#zero-configuration-resolution)
  + [When To Use The Container](https://laravel.com/docs/8.x/container#when-to-use-the-container)
* [Binding](https://laravel.com/docs/8.x/container#binding)
  + [Binding Basics](https://laravel.com/docs/8.x/container#binding-basics)
  + [Binding Interfaces To Implementations](https://laravel.com/docs/8.x/container#binding-interfaces-to-implementations)
  + [Contextual Binding](https://laravel.com/docs/8.x/container#contextual-binding)
  + [Binding Primitives](https://laravel.com/docs/8.x/container#binding-primitives)
  + [Binding Typed Variadics](https://laravel.com/docs/8.x/container#binding-typed-variadics)
  + [Tagging](https://laravel.com/docs/8.x/container#tagging)
  + [Extending Bindings](https://laravel.com/docs/8.x/container#extending-bindings)
* [Resolving](https://laravel.com/docs/8.x/container#resolving)
  + [The Make Method](https://laravel.com/docs/8.x/container#the-make-method)
  + [Automatic Injection](https://laravel.com/docs/8.x/container#automatic-injection)
* [Method Invocation & Injection](https://laravel.com/docs/8.x/container#method-invocation-and-injection)
* [Container Events](https://laravel.com/docs/8.x/container#container-events)
* [PSR-11](https://laravel.com/docs/8.x/container#psr-11)

[Introduction](https://laravel.com/docs/8.x/container#introduction)

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

Let's look at a simple example:

**<?php**

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;

use App\Repositories\UserRepository;

use App\Models\User;

class UserController extends Controller

{

/\*\*

\* The user repository implementation.

\*

\* @var UserRepository

\*/

protected $users;

/\*\*

\* Create a new controller instance.

\*

\* @param UserRepository $users

\* @return void

\*/

public function \_\_construct(UserRepository $users)

{

$this->users = $users;

}

/\*\*

\* Show the profile for the given user.

\*

\* @param int $id

\* @return Response

\*/

public function show($id)

{

$user = $this->users->find($id);

return view('user.profile', ['user' => $user]);

}

}

In this example, the UserController needs to retrieve users from a data source. So, we will inject a service that is able to retrieve users. In this context, our UserRepository most likely uses [Eloquent](https://laravel.com/docs/8.x/eloquent) to retrieve user information from the database. However, since the repository is injected, we are able to easily swap it out with another implementation. We are also able to easily "mock", or create a dummy implementation of the UserRepository when testing our application.

A deep understanding of the Laravel service container is essential to building a powerful, large application, as well as for contributing to the Laravel core itself.

[Zero Configuration Resolution](https://laravel.com/docs/8.x/container#zero-configuration-resolution)

If a class has no dependencies or only depends on other concrete classes (not interfaces), the container does not need to be instructed on how to resolve that class. For example, you may place the following code in your routes/web.php file:

**<?php**

class Service

{

//

}

Route::get('/', function (Service $service) {

die(get\_class($service));

});

In this example, hitting your application's / route will automatically resolve the Service class and inject it into your route's handler. This is game changing. It means you can develop your application and take advantage of dependency injection without worrying about bloated configuration files.

Thankfully, many of the classes you will be writing when building a Laravel application automatically receive their dependencies via the container, including [controllers](https://laravel.com/docs/8.x/controllers), [event listeners](https://laravel.com/docs/8.x/events), [middleware](https://laravel.com/docs/8.x/middleware), and more. Additionally, you may type-hint dependencies in the handle method of [queued jobs](https://laravel.com/docs/8.x/queues). Once you taste the power of automatic and zero configuration dependency injection it feels impossible to develop without it.

[When To Use The Container](https://laravel.com/docs/8.x/container#when-to-use-the-container)

Thanks to zero configuration resolution, you will often type-hint dependencies on routes, controllers, event listeners, and elsewhere without ever manually interacting with the container. For example, you might type-hint the Illuminate\Http\Request object on your route definition so that you can easily access the current request. Even though we never have to interact with the container to write this code, it is managing the injection of these dependencies behind the scenes:

use Illuminate\Http\Request;

Route::get('/', function (Request $request) {

// ...

});

In many cases, thanks to automatic dependency injection and [facades](https://laravel.com/docs/8.x/facades), you can build Laravel applications without ever manually binding or resolving anything from the container. So, when would you ever manually interact with the container? Let's examine two situations.

First, if you write a class that implements an interface and you wish to type-hint that interface on a route or class constructor, you must [tell the container how to resolve that interface](https://laravel.com/docs/8.x/container#binding-interfaces-to-implementations). Secondly, if you are [writing a Laravel package](https://laravel.com/docs/8.x/packages) that you plan to share with other Laravel developers, you may need to bind your package's services into the container.

[Binding](https://laravel.com/docs/8.x/container#binding)

[Binding Basics](https://laravel.com/docs/8.x/container#binding-basics)

[Simple Bindings](https://laravel.com/docs/8.x/container#simple-bindings)

Almost all of your service container bindings will be registered within [service providers](https://laravel.com/docs/8.x/providers), so most of these examples will demonstrate using the container in that context.

Within a service provider, you always have access to the container via the $this->app property. 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:

use App\Services\Transistor;

use App\Services\PodcastParser;

$this->app->bind(Transistor::class, function ($app) {

return new Transistor($app->make(PodcastParser::class));

});

Note 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.

As mentioned, you will typically be interacting with the container within service providers; however, if you would like to interact with the container outside of a service provider, you may do so via the App [facade](https://laravel.com/docs/8.x/facades):

use App\Services\Transistor;

use Illuminate\Support\Facades\App;

App::bind(Transistor::class, function ($app) {

// ...

});

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 on how to build these objects, since it can automatically resolve these objects using reflection.

[Binding A Singleton](https://laravel.com/docs/8.x/container#binding-a-singleton)

The singleton method binds a class or interface into the container that should only be resolved one time. Once a singleton binding is resolved, the same object instance will be returned on subsequent calls into the container:

use App\Services\Transistor;

use App\Services\PodcastParser;

$this->app->singleton(Transistor::class, function ($app) {

return new Transistor($app->make(PodcastParser::class));

});

[Binding Scoped Singletons](https://laravel.com/docs/8.x/container#binding-scoped)

The scoped method binds a class or interface into the container that should only be resolved one time within a given Laravel request / job lifecycle. While this method is similar to the singleton method, instances registered using the scoped method will be flushed whenever the Laravel application starts a new "lifecycle", such as when a [Laravel Octane](https://laravel.com/docs/8.x/octane) worker processes a new request or when a Laravel [queue worker](https://laravel.com/docs/8.x/queues) processes a new job:

use App\Services\Transistor;

use App\Services\PodcastParser;

$this->app->scoped(Transistor::class, function ($app) {

return new Transistor($app->make(PodcastParser::class));

});

[Binding Instances](https://laravel.com/docs/8.x/container#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:

use App\Services\Transistor;

use App\Services\PodcastParser;

$service = new Transistor(new PodcastParser);

$this->app->instance(Transistor::class, $service);

[Binding Interfaces To Implementations](https://laravel.com/docs/8.x/container#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:

use App\Contracts\EventPusher;

use App\Services\RedisEventPusher;

$this->app->bind(EventPusher::class, RedisEventPusher::class);

This statement 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 the constructor of a class that is resolved by the container. Remember, controllers, event listeners, middleware, and various other types of classes within Laravel applications are always resolved using the container:

use App\Contracts\EventPusher;

/\*\*

\* Create a new class instance.

\*

\* @param \App\Contracts\EventPusher $pusher

\* @return void

\*/

public function \_\_construct(EventPusher $pusher)

{

$this->pusher = $pusher;

}

[Contextual Binding](https://laravel.com/docs/8.x/container#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, two controllers may depend on different implementations of the Illuminate\Contracts\Filesystem\Filesystem [contract](https://laravel.com/docs/8.x/contracts). Laravel provides a simple, fluent interface for defining this behavior:

use App\Http\Controllers\PhotoController;

use App\Http\Controllers\UploadController;

use App\Http\Controllers\VideoController;

use Illuminate\Contracts\Filesystem\Filesystem;

use Illuminate\Support\Facades\Storage;

$this->app->when(PhotoController::class)

->needs(Filesystem::class)

->give(function () {

return Storage::disk('local');

});

$this->app->when([VideoController::class, UploadController::class])

->needs(Filesystem::class)

->give(function () {

return Storage::disk('s3');

});

[Binding Primitives](https://laravel.com/docs/8.x/container#binding-primitives)

Sometimes you may have a class that receives some injected classes, but also needs an injected primitive value such as an integer. You may easily use contextual binding to inject any value your class may need:

$this->app->when('App\Http\Controllers\UserController')

->needs('$variableName')

->give($value);

Sometimes a class may depend on an array of [tagged](https://laravel.com/docs/8.x/container#tagging) instances. Using the giveTagged method, you may easily inject all of the container bindings with that tag:

$this->app->when(ReportAggregator::class)

->needs('$reports')

->giveTagged('reports');

If you need to inject a value from one of your application's configuration files, you may use the giveConfig method:

$this->app->when(ReportAggregator::class)

->needs('$timezone')

->giveConfig('app.timezone');

[Binding Typed Variadics](https://laravel.com/docs/8.x/container#binding-typed-variadics)

Occasionally, you may have a class that receives an array of typed objects using a variadic constructor argument:

**<?php**

use App\Models\Filter;

use App\Services\Logger;

class Firewall

{

/\*\*

\* The logger instance.

\*

\* @var \App\Services\Logger

\*/

protected $logger;

/\*\*

\* The filter instances.

\*

\* @var array

\*/

protected $filters;

/\*\*

\* Create a new class instance.

\*

\* @param \App\Services\Logger $logger

\* @param array $filters

\* @return void

\*/

public function \_\_construct(Logger $logger, Filter ...$filters)

{

$this->logger = $logger;

$this->filters = $filters;

}

}

Using contextual binding, you may resolve this dependency by providing the give method with a closure that returns an array of resolved Filter instances:

$this->app->when(Firewall::class)

->needs(Filter::class)

->give(function ($app) {

return [

$app->make(NullFilter::class),

$app->make(ProfanityFilter::class),

$app->make(TooLongFilter::class),

];

});

For convenience, you may also just provide an array of class names to be resolved by the container whenever Firewall needs Filter instances:

$this->app->when(Firewall::class)

->needs(Filter::class)

->give([

NullFilter::class,

ProfanityFilter::class,

TooLongFilter::class,

]);

[Variadic Tag Dependencies](https://laravel.com/docs/8.x/container#variadic-tag-dependencies)

Sometimes a class may have a variadic dependency that is type-hinted as a given class (Report ...$reports). Using the needs and giveTagged methods, you may easily inject all of the container bindings with that [tag](https://laravel.com/docs/8.x/container#tagging) for the given dependency:

$this->app->when(ReportAggregator::class)

->needs(Report::class)

->giveTagged('reports');

[Tagging](https://laravel.com/docs/8.x/container#tagging)

Occasionally, you may need to resolve all of a certain "category" of binding. For example, perhaps you are building a report analyzer 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(CpuReport::class, function () {

//

});

$this->app->bind(MemoryReport::class, function () {

//

});

$this->app->tag([CpuReport::class, MemoryReport::class], 'reports');

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

$this->app->bind(ReportAnalyzer::class, function ($app) {

return new ReportAnalyzer($app->tagged('reports'));

});

[Extending Bindings](https://laravel.com/docs/8.x/container#extending-bindings)

The extend method allows the modification of resolved services. For example, when a service is resolved, you may run additional code to decorate or configure the service. The extend method accepts a closure, which should return the modified service, as its only argument. The closure receives the service being resolved and the container instance:

$this->app->extend(Service::class, function ($service, $app) {

return new DecoratedService($service);

});

[Resolving](https://laravel.com/docs/8.x/container#resolving)

[The make Method](https://laravel.com/docs/8.x/container#the-make-method)

You may use the make method to resolve a class instance from the container. The make method accepts the name of the class or interface you wish to resolve:

use App\Services\Transistor;

$transistor = $this->app->make(Transistor::class);

If some of your class' dependencies are not resolvable via the container, you may inject them by passing them as an associative array into the makeWith method. For example, we may manually pass the $id constructor argument required by the Transistor service:

use App\Services\Transistor;

$transistor = $this->app->makeWith(Transistor::class, ['id' => 1]);

If you are outside of a service provider in a location of your code that does not have access to the $app variable, you may use the App [facade](https://laravel.com/docs/8.x/facades) to resolve a class instance from the container:

use App\Services\Transistor;

use Illuminate\Support\Facades\App;

$transistor = App::make(Transistor::class);

If you would like to have the Laravel container instance itself injected into a class that is being resolved by the container, you may type-hint the Illuminate\Container\Container class on your class' constructor:

use Illuminate\Container\Container;

/\*\*

\* Create a new class instance.

\*

\* @param \Illuminate\Container\Container $container

\* @return void

\*/

public function \_\_construct(Container $container)

{

$this->container = $container;

}

[Automatic Injection](https://laravel.com/docs/8.x/container#automatic-injection)

Alternatively, and importantly, you may type-hint the dependency in the constructor of a class that is resolved by the container, including [controllers](https://laravel.com/docs/8.x/controllers), [event listeners](https://laravel.com/docs/8.x/events), [middleware](https://laravel.com/docs/8.x/middleware), and more. Additionally, you may type-hint dependencies in the handle method of [queued jobs](https://laravel.com/docs/8.x/queues). In practice, this is how most of your objects should be resolved by the container.

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\Repositories\UserRepository;

class UserController extends Controller

{

/\*\*

\* The user repository instance.

\*

\* @var \App\Repositories\UserRepository

\*/

protected $users;

/\*\*

\* Create a new controller instance.

\*

\* @param \App\Repositories\UserRepository $users

\* @return void

\*/

public function \_\_construct(UserRepository $users)

{

$this->users = $users;

}

/\*\*

\* Show the user with the given ID.

\*

\* @param int $id

\* @return \Illuminate\Http\Response

\*/

public function show($id)

{

//

}

}

[Method Invocation & Injection](https://laravel.com/docs/8.x/container#method-invocation-and-injection)

Sometimes you may wish to invoke a method on an object instance while allowing the container to automatically inject that method's dependencies. For example, given the following class:

**<?php**

namespace App;

use App\Repositories\UserRepository;

class UserReport

{

/\*\*

\* Generate a new user report.

\*

\* @param \App\Repositories\UserRepository $repository

\* @return array

\*/

public function generate(UserRepository $repository)

{

// ...

}

}

You may invoke the generate method via the container like so:

use App\UserReport;

use Illuminate\Support\Facades\App;

$report = App::call([new UserReport, 'generate']);

The call method accepts any PHP callable. The container's call method may even be used to invoke a closure while automatically injecting its dependencies:

use App\Repositories\UserRepository;

use Illuminate\Support\Facades\App;

$result = App::call(function (UserRepository $repository) {

// ...

});

[Container Events](https://laravel.com/docs/8.x/container#container-events)

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

use App\Services\Transistor;

$this->app->resolving(Transistor::class, function ($transistor, $app) {

// Called when container resolves objects of type "Transistor"...

});

$this->app->resolving(function ($object, $app) {

// Called when container resolves object of any type...

});

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.

[PSR-11](https://laravel.com/docs/8.x/container#psr-11)

Laravel's service container implements the [PSR-11](https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-11-container.md) interface. Therefore, you may type-hint the PSR-11 container interface to obtain an instance of the Laravel container:

use App\Services\Transistor;

use Psr\Container\ContainerInterface;

Route::get('/', function (ContainerInterface $container) {

$service = $container->get(Transistor::class);

//

});

An exception is thrown if the given identifier can't be resolved. The exception will be an instance of Psr\Container\NotFoundExceptionInterface if the identifier was never bound. If the identifier was bound but was unable to be resolved, an instance of Psr\Container\ContainerExceptionInterface will be thrown.

Service Providers

* [Introduction](https://laravel.com/docs/8.x/providers#introduction)
* [Writing Service Providers](https://laravel.com/docs/8.x/providers#writing-service-providers)
  + [The Register Method](https://laravel.com/docs/8.x/providers#the-register-method)
  + [The Boot Method](https://laravel.com/docs/8.x/providers#the-boot-method)
* [Registering Providers](https://laravel.com/docs/8.x/providers#registering-providers)
* [Deferred Providers](https://laravel.com/docs/8.x/providers#deferred-providers)

[Introduction](https://laravel.com/docs/8.x/providers#introduction)

Service providers are the central place of all Laravel application bootstrapping. Your own application, as well as all of Laravel'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 config/app.php file included with Laravel, you will see a providers array. These are all of the service provider classes that will be loaded for your application. By default, a set of Laravel core service providers are listed in this array. These providers bootstrap the core Laravel components, such as the mailer, queue, cache, and others. Many of these providers are "deferred" providers, meaning they will not be loaded on every request, but only when the services they provide are actually needed.

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

If you would like to learn more about how Laravel handles requests and works internally, check out our documentation on the Laravel [request lifecycle](https://laravel.com/docs/8.x/lifecycle).

[Writing Service Providers](https://laravel.com/docs/8.x/providers#writing-service-providers)

All service providers extend the Illuminate\Support\ServiceProvider class. Most service providers contain a register and a boot method. Within the register method, you should only bind things into the [service container](https://laravel.com/docs/8.x/container). You should never attempt to register any event listeners, routes, or any other piece of functionality within the register method.

The Artisan CLI can generate a new provider via the make:provider command:

php artisan make:provider RiakServiceProvider

[The Register Method](https://laravel.com/docs/8.x/providers#the-register-method)

As mentioned previously, within the register method, you should only bind things into the [service container](https://laravel.com/docs/8.x/container). You should never attempt to register any event listeners, routes, or any other piece of functionality within the register method. Otherwise, you may accidentally use a service that is provided by a service provider which has not loaded yet.

Let's take a look at a basic service provider. Within any of your service provider methods, you always have access to the $app property which provides access to the service container:

**<?php**

namespace App\Providers;

use App\Services\Riak\Connection;

use Illuminate\Support\ServiceProvider;

class RiakServiceProvider extends ServiceProvider

{

/\*\*

\* Register any application services.

\*

\* @return void

\*/

public function register()

{

$this->app->singleton(Connection::class, function ($app) {

return new Connection(config('riak'));

});

}

}

This service provider only defines a register method, and uses that method to define an implementation of App\Services\Riak\Connection in the service container. If you're not yet familiar with Laravel's service container, check out [its documentation](https://laravel.com/docs/8.x/container).

[The bindings And singletons Properties](https://laravel.com/docs/8.x/providers#the-bindings-and-singletons-properties)

If your service provider registers many simple bindings, you may wish to use the bindings and singletons properties instead of manually registering each container binding. When the service provider is loaded by the framework, it will automatically check for these properties and register their bindings:

**<?php**

namespace App\Providers;

use App\Contracts\DowntimeNotifier;

use App\Contracts\ServerProvider;

use App\Services\DigitalOceanServerProvider;

use App\Services\PingdomDowntimeNotifier;

use App\Services\ServerToolsProvider;

use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider

{

/\*\*

\* All of the container bindings that should be registered.

\*

\* @var array

\*/

public $bindings = [

ServerProvider::class => DigitalOceanServerProvider::class,

];

/\*\*

\* All of the container singletons that should be registered.

\*

\* @var array

\*/

public $singletons = [

DowntimeNotifier::class => PingdomDowntimeNotifier::class,

ServerProvider::class => ServerToolsProvider::class,

];

}

[The Boot Method](https://laravel.com/docs/8.x/providers#the-boot-method)

So, what if we need to register a [view composer](https://laravel.com/docs/8.x/views#view-composers) 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\Facades\View;

use Illuminate\Support\ServiceProvider;

class ComposerServiceProvider extends ServiceProvider

{

/\*\*

\* Bootstrap any application services.

\*

\* @return void

\*/

public function boot()

{

View::composer('view', function () {

//

});

}

}

[Boot Method Dependency Injection](https://laravel.com/docs/8.x/providers#boot-method-dependency-injection)

You may type-hint dependencies for your service provider's boot method. The [service container](https://laravel.com/docs/8.x/container) will automatically inject any dependencies you need:

use Illuminate\Contracts\Routing\ResponseFactory;

/\*\*

\* Bootstrap any application services.

\*

\* @param \Illuminate\Contracts\Routing\ResponseFactory $response

\* @return void

\*/

public function boot(ResponseFactory $response)

{

$response->macro('serialized', function ($value) {

//

});

}

[Registering Providers](https://laravel.com/docs/8.x/providers#registering-providers)

All service providers are registered in the config/app.php configuration file. This file contains a providers array where you can list the class names of your service providers. By default, a set of Laravel core service providers are listed in this array. These providers bootstrap the core Laravel components, such as the mailer, queue, cache, and others.

To register your provider, add it to the array:

'providers' => [

// Other Service Providers

App\Providers\ComposerServiceProvider::class,

],

[Deferred Providers](https://laravel.com/docs/8.x/providers#deferred-providers)

If your provider is only registering bindings in the [service container](https://laravel.com/docs/8.x/container), you may choose to defer its registration until one of the registered bindings is actually needed. Deferring the loading of such a provider will improve the performance of your application, since it is not loaded from the filesystem on every request.

Laravel compiles and stores a list of all of the services supplied by deferred service providers, along with the name of its service provider class. Then, only when you attempt to resolve one of these services does Laravel load the service provider.

To defer the loading of a provider, implement the \Illuminate\Contracts\Support\DeferrableProvider interface and define a provides method. The provides method should return the service container bindings registered by the provider:

**<?php**

namespace App\Providers;

use App\Services\Riak\Connection;

use Illuminate\Contracts\Support\DeferrableProvider;

use Illuminate\Support\ServiceProvider;

class RiakServiceProvider extends ServiceProvider implements DeferrableProvider

{

/\*\*

\* Register any application services.

\*

\* @return void

\*/

public function register()

{

$this->app->singleton(Connection::class, function ($app) {

return new Connection($app['config']['riak']);

});

}

/\*\*

\* Get the services provided by the provider.

\*

\* @return array

\*/

public function provides()

{

return [Connection::class];

}

}

# Facades

* [Introduction](https://laravel.com/docs/8.x/facades#introduction)
* [When To Use Facades](https://laravel.com/docs/8.x/facades#when-to-use-facades)
  + [Facades Vs. Dependency Injection](https://laravel.com/docs/8.x/facades#facades-vs-dependency-injection)
  + [Facades Vs. Helper Functions](https://laravel.com/docs/8.x/facades#facades-vs-helper-functions)
* [How Facades Work](https://laravel.com/docs/8.x/facades#how-facades-work)
* [Real-Time Facades](https://laravel.com/docs/8.x/facades#real-time-facades)
* [Facade Class Reference](https://laravel.com/docs/8.x/facades#facade-class-reference)

[Introduction](https://laravel.com/docs/8.x/facades#introduction)

Throughout the Laravel documentation, you will see examples of code that interacts with Laravel's features via "facades". Facades provide a "static" interface to classes that are available in the application's [service container](https://laravel.com/docs/8.x/container). Laravel ships with many facades which provide access to almost all of Laravel's features.

Laravel facades serve as "static proxies" to underlying classes in the service container, providing the benefit of a terse, expressive syntax while maintaining more testability and flexibility than traditional static methods. It's perfectly fine if you don't totally understand how facades work under the hood - just go with the flow and continue learning about Laravel.

All of Laravel's facades are defined in the Illuminate\Support\Facades namespace. So, we can easily access a facade like so:

use Illuminate\Support\Facades\Cache;

use Illuminate\Support\Facades\Route;

Route::get('/cache', function () {

return Cache::get('key');

});

Throughout the Laravel documentation, many of the examples will use facades to demonstrate various features of the framework.

[Helper Functions](https://laravel.com/docs/8.x/facades#helper-functions)

To complement facades, Laravel offers a variety of global "helper functions" that make it even easier to interact with common Laravel features. Some of the common helper functions you may interact with are view, response, url, config, and more. Each helper function offered by Laravel is documented with their corresponding feature; however, a complete list is available within the dedicated [helper documentation](https://laravel.com/docs/8.x/helpers).

For example, instead of using the Illuminate\Support\Facades\Response facade to generate a JSON response, we may simply use the response function. Because helper functions are globally available, you do not need to import any classes in order to use them:

use Illuminate\Support\Facades\Response;

Route::get('/users', function () {

return Response::json([

// ...

]);

});

Route::get('/users', function () {

return response()->json([

// ...

]);

});

[When To Use Facades](https://laravel.com/docs/8.x/facades#when-to-use-facades)

Facades have many benefits. They provide a terse, memorable syntax that allows you to use Laravel's features without remembering long class names that must be injected or configured manually. Furthermore, because of their unique usage of PHP's dynamic methods, they are easy to test.

However, some care must be taken when using facades. The primary danger of facades is class "scope creep". Since facades are so easy to use and do not require injection, it can be easy to let your classes continue to grow and use many facades in a single class. Using dependency injection, this potential is mitigated by the visual feedback a large constructor gives you that your class is growing too large. So, when using facades, pay special attention to the size of your class so that its scope of responsibility stays narrow. If your class is getting too large, consider splitting it into multiple smaller classes.

[Facades Vs. Dependency Injection](https://laravel.com/docs/8.x/facades#facades-vs-dependency-injection)

One of the primary benefits of dependency injection is the ability to swap implementations of the injected class. This is useful during testing since you can inject a mock or stub and assert that various methods were called on the stub.

Typically, it would not be possible to mock or stub a truly static class method. However, since facades use dynamic methods to proxy method calls to objects resolved from the service container, we actually can test facades just as we would test an injected class instance. For example, given the following route:

use Illuminate\Support\Facades\Cache;

Route::get('/cache', function () {

return Cache::get('key');

});

Using Laravel's facade testing methods, we can write the following test to verify that the Cache::get method was called with the argument we expected:

use Illuminate\Support\Facades\Cache;

/\*\*

\* A basic functional test example.

\*

\* @return void

\*/

public function testBasicExample()

{

Cache::shouldReceive('get')

->with('key')

->andReturn('value');

$response = $this->get('/cache');

$response->assertSee('value');

}

[Facades Vs. Helper Functions](https://laravel.com/docs/8.x/facades#facades-vs-helper-functions)

In addition to facades, Laravel includes a variety of "helper" functions which can perform common tasks like generating views, firing events, dispatching jobs, or sending HTTP responses. Many of these helper functions perform the same function as a corresponding facade. For example, this facade call and helper call are equivalent:

return Illuminate\Support\Facades\View::make('profile');

return view('profile');

There is absolutely no practical difference between facades and helper functions. When using helper functions, you may still test them exactly as you would the corresponding facade. For example, given the following route:

Route::get('/cache', function () {

return cache('key');

});

Under the hood, the cache helper is going to call the get method on the class underlying the Cache facade. So, even though we are using the helper function, we can write the following test to verify that the method was called with the argument we expected:

use Illuminate\Support\Facades\Cache;

/\*\*

\* A basic functional test example.

\*

\* @return void

\*/

public function testBasicExample()

{

Cache::shouldReceive('get')

->with('key')

->andReturn('value');

$response = $this->get('/cache');

$response->assertSee('value');

}

[How Facades Work](https://laravel.com/docs/8.x/facades#how-facades-work)

In a Laravel application, a facade is a class that provides access to an object from the container. The machinery that makes this work is in the Facade class. Laravel's facades, and any custom facades you create, will extend the base Illuminate\Support\Facades\Facade class.

The Facade base class makes use of the \_\_callStatic() magic-method to defer calls from your facade to an object resolved from the container. In the example below, a call is made to the Laravel cache system. By glancing at this code, one might assume that the static get method is being called on the Cache class:

**<?php**

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;

use Illuminate\Support\Facades\Cache;

class UserController extends Controller

{

/\*\*

\* Show the profile for the given user.

\*

\* @param int $id

\* @return Response

\*/

public function showProfile($id)

{

$user = Cache::get('user:'.$id);

return view('profile', ['user' => $user]);

}

}

Notice that near the top of the file we are "importing" the Cache facade. This facade serves as a proxy for accessing the underlying implementation of the Illuminate\Contracts\Cache\Factory interface. Any calls we make using the facade will be passed to the underlying instance of Laravel's cache service.

If we look at that Illuminate\Support\Facades\Cache class, you'll see that there is no static method get:

class Cache extends Facade

{

/\*\*

\* Get the registered name of the component.

\*

\* @return string

\*/

protected static function getFacadeAccessor() { return 'cache'; }

}

Instead, the Cache facade extends the base Facade class and defines the method getFacadeAccessor(). This method's job is to return the name of a service container binding. When a user references any static method on the Cache facade, Laravel resolves the cache binding from the [service container](https://laravel.com/docs/8.x/container) and runs the requested method (in this case, get) against that object.

[Real-Time Facades](https://laravel.com/docs/8.x/facades#real-time-facades)

Using real-time facades, you may treat any class in your application as if it was a facade. To illustrate how this can be used, let's first examine some code that does not use real-time facades. For example, let's assume our Podcast model has a publish method. However, in order to publish the podcast, we need to inject a Publisher instance:

**<?php**

namespace App\Models;

use App\Contracts\Publisher;

use Illuminate\Database\Eloquent\Model;

class Podcast extends Model

{

/\*\*

\* Publish the podcast.

\*

\* @param Publisher $publisher

\* @return void

\*/

public function publish(Publisher $publisher)

{

$this->update(['publishing' => now()]);

$publisher->publish($this);

}

}

Injecting a publisher implementation into the method allows us to easily test the method in isolation since we can mock the injected publisher. However, it requires us to always pass a publisher instance each time we call the publish method. Using real-time facades, we can maintain the same testability while not being required to explicitly pass a Publisher instance. To generate a real-time facade, prefix the namespace of the imported class with Facades:

**<?php**

namespace App\Models;

use Facades\App\Contracts\Publisher;

use Illuminate\Database\Eloquent\Model;

class Podcast extends Model

{

/\*\*

\* Publish the podcast.

\*

\* @return void

\*/

public function publish()

{

$this->update(['publishing' => now()]);

Publisher::publish($this);

}

}

When the real-time facade is used, the publisher implementation will be resolved out of the service container using the portion of the interface or class name that appears after the Facades prefix. When testing, we can use Laravel's built-in facade testing helpers to mock this method call:

**<?php**

namespace Tests\Feature;

use App\Models\Podcast;

use Facades\App\Contracts\Publisher;

use Illuminate\Foundation\Testing\RefreshDatabase;

use Tests\TestCase;

class PodcastTest extends TestCase

{

use RefreshDatabase;

/\*\*

\* A test example.

\*

\* @return void

\*/

public function test\_podcast\_can\_be\_published()

{

$podcast = Podcast::factory()->create();

Publisher::shouldReceive('publish')->once()->with($podcast);

$podcast->publish();

}

}

[Facade Class Reference](https://laravel.com/docs/8.x/facades#facade-class-reference)

Below you will find every facade and its underlying class. This is a useful tool for quickly digging into the API documentation for a given facade root. The [service container binding](https://laravel.com/docs/8.x/container) key is also included where applicable.

| **Facade** | **Class** | **Service Container Binding** |
| --- | --- | --- |
| App | [Illuminate\Foundation\Application](https://laravel.com/api/8.x/Illuminate/Foundation/Application.html) | app |
| Artisan | [Illuminate\Contracts\Console\Kernel](https://laravel.com/api/8.x/Illuminate/Contracts/Console/Kernel.html) | artisan |
| Auth | [Illuminate\Auth\AuthManager](https://laravel.com/api/8.x/Illuminate/Auth/AuthManager.html) | auth |
| Auth (Instance) | [Illuminate\Contracts\Auth\Guard](https://laravel.com/api/8.x/Illuminate/Contracts/Auth/Guard.html) | auth.driver |
| Blade | [Illuminate\View\Compilers\BladeCompiler](https://laravel.com/api/8.x/Illuminate/View/Compilers/BladeCompiler.html) | blade.compiler |
| Broadcast | [Illuminate\Contracts\Broadcasting\Factory](https://laravel.com/api/8.x/Illuminate/Contracts/Broadcasting/Factory.html) |  |
| Broadcast (Instance) | [Illuminate\Contracts\Broadcasting\Broadcaster](https://laravel.com/api/8.x/Illuminate/Contracts/Broadcasting/Broadcaster.html) |  |
| Bus | [Illuminate\Contracts\Bus\Dispatcher](https://laravel.com/api/8.x/Illuminate/Contracts/Bus/Dispatcher.html) |  |
| Cache | [Illuminate\Cache\CacheManager](https://laravel.com/api/8.x/Illuminate/Cache/CacheManager.html) | cache |
| Cache (Instance) | [Illuminate\Cache\Repository](https://laravel.com/api/8.x/Illuminate/Cache/Repository.html) | cache.store |
| Config | [Illuminate\Config\Repository](https://laravel.com/api/8.x/Illuminate/Config/Repository.html) | config |
| Cookie | [Illuminate\Cookie\CookieJar](https://laravel.com/api/8.x/Illuminate/Cookie/CookieJar.html) | cookie |
| Crypt | [Illuminate\Encryption\Encrypter](https://laravel.com/api/8.x/Illuminate/Encryption/Encrypter.html) | encrypter |
| Date | [Illuminate\Support\DateFactory](https://laravel.com/api/8.x/Illuminate/Support/DateFactory.html) | date |
| DB | [Illuminate\Database\DatabaseManager](https://laravel.com/api/8.x/Illuminate/Database/DatabaseManager.html) | db |
| DB (Instance) | [Illuminate\Database\Connection](https://laravel.com/api/8.x/Illuminate/Database/Connection.html) | db.connection |
| Event | [Illuminate\Events\Dispatcher](https://laravel.com/api/8.x/Illuminate/Events/Dispatcher.html) | events |
| File | [Illuminate\Filesystem\Filesystem](https://laravel.com/api/8.x/Illuminate/Filesystem/Filesystem.html) | files |
| Gate | [Illuminate\Contracts\Auth\Access\Gate](https://laravel.com/api/8.x/Illuminate/Contracts/Auth/Access/Gate.html) |  |
| Hash | [Illuminate\Contracts\Hashing\Hasher](https://laravel.com/api/8.x/Illuminate/Contracts/Hashing/Hasher.html) | hash |
| Http | [Illuminate\Http\Client\Factory](https://laravel.com/api/8.x/Illuminate/Http/Client/Factory.html) |  |
| Lang | [Illuminate\Translation\Translator](https://laravel.com/api/8.x/Illuminate/Translation/Translator.html) | translator |
| Log | [Illuminate\Log\LogManager](https://laravel.com/api/8.x/Illuminate/Log/LogManager.html) | log |
| Mail | [Illuminate\Mail\Mailer](https://laravel.com/api/8.x/Illuminate/Mail/Mailer.html) | mailer |
| Notification | [Illuminate\Notifications\ChannelManager](https://laravel.com/api/8.x/Illuminate/Notifications/ChannelManager.html) |  |
| Password | [Illuminate\Auth\Passwords\PasswordBrokerManager](https://laravel.com/api/8.x/Illuminate/Auth/Passwords/PasswordBrokerManager.html) | auth.password |
| Password (Instance) | [Illuminate\Auth\Passwords\PasswordBroker](https://laravel.com/api/8.x/Illuminate/Auth/Passwords/PasswordBroker.html) | auth.password.broker |
| Queue | [Illuminate\Queue\QueueManager](https://laravel.com/api/8.x/Illuminate/Queue/QueueManager.html) | queue |
| Queue (Instance) | [Illuminate\Contracts\Queue\Queue](https://laravel.com/api/8.x/Illuminate/Contracts/Queue/Queue.html) | queue.connection |
| Queue (Base Class) | [Illuminate\Queue\Queue](https://laravel.com/api/8.x/Illuminate/Queue/Queue.html) |  |
| Redirect | [Illuminate\Routing\Redirector](https://laravel.com/api/8.x/Illuminate/Routing/Redirector.html) | redirect |
| Redis | [Illuminate\Redis\RedisManager](https://laravel.com/api/8.x/Illuminate/Redis/RedisManager.html) | redis |
| Redis (Instance) | [Illuminate\Redis\Connections\Connection](https://laravel.com/api/8.x/Illuminate/Redis/Connections/Connection.html) | redis.connection |
| Request | [Illuminate\Http\Request](https://laravel.com/api/8.x/Illuminate/Http/Request.html) | request |
| Response | [Illuminate\Contracts\Routing\ResponseFactory](https://laravel.com/api/8.x/Illuminate/Contracts/Routing/ResponseFactory.html) |  |
| Response (Instance) | [Illuminate\Http\Response](https://laravel.com/api/8.x/Illuminate/Http/Response.html) |  |
| Route | [Illuminate\Routing\Router](https://laravel.com/api/8.x/Illuminate/Routing/Router.html) | router |
| Schema | [Illuminate\Database\Schema\Builder](https://laravel.com/api/8.x/Illuminate/Database/Schema/Builder.html) |  |
| Session | [Illuminate\Session\SessionManager](https://laravel.com/api/8.x/Illuminate/Session/SessionManager.html) | session |
| Session (Instance) | [Illuminate\Session\Store](https://laravel.com/api/8.x/Illuminate/Session/Store.html) | session.store |
| Storage | [Illuminate\Filesystem\FilesystemManager](https://laravel.com/api/8.x/Illuminate/Filesystem/FilesystemManager.html) | filesystem |
| Storage (Instance) | [Illuminate\Contracts\Filesystem\Filesystem](https://laravel.com/api/8.x/Illuminate/Contracts/Filesystem/Filesystem.html) | filesystem.disk |
| URL | [Illuminate\Routing\UrlGenerator](https://laravel.com/api/8.x/Illuminate/Routing/UrlGenerator.html) | url |
| Validator | [Illuminate\Validation\Factory](https://laravel.com/api/8.x/Illuminate/Validation/Factory.html) | validator |
| Validator (Instance) | [Illuminate\Validation\Validator](https://laravel.com/api/8.x/Illuminate/Validation/Validator.html) |  |
| View | [Illuminate\View\Factory](https://laravel.com/api/8.x/Illuminate/View/Factory.html) | view |
| View (Instance) | [Illuminate\View\View](https://laravel.com/api/8.x/Illuminate/View/View.html) |  |

Routing

* [Basic Routing](https://laravel.com/docs/8.x/routing#basic-routing)
  + [Redirect Routes](https://laravel.com/docs/8.x/routing#redirect-routes)
  + [View Routes](https://laravel.com/docs/8.x/routing#view-routes)
* [Route Parameters](https://laravel.com/docs/8.x/routing#route-parameters)
  + [Required Parameters](https://laravel.com/docs/8.x/routing#required-parameters)
  + [Optional Parameters](https://laravel.com/docs/8.x/routing#parameters-optional-parameters)
  + [Regular Expression Constraints](https://laravel.com/docs/8.x/routing#parameters-regular-expression-constraints)
* [Named Routes](https://laravel.com/docs/8.x/routing#named-routes)
* [Route Groups](https://laravel.com/docs/8.x/routing#route-groups)
  + [Middleware](https://laravel.com/docs/8.x/routing#route-group-middleware)
  + [Subdomain Routing](https://laravel.com/docs/8.x/routing#route-group-subdomain-routing)
  + [Route Prefixes](https://laravel.com/docs/8.x/routing#route-group-prefixes)
  + [Route Name Prefixes](https://laravel.com/docs/8.x/routing#route-group-name-prefixes)
* [Route Model Binding](https://laravel.com/docs/8.x/routing#route-model-binding)
  + [Implicit Binding](https://laravel.com/docs/8.x/routing#implicit-binding)
  + [Explicit Binding](https://laravel.com/docs/8.x/routing#explicit-binding)
* [Fallback Routes](https://laravel.com/docs/8.x/routing#fallback-routes)
* [Rate Limiting](https://laravel.com/docs/8.x/routing#rate-limiting)
  + [Defining Rate Limiters](https://laravel.com/docs/8.x/routing#defining-rate-limiters)
  + [Attaching Rate Limiters To Routes](https://laravel.com/docs/8.x/routing#attaching-rate-limiters-to-routes)
* [Form Method Spoofing](https://laravel.com/docs/8.x/routing#form-method-spoofing)
* [Accessing The Current Route](https://laravel.com/docs/8.x/routing#accessing-the-current-route)
* [Cross-Origin Resource Sharing (CORS)](https://laravel.com/docs/8.x/routing#cors)
* [Route Caching](https://laravel.com/docs/8.x/routing#route-caching)

[Basic Routing](https://laravel.com/docs/8.x/routing#basic-routing)

The most basic Laravel routes accept a URI and a closure, providing a very simple and expressive method of defining routes and behavior without complicated routing configuration files:

use Illuminate\Support\Facades\Route;

Route::get('/greeting', function () {

return 'Hello World';

});

[The Default Route Files](https://laravel.com/docs/8.x/routing#the-default-route-files)

All Laravel routes are defined in your route files, which are located in the routes directory. These files are automatically loaded by your application's App\Providers\RouteServiceProvider. The routes/web.php file defines routes that are for your web interface. These routes are assigned the web middleware group, which provides features like session state and CSRF protection. The routes in routes/api.php are stateless and are assigned the api middleware group.

For most applications, you will begin by defining routes in your routes/web.php file. The routes defined in routes/web.php may be accessed by entering the defined route's URL in your browser. For example, you may access the following route by navigating to http://example.com/user in your browser:

use App\Http\Controllers\UserController;

Route::get('/user', [UserController::class, 'index']);

Routes defined in the routes/api.php file are nested within a route group by the RouteServiceProvider. Within this group, the /api URI prefix is automatically applied so you do not need to manually apply it to every route in the file. You may modify the prefix and other route group options by modifying your RouteServiceProvider class.

[Available Router Methods](https://laravel.com/docs/8.x/routing#available-router-methods)

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

Route::get($uri, $callback);

Route::post($uri, $callback);

Route::put($uri, $callback);

Route::patch($uri, $callback);

Route::delete($uri, $callback);

Route::options($uri, $callback);

Sometimes you may need to register a route that responds to multiple HTTP verbs. You may do so using the match method. Or, you may even register a route that responds to all HTTP verbs using the any method:

Route::match(['get', 'post'], '/', function () {

//

});

Route::any('/', function () {

//

});

[Dependency Injection](https://laravel.com/docs/8.x/routing#dependency-injection)

You may type-hint any dependencies required by your route in your route's callback signature. The declared dependencies will automatically be resolved and injected into the callback by the Laravel [service container](https://laravel.com/docs/8.x/container). For example, you may type-hint the Illuminate\Http\Request class to have the current HTTP request automatically injected into your route callback:

use Illuminate\Http\Request;

Route::get('/users', function (Request $request) {

// ...

});

[CSRF Protection](https://laravel.com/docs/8.x/routing#csrf-protection)

Remember, any HTML forms pointing to POST, PUT, PATCH, or DELETE routes that are defined in the web routes file should include a CSRF token field. Otherwise, the request will be rejected. You can read more about CSRF protection in the [CSRF documentation](https://laravel.com/docs/8.x/csrf):

<form method="POST" action="/profile">

@csrf

...

</form>

[Redirect Routes](https://laravel.com/docs/8.x/routing#redirect-routes)

If you are defining a route that redirects to another URI, you may use the Route::redirect method. This method provides a convenient shortcut so that you do not have to define a full route or controller for performing a simple redirect:

Route::redirect('/here', '/there');

By default, Route::redirect returns a 302 status code. You may customize the status code using the optional third parameter:

Route::redirect('/here', '/there', 301);

Or, you may use the Route::permanentRedirect method to return a 301 status code:

Route::permanentRedirect('/here', '/there');

When using route parameters in redirect routes, the following parameters are reserved by Laravel and cannot be used: destination and status.

[View Routes](https://laravel.com/docs/8.x/routing#view-routes)

If your route only needs to return a [view](https://laravel.com/docs/8.x/views), you may use the Route::view method. Like the redirect method, this method provides a simple shortcut so that you do not have to define a full route or controller. The view method accepts a URI as its first argument and a view name as its second argument. In addition, you may provide an array of data to pass to the view as an optional third argument:

Route::view('/welcome', 'welcome');

Route::view('/welcome', 'welcome', ['name' => 'Taylor']);

When using route parameters in view routes, the following parameters are reserved by Laravel and cannot be used: view, data, status, and headers.

[Route Parameters](https://laravel.com/docs/8.x/routing#route-parameters)

[Required Parameters](https://laravel.com/docs/8.x/routing#required-parameters)

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:

Route::get('/user/{id}', function ($id) {

return 'User '.$id;

});

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

Route::get('/posts/{post}/comments/{comment}', function ($postId, $commentId) {

//

});

Route parameters are always encased within {} braces and should consist of alphabetic characters. Underscores (\_) are also acceptable within route parameter names. Route parameters are injected into route callbacks / controllers based on their order - the names of the route callback / controller arguments do not matter.

[Parameters & Dependency Injection](https://laravel.com/docs/8.x/routing#parameters-and-dependency-injection)

If your route has dependencies that you would like the Laravel service container to automatically inject into your route's callback, you should list your route parameters after your dependencies:

use Illuminate\Http\Request;

Route::get('/user/{id}', function (Request $request, $id) {

return 'User '.$id;

});

[Optional Parameters](https://laravel.com/docs/8.x/routing#parameters-optional-parameters)

Occasionally you may need to specify a route parameter that may not always be present in the URI. You may do so by placing a ? mark after the parameter name. Make sure to give the route's corresponding variable a default value:

Route::get('/user/{name?}', function ($name = null) {

return $name;

});

Route::get('/user/{name?}', function ($name = 'John') {

return $name;

});

[Regular Expression Constraints](https://laravel.com/docs/8.x/routing#parameters-regular-expression-constraints)

You may constrain the format of your route parameters using the where method on a route instance. The where method accepts the name of the parameter and a regular expression defining how the parameter should be constrained:

Route::get('/user/{name}', function ($name) {

//

})->where('name', '[A-Za-z]+');

Route::get('/user/{id}', function ($id) {

//

})->where('id', '[0-9]+');

Route::get('/user/{id}/{name}', function ($id, $name) {

//

})->where(['id' => '[0-9]+', 'name' => '[a-z]+']);

For convenience, some commonly used regular expression patterns have helper methods that allow you to quickly add pattern constraints to your routes:

Route::get('/user/{id}/{name}', function ($id, $name) {

//

})->whereNumber('id')->whereAlpha('name');

Route::get('/user/{name}', function ($name) {

//

})->whereAlphaNumeric('name');

Route::get('/user/{id}', function ($id) {

//

})->whereUuid('id');

If the incoming request does not match the route pattern constraints, a 404 HTTP response will be returned.

[Global Constraints](https://laravel.com/docs/8.x/routing#parameters-global-constraints)

If you would like a route parameter to always be constrained by a given regular expression, you may use the pattern method. You should define these patterns in the boot method of your App\Providers\RouteServiceProvider class:

/\*\*

\* Define your route model bindings, pattern filters, etc.

\*

\* @return void

\*/

public function boot()

{

Route::pattern('id', '[0-9]+');

}

Once the pattern has been defined, it is automatically applied to all routes using that parameter name:

Route::get('/user/{id}', function ($id) {

// Only executed if {id} is numeric...

});

[Encoded Forward Slashes](https://laravel.com/docs/8.x/routing#parameters-encoded-forward-slashes)

The Laravel routing component allows all characters except / to be present within route parameter values. You must explicitly allow / to be part of your placeholder using a where condition regular expression:

Route::get('/search/{search}', function ($search) {

return $search;

})->where('search', '.\*');

Encoded forward slashes are only supported within the last route segment.

[Named Routes](https://laravel.com/docs/8.x/routing#named-routes)

Named routes allow the convenient generation of URLs or redirects for specific routes. You may specify a name for a route by chaining the name method onto the route definition:

Route::get('/user/profile', function () {

//

})->name('profile');

You may also specify route names for controller actions:

Route::get(

'/user/profile',

[UserProfileController::class, 'show']

)->name('profile');

Route names should always be unique.

[Generating URLs To Named Routes](https://laravel.com/docs/8.x/routing#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 Laravel's route and redirect helper functions:

// Generating URLs...

$url = route('profile');

// Generating Redirects...

return redirect()->route('profile');

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

Route::get('/user/{id}/profile', function ($id) {

//

})->name('profile');

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

If you pass additional parameters in the array, those key / value pairs will automatically be added to the generated URL's query string:

Route::get('/user/{id}/profile', function ($id) {

//

})->name('profile');

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

// /user/1/profile?photos=yes

Sometimes, you may wish to specify request-wide default values for URL parameters, such as the current locale. To accomplish this, you may use the [URL::defaults method](https://laravel.com/docs/8.x/urls#default-values).

[Inspecting The Current Route](https://laravel.com/docs/8.x/routing#inspecting-the-current-route)

If you would like to determine if the current request was routed to a given named route, you may use the named method on a Route instance. For example, you may check the current route name from a route middleware:

/\*\*

\* Handle an incoming request.

\*

\* @param \Illuminate\Http\Request $request

\* @param \Closure $next

\* @return mixed

\*/

public function handle($request, Closure $next)

{

if ($request->route()->named('profile')) {

//

}

return $next($request);

}

[Route Groups](https://laravel.com/docs/8.x/routing#route-groups)

Route groups allow you to share route attributes, such as middleware, across a large number of routes without needing to define those attributes on each individual route.

Nested groups attempt to intelligently "merge" attributes with their parent group. Middleware and where conditions are merged while names and prefixes are appended. Namespace delimiters and slashes in URI prefixes are automatically added where appropriate.

[Middleware](https://laravel.com/docs/8.x/routing#route-group-middleware)

To assign [middleware](https://laravel.com/docs/8.x/middleware) to all routes within a group, you may use the middleware method before defining the group. Middleware are executed in the order they are listed in the array:

Route::middleware(['first', 'second'])->group(function () {

Route::get('/', function () {

// Uses first & second middleware...

});

Route::get('/user/profile', function () {

// Uses first & second middleware...

});

});

[Subdomain Routing](https://laravel.com/docs/8.x/routing#route-group-subdomain-routing)

Route groups may also be used to handle subdomain routing. Subdomains may be assigned route parameters just like route URIs, allowing you to capture a portion of the subdomain for usage in your route or controller. The subdomain may be specified by calling the domain method before defining the group:

Route::domain('{account}.example.com')->group(function () {

Route::get('user/{id}', function ($account, $id) {

//

});

});

In order to ensure your subdomain routes are reachable, you should register subdomain routes before registering root domain routes. This will prevent root domain routes from overwriting subdomain routes which have the same URI path.

[Route Prefixes](https://laravel.com/docs/8.x/routing#route-group-prefixes)

The prefix method 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:

Route::prefix('admin')->group(function () {

Route::get('/users', function () {

// Matches The "/admin/users" URL

});

});

[Route Name Prefixes](https://laravel.com/docs/8.x/routing#route-group-name-prefixes)

The name method may be used to prefix each route name in the group with a given string. For example, you may want to prefix all of the grouped route's names with admin. The given string is prefixed to the route name exactly as it is specified, so we will be sure to provide the trailing . character in the prefix:

Route::name('admin.')->group(function () {

Route::get('/users', function () {

// Route assigned name "admin.users"...

})->name('users');

});

[Route Model Binding](https://laravel.com/docs/8.x/routing#route-model-binding)

When injecting a model ID to a route or controller action, you will often query the database to retrieve the model that corresponds to that ID. Laravel route model binding provides a convenient way to automatically inject the model instances directly into your routes. For example, instead of injecting a user's ID, you can inject the entire User model instance that matches the given ID.

[Implicit Binding](https://laravel.com/docs/8.x/routing#implicit-binding)

Laravel automatically resolves Eloquent models defined in routes or controller actions whose type-hinted variable names match a route segment name. For example:

use App\Models\User;

Route::get('/users/{user}', function (User $user) {

return $user->email;

});

Since the $user variable is type-hinted as the App\Models\User Eloquent model and the variable name matches the {user} URI segment, Laravel will automatically inject the model instance that has an ID matching the corresponding value from the request URI. If a matching model instance is not found in the database, a 404 HTTP response will automatically be generated.

Of course, implicit binding is also possible when using controller methods. Again, note the {user} URI segment matches the $user variable in the controller which contains an App\Models\User type-hint:

use App\Http\Controllers\UserController;

use App\Models\User;

// Route definition...

Route::get('/users/{user}', [UserController::class, 'show']);

// Controller method definition...

public function show(User $user)

{

return view('user.profile', ['user' => $user]);

}

[Soft Deleted Models](https://laravel.com/docs/8.x/routing#implicit-soft-deleted-models)

Typically, implicit model binding will not retrieve models that have been [soft deleted](https://laravel.com/docs/8.x/eloquent#soft-deleting). However, you may instruct the implicit binding to retrieve these models by chaining the withTrashed method onto your route's definition:

use App\Models\User;

Route::get('/users/{user}', function (User $user) {

return $user->email;

})->withTrashed();

[Customizing The Key](https://laravel.com/docs/8.x/routing#customizing-the-default-key-name)

Sometimes you may wish to resolve Eloquent models using a column other than id. To do so, you may specify the column in the route parameter definition:

use App\Models\Post;

Route::get('/posts/{post:slug}', function (Post $post) {

return $post;

});

If you would like model binding to always use a database column other than id when retrieving a given model class, you may override the getRouteKeyName method on the Eloquent model:

/\*\*

\* Get the route key for the model.

\*

\* @return string

\*/

public function getRouteKeyName()

{

return 'slug';

}

[Custom Keys & Scoping](https://laravel.com/docs/8.x/routing#implicit-model-binding-scoping)

When implicitly binding multiple Eloquent models in a single route definition, you may wish to scope the second Eloquent model such that it must be a child of the previous Eloquent model. For example, consider this route definition that retrieves a blog post by slug for a specific user:

use App\Models\Post;

use App\Models\User;

Route::get('/users/{user}/posts/{post:slug}', function (User $user, Post $post) {

return $post;

});

When using a custom keyed implicit binding as a nested route parameter, Laravel will automatically scope the query to retrieve the nested model by its parent using conventions to guess the relationship name on the parent. In this case, it will be assumed that the User model has a relationship named posts (the plural form of the route parameter name) which can be used to retrieve the Post model.

[Customizing Missing Model Behavior](https://laravel.com/docs/8.x/routing#customizing-missing-model-behavior)

Typically, a 404 HTTP response will be generated if an implicitly bound model is not found. However, you may customize this behavior by calling the missing method when defining your route. The missing method accepts a closure that will be invoked if an implicitly bound model can not be found:

use App\Http\Controllers\LocationsController;

use Illuminate\Http\Request;

use Illuminate\Support\Facades\Redirect;

Route::get('/locations/{location:slug}', [LocationsController::class, 'show'])

->name('locations.view')

->missing(function (Request $request) {

return Redirect::route('locations.index');

});

[Explicit Binding](https://laravel.com/docs/8.x/routing#explicit-binding)

You are not required to use Laravel's implicit, convention based model resolution in order to use model binding. You can also explicitly define how route parameters correspond to models. To register an explicit binding, use the router's model method to specify the class for a given parameter. You should define your explicit model bindings at the beginning of the boot method of your RouteServiceProvider class:

use App\Models\User;

use Illuminate\Support\Facades\Route;

/\*\*

\* Define your route model bindings, pattern filters, etc.

\*

\* @return void

\*/

public function boot()

{

Route::model('user', User::class);

// ...

}

Next, define a route that contains a {user} parameter:

use App\Models\User;

Route::get('/users/{user}', function (User $user) {

//

});

Since we have bound all {user} parameters to the App\Models\User model, an instance of that class will be injected into the route. So, for example, a request to users/1 will inject the User instance from the database which has an ID of 1.

If a matching model instance is not found in the database, a 404 HTTP response will be automatically generated.

[Customizing The Resolution Logic](https://laravel.com/docs/8.x/routing#customizing-the-resolution-logic)

If you wish to define your own model binding resolution logic, you may use the Route::bind method. The closure you pass to the bind method will receive the value of the URI segment and should return the instance of the class that should be injected into the route. Again, this customization should take place in the boot method of your application's RouteServiceProvider:

use App\Models\User;

use Illuminate\Support\Facades\Route;

/\*\*

\* Define your route model bindings, pattern filters, etc.

\*

\* @return void

\*/

public function boot()

{

Route::bind('user', function ($value) {

return User::where('name', $value)->firstOrFail();

});

// ...

}

Alternatively, you may override the resolveRouteBinding method on your Eloquent model. This method will receive the value of the URI segment and should return the instance of the class that should be injected into the route:

/\*\*

\* Retrieve the model for a bound value.

\*

\* @param mixed $value

\* @param string|null $field

\* @return \Illuminate\Database\Eloquent\Model|null

\*/

public function resolveRouteBinding($value, $field = null)

{

return $this->where('name', $value)->firstOrFail();

}

If a route is utilizing [implicit binding scoping](https://laravel.com/docs/8.x/routing#implicit-model-binding-scoping), the resolveChildRouteBinding method will be used to resolve the child binding of the parent model:

/\*\*

\* Retrieve the child model for a bound value.

\*

\* @param string $childType

\* @param mixed $value

\* @param string|null $field

\* @return \Illuminate\Database\Eloquent\Model|null

\*/

public function resolveChildRouteBinding($childType, $value, $field)

{

return parent::resolveChildRouteBinding($childType, $value, $field);

}

[Fallback Routes](https://laravel.com/docs/8.x/routing#fallback-routes)

Using the Route::fallback method, you may define a route that will be executed when no other route matches the incoming request. Typically, unhandled requests will automatically render a "404" page via your application's exception handler. However, since you would typically define the fallback route within your routes/web.php file, all middleware in the web middleware group will apply to the route. You are free to add additional middleware to this route as needed:

Route::fallback(function () {

//

});

The fallback route should always be the last route registered by your application.

[Rate Limiting](https://laravel.com/docs/8.x/routing#rate-limiting)

[Defining Rate Limiters](https://laravel.com/docs/8.x/routing#defining-rate-limiters)

Laravel includes powerful and customizable rate limiting services that you may utilize to restrict the amount of traffic for a given route or group of routes. To get started, you should define rate limiter configurations that meet your application's needs. Typically, this should be done within the configureRateLimiting method of your application's App\Providers\RouteServiceProvider class.

Rate limiters are defined using the RateLimiter facade's for method. The for method accepts a rate limiter name and a closure that returns the limit configuration that should apply to routes that are assigned to the rate limiter. Limit configuration are instances of the Illuminate\Cache\RateLimiting\Limit class. This class contains helpful "builder" methods so that you can quickly define your limit. The rate limiter name may be any string you wish:

use Illuminate\Cache\RateLimiting\Limit;

use Illuminate\Support\Facades\RateLimiter;

/\*\*

\* Configure the rate limiters for the application.

\*

\* @return void

\*/

protected function configureRateLimiting()

{

RateLimiter::for('global', function (Request $request) {

return Limit::perMinute(1000);

});

}

If the incoming request exceeds the specified rate limit, a response with a 429 HTTP status code will automatically be returned by Laravel. If you would like to define your own response that should be returned by a rate limit, you may use the response method:

RateLimiter::for('global', function (Request $request) {

return Limit::perMinute(1000)->response(function () {

return response('Custom response...', 429);

});

});

Since rate limiter callbacks receive the incoming HTTP request instance, you may build the appropriate rate limit dynamically based on the incoming request or authenticated user:

RateLimiter::for('uploads', function (Request $request) {

return $request->user()->vipCustomer()

? Limit::none()

: Limit::perMinute(100);

});

[Segmenting Rate Limits](https://laravel.com/docs/8.x/routing#segmenting-rate-limits)

Sometimes you may wish to segment rate limits by some arbitrary value. For example, you may wish to allow users to access a given route 100 times per minute per IP address. To accomplish this, you may use the by method when building your rate limit:

RateLimiter::for('uploads', function (Request $request) {

return $request->user()->vipCustomer()

? Limit::none()

: Limit::perMinute(100)->by($request->ip());

});

To illustrate this feature using another example, we can limit access to the route to 100 times per minute per authenticated user ID or 10 times per minute per IP address for guests:

RateLimiter::for('uploads', function (Request $request) {

return $request->user()

? Limit::perMinute(100)->by($request->user()->id)

: Limit::perMinute(10)->by($request->ip());

});

[Multiple Rate Limits](https://laravel.com/docs/8.x/routing#multiple-rate-limits)

If needed, you may return an array of rate limits for a given rate limiter configuration. Each rate limit will be evaluated for the route based on the order they are placed within the array:

RateLimiter::for('login', function (Request $request) {

return [

Limit::perMinute(500),

Limit::perMinute(3)->by($request->input('email')),

];

});

[Attaching Rate Limiters To Routes](https://laravel.com/docs/8.x/routing#attaching-rate-limiters-to-routes)

Rate limiters may be attached to routes or route groups using the throttle [middleware](https://laravel.com/docs/8.x/middleware). The throttle middleware accepts the name of the rate limiter you wish to assign to the route:

Route::middleware(['throttle:uploads'])->group(function () {

Route::post('/audio', function () {

//

});

Route::post('/video', function () {

//

});

});

[Throttling With Redis](https://laravel.com/docs/8.x/routing#throttling-with-redis)

Typically, the throttle middleware is mapped to the Illuminate\Routing\Middleware\ThrottleRequests class. This mapping is defined in your application's HTTP kernel (App\Http\Kernel). However, if you are using Redis as your application's cache driver, you may wish to change this mapping to use the Illuminate\Routing\Middleware\ThrottleRequestsWithRedis class. This class is more efficient at managing rate limiting using Redis:

'throttle' => \Illuminate\Routing\Middleware\ThrottleRequestsWithRedis::class,

[Form Method Spoofing](https://laravel.com/docs/8.x/routing#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="/example" method="POST">

<input type="hidden" name="\_method" value="PUT">

<input type="hidden" name="\_token" value="{{ csrf\_token() }}">

</form>

For convenience, you may use the @method [Blade directive](https://laravel.com/docs/8.x/blade) to generate the \_method input field:

<form action="/example" method="POST">

@method('PUT')

@csrf

</form>

[Accessing The Current Route](https://laravel.com/docs/8.x/routing#accessing-the-current-route)

You may use the current, currentRouteName, and currentRouteAction methods on the Route facade to access information about the route handling the incoming request:

use Illuminate\Support\Facades\Route;

$route = Route::current(); // Illuminate\Routing\Route

$name = Route::currentRouteName(); // string

$action = Route::currentRouteAction(); // string

You may refer to the API documentation for both the [underlying class of the Route facade](https://laravel.com/api/8.x/Illuminate/Routing/Router.html) and [Route instance](https://laravel.com/api/8.x/Illuminate/Routing/Route.html) to review all of the methods that are available on the router and route classes.

[Cross-Origin Resource Sharing (CORS)](https://laravel.com/docs/8.x/routing#cors)

Laravel can automatically respond to CORS OPTIONS HTTP requests with values that you configure. All CORS settings may be configured in your application's config/cors.php configuration file. The OPTIONS requests will automatically be handled by the HandleCors [middleware](https://laravel.com/docs/8.x/middleware) that is included by default in your global middleware stack. Your global middleware stack is located in your application's HTTP kernel (App\Http\Kernel).

For more information on CORS and CORS headers, please consult the [MDN web documentation on CORS](https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS#The_HTTP_response_headers).

[Route Caching](https://laravel.com/docs/8.x/routing#route-caching)

When deploying your application to production, you should take advantage of Laravel's route cache. Using the route cache will drastically decrease the amount of time it takes to register all of your application's routes. To generate a route cache, execute the route:cache Artisan command:

php artisan route:cache

After running this command, your cached routes file will be loaded on every request. Remember, if you add any new routes you will need to generate a fresh route cache. Because of this, you should only run the route:cache command during your project's deployment.

You may use the route:clear command to clear the route cache:

php artisan route:clear

Middleware

* [Introduction](https://laravel.com/docs/8.x/middleware#introduction)
* [Defining Middleware](https://laravel.com/docs/8.x/middleware#defining-middleware)
* [Registering Middleware](https://laravel.com/docs/8.x/middleware#registering-middleware)
  + [Global Middleware](https://laravel.com/docs/8.x/middleware#global-middleware)
  + [Assigning Middleware To Routes](https://laravel.com/docs/8.x/middleware#assigning-middleware-to-routes)
  + [Middleware Groups](https://laravel.com/docs/8.x/middleware#middleware-groups)
  + [Sorting Middleware](https://laravel.com/docs/8.x/middleware#sorting-middleware)
* [Middleware Parameters](https://laravel.com/docs/8.x/middleware#middleware-parameters)
* [Terminable Middleware](https://laravel.com/docs/8.x/middleware#terminable-middleware)

[Introduction](https://laravel.com/docs/8.x/middleware#introduction)

Middleware provide a convenient mechanism for inspecting and filtering HTTP requests entering your application. For example, Laravel 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 your application's login screen. However, if the user is authenticated, the middleware will allow the request to proceed further into the application.

Additional middleware can be written to perform a variety of tasks besides authentication. For example, a logging middleware might log all incoming requests to your application. There are several middleware included in the Laravel framework, including middleware for authentication and CSRF protection. All of these middleware are located in the app/Http/Middleware directory.

[Defining Middleware](https://laravel.com/docs/8.x/middleware#defining-middleware)

To create a new middleware, use the make:middleware Artisan command:

php artisan make:middleware EnsureTokenIsValid

This command will place a new EnsureTokenIsValid class within your app/Http/Middleware directory. In this middleware, we will only allow access to the route if the supplied token input matches a specified value. Otherwise, we will redirect the users back to the home URI:

**<?php**

namespace App\Http\Middleware;

use Closure;

class EnsureTokenIsValid

{

/\*\*

\* Handle an incoming request.

\*

\* @param \Illuminate\Http\Request $request

\* @param \Closure $next

\* @return mixed

\*/

public function handle($request, Closure $next)

{

if ($request->input('token') !== 'my-secret-token') {

return redirect('home');

}

return $next($request);

}

}

As you can see, if the given token does not match our secret token, 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"), you should 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.

All middleware are resolved via the [service container](https://laravel.com/docs/8.x/container), so you may type-hint any dependencies you need within a middleware's constructor.

[Middleware & Responses](https://laravel.com/docs/8.x/middleware#middleware-and-responses)

Of course, a middleware can perform tasks before or after passing the request deeper into the application. 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](https://laravel.com/docs/8.x/middleware#registering-middleware)

[Global Middleware](https://laravel.com/docs/8.x/middleware#global-middleware)

If you want a middleware to run during every HTTP request to your application, list the middleware class in the $middleware property of your app/Http/Kernel.php class.

[Assigning Middleware To Routes](https://laravel.com/docs/8.x/middleware#assigning-middleware-to-routes)

If you would like to assign middleware to specific routes, you should first assign the middleware a key in your application's app/Http/Kernel.php file. By default, the $routeMiddleware property of this class contains entries for the middleware included with Laravel. You may add your own middleware to this list and assign it a key of your choosing:

// Within App\Http\Kernel class...

protected $routeMiddleware = [

'auth' => \App\Http\Middleware\Authenticate::class,

'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,

'bindings' => \Illuminate\Routing\Middleware\SubstituteBindings::class,

'cache.headers' => \Illuminate\Http\Middleware\SetCacheHeaders::class,

'can' => \Illuminate\Auth\Middleware\Authorize::class,

'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,

'signed' => \Illuminate\Routing\Middleware\ValidateSignature::class,

'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,

'verified' => \Illuminate\Auth\Middleware\EnsureEmailIsVerified::class,

];

Once the middleware has been defined in the HTTP kernel, you may use the middleware method to assign middleware to a route:

Route::get('/profile', function () {

//

})->middleware('auth');

You may assign multiple middleware to the route by passing an array of middleware names to the middleware method:

Route::get('/', function () {

//

})->middleware(['first', 'second']);

When assigning middleware, you may also pass the fully qualified class name:

use App\Http\Middleware\EnsureTokenIsValid;

Route::get('/profile', function () {

//

})->middleware(EnsureTokenIsValid::class);

When assigning middleware to a group of routes, you may occasionally need to prevent the middleware from being applied to an individual route within the group. You may accomplish this using the withoutMiddleware method:

use App\Http\Middleware\EnsureTokenIsValid;

Route::middleware([EnsureTokenIsValid::class])->group(function () {

Route::get('/', function () {

//

});

Route::get('/profile', function () {

//

})->withoutMiddleware([EnsureTokenIsValid::class]);

});

The withoutMiddleware method can only remove route middleware and does not apply to [global middleware](https://laravel.com/docs/8.x/middleware#global-middleware).

[Middleware Groups](https://laravel.com/docs/8.x/middleware#middleware-groups)

Sometimes you may want to group several middleware under a single key to make them easier to assign to routes. You may accomplish this using the $middlewareGroups property of your HTTP kernel.

Out of the box, Laravel comes with web and api middleware groups that contain common middleware you may want to apply to your web and API routes. Remember, these middleware groups are automatically applied by your application's App\Providers\RouteServiceProvider service provider to routes within your corresponding web and api route files:

/\*\*

\* The application's route middleware groups.

\*

\* @var array

\*/

protected $middlewareGroups = [

'web' => [

\App\Http\Middleware\EncryptCookies::class,

\Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,

\Illuminate\Session\Middleware\StartSession::class,

// \Illuminate\Session\Middleware\AuthenticateSession::class,

\Illuminate\View\Middleware\ShareErrorsFromSession::class,

\App\Http\Middleware\VerifyCsrfToken::class,

\Illuminate\Routing\Middleware\SubstituteBindings::class,

],

'api' => [

'throttle:api',

\Illuminate\Routing\Middleware\SubstituteBindings::class,

],

];

Middleware groups may be assigned to routes and controller actions using the same syntax as individual middleware. Again, middleware groups make it more convenient to assign many middleware to a route at once:

Route::get('/', function () {

//

})->middleware('web');

Route::middleware(['web'])->group(function () {

//

});

Out of the box, the web and api middleware groups are automatically applied to your application's corresponding routes/web.php and routes/api.php files by the App\Providers\RouteServiceProvider.

[Sorting Middleware](https://laravel.com/docs/8.x/middleware#sorting-middleware)

Rarely, you may need your middleware to execute in a specific order but not have control over their order when they are assigned to the route. In this case, you may specify your middleware priority using the $middlewarePriority property of your app/Http/Kernel.php file. This property may not exist in your HTTP kernel by default. If it does not exist, you may copy its default definition below:

/\*\*

\* The priority-sorted list of middleware.

\*

\* This forces non-global middleware to always be in the given order.

\*

\* @var array

\*/

protected $middlewarePriority = [

\Illuminate\Cookie\Middleware\EncryptCookies::class,

\Illuminate\Session\Middleware\StartSession::class,

\Illuminate\View\Middleware\ShareErrorsFromSession::class,

\Illuminate\Contracts\Auth\Middleware\AuthenticatesRequests::class,

\Illuminate\Routing\Middleware\ThrottleRequests::class,

\Illuminate\Session\Middleware\AuthenticateSession::class,

\Illuminate\Routing\Middleware\SubstituteBindings::class,

\Illuminate\Auth\Middleware\Authorize::class,

];

[Middleware Parameters](https://laravel.com/docs/8.x/middleware#middleware-parameters)

Middleware can also receive additional 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 an EnsureUserHasRole middleware 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 EnsureUserHasRole

{

/\*\*

\* Handle the incoming request.

\*

\* @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:

Route::put('/post/{id}', function ($id) {

//

})->middleware('role:editor');

[Terminable Middleware](https://laravel.com/docs/8.x/middleware#terminable-middleware)

Sometimes a middleware may need to do some work after the HTTP response has been sent to the browser. If you define a terminate method on your middleware and your web server is using FastCGI, the terminate method will automatically be called after the response is sent to the browser:

**<?php**

namespace Illuminate\Session\Middleware;

use Closure;

class TerminatingMiddleware

{

/\*\*

\* Handle an incoming request.

\*

\* @param \Illuminate\Http\Request $request

\* @param \Closure $next

\* @return mixed

\*/

public function handle($request, Closure $next)

{

return $next($request);

}

/\*\*

\* Handle tasks after the response has been sent to the browser.

\*

\* @param \Illuminate\Http\Request $request

\* @param \Illuminate\Http\Response $response

\* @return void

\*/

public function terminate($request, $response)

{

// ...

}

}

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 routes or global middleware in the app/Http/Kernel.php file.

When calling the terminate method on your middleware, Laravel will resolve a fresh instance of the middleware from the [service container](https://laravel.com/docs/8.x/container). If you would like to use the same middleware instance when the handle and terminate methods are called, register the middleware with the container using the container's singleton method. Typically this should be done in the register method of your AppServiceProvider:

use App\Http\Middleware\TerminatingMiddleware;

/\*\*

\* Register any application services.

\*

\* @return void

\*/

public function register()

{

$this->app->singleton(TerminatingMiddleware::class);

}

CSRF Protection

* [Introduction](https://laravel.com/docs/8.x/csrf#csrf-introduction)
* [Preventing CSRF Requests](https://laravel.com/docs/8.x/csrf#preventing-csrf-requests)
  + [Excluding URIs](https://laravel.com/docs/8.x/csrf#csrf-excluding-uris)
* [X-CSRF-Token](https://laravel.com/docs/8.x/csrf#csrf-x-csrf-token)
* [X-XSRF-Token](https://laravel.com/docs/8.x/csrf#csrf-x-xsrf-token)

[Introduction](https://laravel.com/docs/8.x/csrf#csrf-introduction)

Cross-site request forgeries are a type of malicious exploit whereby unauthorized commands are performed on behalf of an authenticated user. Thankfully, Laravel makes it easy to protect your application from [cross-site request forgery](https://en.wikipedia.org/wiki/Cross-site_request_forgery) (CSRF) attacks.

[An Explanation Of The Vulnerability](https://laravel.com/docs/8.x/csrf#csrf-explanation)

In case you're not familiar with cross-site request forgeries, let's discuss an example of how this vulnerability can be exploited. Imagine your application has a /user/email route that accepts a POST request to change the authenticated user's email address. Most likely, this route expects an email input field to contain the email address the user would like to begin using.

Without CSRF protection, a malicious website could create an HTML form that points to your application's /user/email route and submits the malicious user's own email address:

<form action="https://your-application.com/user/email" method="POST">

<input type="email" value="malicious-email@example.com">

</form>

<script>

document.forms[0].submit();

</script>

If the malicious website automatically submits the form when the page is loaded, the malicious user only needs to lure an unsuspecting user of your application to visit their website and their email address will be changed in your application.

To prevent this vulnerability, we need to inspect every incoming POST, PUT, PATCH, or DELETE request for a secret session value that the malicious application is unable to access.

[Preventing CSRF Requests](https://laravel.com/docs/8.x/csrf#preventing-csrf-requests)

Laravel automatically generates a CSRF "token" for each active [user session](https://laravel.com/docs/8.x/session) managed by the application. This token is used to verify that the authenticated user is the person actually making the requests to the application. Since this token is stored in the user's session and changes each time the session is regenerated, a malicious application is unable to access it.

The current session's CSRF token can be accessed via the request's session or via the csrf\_token helper function:

use Illuminate\Http\Request;

Route::get('/token', function (Request $request) {

$token = $request->session()->token();

$token = csrf\_token();

// ...

});

Anytime you define a "POST", "PUT", "PATCH", or "DELETE" HTML form in your application, you should include a hidden CSRF \_token field in the form so that the CSRF protection middleware can validate the request. For convenience, you may use the @csrf Blade directive to generate the hidden token input field:

<form method="POST" action="/profile">

@csrf

<!-- Equivalent to... -->

<input type="hidden" name="\_token" value="{{ csrf\_token() }}" />

</form>

The App\Http\Middleware\VerifyCsrfToken [middleware](https://laravel.com/docs/8.x/middleware), which is included in the web middleware group by default, will automatically verify that the token in the request input matches the token stored in the session. When these two tokens match, we know that the authenticated user is the one initiating the request.

[CSRF Tokens & SPAs](https://laravel.com/docs/8.x/csrf#csrf-tokens-and-spas)

If you are building an SPA that is utilizing Laravel as an API backend, you should consult the [Laravel Sanctum documentation](https://laravel.com/docs/8.x/sanctum) for information on authenticating with your API and protecting against CSRF vulnerabilities.

[Excluding URIs From CSRF Protection](https://laravel.com/docs/8.x/csrf#csrf-excluding-uris)

Sometimes you may wish to exclude a set of URIs from CSRF protection. For example, if you are using [Stripe](https://stripe.com/) to process payments and are utilizing their webhook system, you will need to exclude your Stripe webhook handler route from CSRF protection since Stripe will not know what CSRF token to send to your routes.

Typically, you should place these kinds of routes outside of the web middleware group that the App\Providers\RouteServiceProvider applies to all routes in the routes/web.php file. However, you may also exclude the routes by adding their URIs to the $except property of the VerifyCsrfToken middleware:

**<?php**

namespace App\Http\Middleware;

use Illuminate\Foundation\Http\Middleware\VerifyCsrfToken as Middleware;

class VerifyCsrfToken extends Middleware

{

/\*\*

\* The URIs that should be excluded from CSRF verification.

\*

\* @var array

\*/

protected $except = [

'stripe/\*',

'http://example.com/foo/bar',

'http://example.com/foo/\*',

];

}

For convenience, the CSRF middleware is automatically disabled for all routes when [running tests](https://laravel.com/docs/8.x/testing).

[X-CSRF-TOKEN](https://laravel.com/docs/8.x/csrf#csrf-x-csrf-token)

In addition to checking for the CSRF token as a POST parameter, the App\Http\Middleware\VerifyCsrfToken middleware will also check for the X-CSRF-TOKEN request header. You could, for example, store the token in an HTML meta tag:

<meta name="csrf-token" content="{{ csrf\_token() }}">

Then, you can instruct a library like jQuery to automatically add the token to all request headers. This provides simple, convenient CSRF protection for your AJAX based applications using legacy JavaScript technology:

$.ajaxSetup({

headers: {

'X-CSRF-TOKEN': $('meta[name="csrf-token"]').attr('content')

}

});

[X-XSRF-TOKEN](https://laravel.com/docs/8.x/csrf#csrf-x-xsrf-token)

Laravel stores the current CSRF token in an encrypted XSRF-TOKEN cookie that is included with each response generated by the framework. You can use the cookie value to set the X-XSRF-TOKEN request header.

This cookie is primarily sent as a developer convenience since some JavaScript frameworks and libraries, like Angular and Axios, automatically place its value in the X-XSRF-TOKEN header on same-origin requests.

By default, the resources/js/bootstrap.js file includes the Axios HTTP library which will automatically send the X-XSRF-TOKEN header for you.

Controllers

* [Introduction](https://laravel.com/docs/8.x/controllers#introduction)
* [Writing Controllers](https://laravel.com/docs/8.x/controllers#writing-controllers)
  + [Basic Controllers](https://laravel.com/docs/8.x/controllers#basic-controllers)
  + [Single Action Controllers](https://laravel.com/docs/8.x/controllers#single-action-controllers)
* [Controller Middleware](https://laravel.com/docs/8.x/controllers#controller-middleware)
* [Resource Controllers](https://laravel.com/docs/8.x/controllers#resource-controllers)
  + [Partial Resource Routes](https://laravel.com/docs/8.x/controllers#restful-partial-resource-routes)
  + [Nested Resources](https://laravel.com/docs/8.x/controllers#restful-nested-resources)
  + [Naming Resource Routes](https://laravel.com/docs/8.x/controllers#restful-naming-resource-routes)
  + [Naming Resource Route Parameters](https://laravel.com/docs/8.x/controllers#restful-naming-resource-route-parameters)
  + [Scoping Resource Routes](https://laravel.com/docs/8.x/controllers#restful-scoping-resource-routes)
  + [Localizing Resource URIs](https://laravel.com/docs/8.x/controllers#restful-localizing-resource-uris)
  + [Supplementing Resource Controllers](https://laravel.com/docs/8.x/controllers#restful-supplementing-resource-controllers)
* [Dependency Injection & Controllers](https://laravel.com/docs/8.x/controllers#dependency-injection-and-controllers)

[Introduction](https://laravel.com/docs/8.x/controllers#introduction)

Instead of defining all of your request handling logic as closures in your route files, you may wish to organize this behavior using "controller" classes. Controllers can group related request handling logic into a single class. For example, a UserController class might handle all incoming requests related to users, including showing, creating, updating, and deleting users. By default, controllers are stored in the app/Http/Controllers directory.

[Writing Controllers](https://laravel.com/docs/8.x/controllers#writing-controllers)

[Basic Controllers](https://laravel.com/docs/8.x/controllers#basic-controllers)

Let's take a look at an example of a basic controller. Note that the controller extends the base controller class included with Laravel: App\Http\Controllers\Controller:

**<?php**

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;

use App\Models\User;

class UserController extends Controller

{

/\*\*

\* Show the profile for a given user.

\*

\* @param int $id

\* @return \Illuminate\View\View

\*/

public function show($id)

{

return view('user.profile', [

'user' => User::findOrFail($id)

]);

}

}

You can define a route to this controller method like so:

use App\Http\Controllers\UserController;

Route::get('/user/{id}', [UserController::class, 'show']);

When an incoming request matches the specified route URI, the show method on the App\Http\Controllers\UserController class will be invoked and the route parameters will be passed to the method.

Controllers are not required to extend a base class. However, you will not have access to convenient features such as the middleware and authorize methods.

[Single Action Controllers](https://laravel.com/docs/8.x/controllers#single-action-controllers)

If a controller action is particularly complex, you might find it convenient to dedicate an entire controller class to that single action. To accomplish this, you may define a single \_\_invoke method within the controller:

**<?php**

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;

use App\Models\User;

class ProvisionServer extends Controller

{

/\*\*

\* Provision a new web server.

\*

\* @return \Illuminate\Http\Response

\*/

public function \_\_invoke()

{

// ...

}

}

When registering routes for single action controllers, you do not need to specify a controller method. Instead, you may simply pass the name of the controller to the router:

use App\Http\Controllers\ProvisionServer;

Route::post('/server', ProvisionServer::class);

You may generate an invokable controller by using the --invokable option of the make:controller Artisan command:

php artisan make:controller ProvisionServer --invokable

Controller stubs may be customized using [stub publishing](https://laravel.com/docs/8.x/artisan#stub-customization).

[Controller Middleware](https://laravel.com/docs/8.x/controllers#controller-middleware)

[Middleware](https://laravel.com/docs/8.x/middleware) may be assigned to the controller's routes in your route files:

Route::get('profile', [UserController::class, 'show'])->middleware('auth');

Or, you may find it convenient to specify middleware within your controller's constructor. Using the middleware method within your controller's constructor, you can assign middleware to the controller's actions:

class UserController extends Controller

{

/\*\*

\* Instantiate a new controller instance.

\*

\* @return void

\*/

public function \_\_construct()

{

$this->middleware('auth');

$this->middleware('log')->only('index');

$this->middleware('subscribed')->except('store');

}

}

Controllers also allow you to register middleware using a closure. This provides a convenient way to define an inline middleware for a single controller without defining an entire middleware class:

$this->middleware(function ($request, $next) {

return $next($request);

});

[Resource Controllers](https://laravel.com/docs/8.x/controllers#resource-controllers)

If you think of each Eloquent model in your application as a "resource", it is typical to perform the same sets of actions against each resource in your application. For example, imagine your application contains a Photo model and a Movie model. It is likely that users can create, read, update, or delete these resources.

Because of this common use case, Laravel resource routing assigns the typical create, read, update, and delete ("CRUD") routes to a controller with a single line of code. To get started, we can use the make:controller Artisan command's --resource option to quickly create a controller to handle these actions:

php artisan make:controller PhotoController --resource

This command will generate a controller at app/Http/Controllers/PhotoController.php. The controller will contain a method for each of the available resource operations. Next, you may register a resource route that points to the controller:

use App\Http\Controllers\PhotoController;

Route::resource('photos', PhotoController::class);

This single route declaration creates multiple routes to handle a variety of actions on the resource. The generated controller will already have methods stubbed for each of these actions. Remember, you can always get a quick overview of your application's routes by running the route:list Artisan command.

You may even register many resource controllers at once by passing an array to the resources method:

Route::resources([

'photos' => PhotoController::class,

'posts' => PostController::class,

]);

[Actions Handled By Resource Controller](https://laravel.com/docs/8.x/controllers#actions-handled-by-resource-controller)

| **Verb** | **URI** | **Action** | **Route Name** |
| --- | --- | --- | --- |
| GET | /photos | index | photos.index |
| GET | /photos/create | create | photos.create |
| POST | /photos | store | photos.store |
| GET | /photos/{photo} | show | photos.show |
| GET | /photos/{photo}/edit | edit | photos.edit |
| PUT/PATCH | /photos/{photo} | update | photos.update |
| DELETE | /photos/{photo} | destroy | photos.destroy |

[Customizing Missing Model Behavior](https://laravel.com/docs/8.x/controllers#customizing-missing-model-behavior)

Typically, a 404 HTTP response will be generated if an implicitly bound resource model is not found. However, you may customize this behavior by calling the missing method when defining your resource route. The missing method accepts a closure that will be invoked if an implicitly bound model can not be found for any of the resource's routes:

use App\Http\Controllers\PhotoController;

use Illuminate\Http\Request;

use Illuminate\Support\Facades\Redirect;

Route::resource('photos', PhotoController::class)

->missing(function (Request $request) {

return Redirect::route('photos.index');

});

[Specifying The Resource Model](https://laravel.com/docs/8.x/controllers#specifying-the-resource-model)

If you are using [route model binding](https://laravel.com/docs/8.x/routing#route-model-binding) and would like the resource controller's methods to type-hint a model instance, you may use the --model option when generating the controller:

php artisan make:controller PhotoController --resource --model=Photo

[Partial Resource Routes](https://laravel.com/docs/8.x/controllers#restful-partial-resource-routes)

When declaring a resource route, you may specify a subset of actions the controller should handle instead of the full set of default actions:

use App\Http\Controllers\PhotoController;

Route::resource('photos', PhotoController::class)->only([

'index', 'show'

]);

Route::resource('photos', PhotoController::class)->except([

'create', 'store', 'update', 'destroy'

]);

[API Resource Routes](https://laravel.com/docs/8.x/controllers#api-resource-routes)

When declaring resource routes that will be consumed by APIs, you will commonly want to exclude routes that present HTML templates such as create and edit. For convenience, you may use the apiResource method to automatically exclude these two routes:

use App\Http\Controllers\PhotoController;

Route::apiResource('photos', PhotoController::class);

You may register many API resource controllers at once by passing an array to the apiResources method:

use App\Http\Controllers\PhotoController;

use App\Http\Controllers\PostController;

Route::apiResources([

'photos' => PhotoController::class,

'posts' => PostController::class,

]);

To quickly generate an API resource controller that does not include the create or edit methods, use the --api switch when executing the make:controller command:

php artisan make:controller PhotoController --api

[Nested Resources](https://laravel.com/docs/8.x/controllers#restful-nested-resources)

Sometimes you may need to define routes to a nested resource. For example, a photo resource may have multiple comments that may be attached to the photo. To nest the resource controllers, you may use "dot" notation in your route declaration:

use App\Http\Controllers\PhotoCommentController;

Route::resource('photos.comments', PhotoCommentController::class);

This route will register a nested resource that may be accessed with URIs like the following:

/photos/{photo}/comments/{comment}

[Scoping Nested Resources](https://laravel.com/docs/8.x/controllers#scoping-nested-resources)

Laravel's [implicit model binding](https://laravel.com/docs/8.x/routing#implicit-model-binding-scoping) feature can automatically scope nested bindings such that the resolved child model is confirmed to belong to the parent model. By using the scoped method when defining your nested resource, you may enable automatic scoping as well as instruct Laravel which field the child resource should be retrieved by. For more information on how to accomplish this, please see the documentation on [scoping resource routes](https://laravel.com/docs/8.x/controllers#restful-scoping-resource-routes).

[Shallow Nesting](https://laravel.com/docs/8.x/controllers#shallow-nesting)

Often, it is not entirely necessary to have both the parent and the child IDs within a URI since the child ID is already a unique identifier. When using unique identifiers such as auto-incrementing primary keys to identify your models in URI segments, you may choose to use "shallow nesting":

use App\Http\Controllers\CommentController;

Route::resource('photos.comments', CommentController::class)->shallow();

This route definition will define the following routes:

| **Verb** | **URI** | **Action** | **Route Name** |
| --- | --- | --- | --- |
| GET | /photos/{photo}/comments | index | photos.comments.index |
| GET | /photos/{photo}/comments/create | create | photos.comments.create |
| POST | /photos/{photo}/comments | store | photos.comments.store |
| GET | /comments/{comment} | show | comments.show |
| GET | /comments/{comment}/edit | edit | comments.edit |
| PUT/PATCH | /comments/{comment} | update | comments.update |
| DELETE | /comments/{comment} | destroy | comments.destroy |

[Naming Resource Routes](https://laravel.com/docs/8.x/controllers#restful-naming-resource-routes)

By default, all resource controller actions have a route name; however, you can override these names by passing a names array with your desired route names:

use App\Http\Controllers\PhotoController;

Route::resource('photos', PhotoController::class)->names([

'create' => 'photos.build'

]);

[Naming Resource Route Parameters](https://laravel.com/docs/8.x/controllers#restful-naming-resource-route-parameters)

By default, Route::resource will create the route parameters for your resource routes based on the "singularized" version of the resource name. You can easily override this on a per resource basis using the parameters method. The array passed into the parameters method should be an associative array of resource names and parameter names:

use App\Http\Controllers\AdminUserController;

Route::resource('users', AdminUserController::class)->parameters([

'users' => 'admin\_user'

]);

The example above generates the following URI for the resource's show route:

/users/{admin\_user}

[Scoping Resource Routes](https://laravel.com/docs/8.x/controllers#restful-scoping-resource-routes)

Laravel's [scoped implicit model binding](https://laravel.com/docs/8.x/routing#implicit-model-binding-scoping) feature can automatically scope nested bindings such that the resolved child model is confirmed to belong to the parent model. By using the scoped method when defining your nested resource, you may enable automatic scoping as well as instruct Laravel which field the child resource should be retrieved by:

use App\Http\Controllers\PhotoCommentController;

Route::resource('photos.comments', PhotoCommentController::class)->scoped([

'comment' => 'slug',

]);

This route will register a scoped nested resource that may be accessed with URIs like the following:

/photos/{photo}/comments/{comment:slug}

When using a custom keyed implicit binding as a nested route parameter, Laravel will automatically scope the query to retrieve the nested model by its parent using conventions to guess the relationship name on the parent. In this case, it will be assumed that the Photo model has a relationship named comments (the plural of the route parameter name) which can be used to retrieve the Comment model.

[Localizing Resource URIs](https://laravel.com/docs/8.x/controllers#restful-localizing-resource-uris)

By default, Route::resource will create resource URIs using English verbs. If you need to localize the create and edit action verbs, you may use the Route::resourceVerbs method. This may be done at the beginning of the boot method within your application's App\Providers\RouteServiceProvider:

/\*\*

\* Define your route model bindings, pattern filters, etc.

\*

\* @return void

\*/

public function boot()

{

Route::resourceVerbs([

'create' => 'crear',

'edit' => 'editar',

]);

// ...

}

Once the verbs have been customized, a resource route registration such as Route::resource('fotos', PhotoController::class) will produce the following URIs:

/fotos/crear

/fotos/{foto}/editar

[Supplementing Resource Controllers](https://laravel.com/docs/8.x/controllers#restful-supplementing-resource-controllers)

If you need to add additional routes to a resource controller beyond the default set of resource routes, you should define those routes before your call to the Route::resource method; otherwise, the routes defined by the resource method may unintentionally take precedence over your supplemental routes:

use App\Http\Controller\PhotoController;

Route::get('/photos/popular', [PhotoController::class, 'popular']);

Route::resource('photos', PhotoController::class);

Remember to keep your controllers focused. If you find yourself routinely needing methods outside of the typical set of resource actions, consider splitting your controller into two, smaller controllers.

[Dependency Injection & Controllers](https://laravel.com/docs/8.x/controllers#dependency-injection-and-controllers)

[Constructor Injection](https://laravel.com/docs/8.x/controllers#constructor-injection)

The Laravel [service container](https://laravel.com/docs/8.x/container) is used to resolve all Laravel controllers. As a result, you are able to type-hint any dependencies your controller may need in its constructor. The declared 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 \App\Repositories\UserRepository $users

\* @return void

\*/

public function \_\_construct(UserRepository $users)

{

$this->users = $users;

}

}

[Method Injection](https://laravel.com/docs/8.x/controllers#method-injection)

In addition to constructor injection, you may also type-hint dependencies on your controller's methods. A common use-case for method injection is injecting the Illuminate\Http\Request instance into your controller methods:

**<?php**

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller

{

/\*\*

\* Store a new user.

\*

\* @param \Illuminate\Http\Request $request

\* @return \Illuminate\Http\Response

\*/

public function store(Request $request)

{

$name = $request->name;

//

}

}

If your controller method is also expecting input from a route parameter, list your route arguments after your other dependencies. For example, if your route is defined like so:

use App\Http\Controllers\UserController;

Route::put('/user/{id}', [UserController::class, 'update']);

You may still type-hint the Illuminate\Http\Request and access your id parameter by defining your controller method as follows:

**<?php**

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller

{

/\*\*

\* Update the given user.

\*

\* @param \Illuminate\Http\Request $request

\* @param string $id

\* @return \Illuminate\Http\Response

\*/

public function update(Request $request, $id)

{

//

}

}

HTTP Requests

* [Introduction](https://laravel.com/docs/8.x/requests#introduction)
* [Interacting With The Request](https://laravel.com/docs/8.x/requests#interacting-with-the-request)
  + [Accessing The Request](https://laravel.com/docs/8.x/requests#accessing-the-request)
  + [Request Path & Method](https://laravel.com/docs/8.x/requests#request-path-and-method)
  + [Request Headers](https://laravel.com/docs/8.x/requests#request-headers)
  + [Request IP Address](https://laravel.com/docs/8.x/requests#request-ip-address)
  + [Content Negotiation](https://laravel.com/docs/8.x/requests#content-negotiation)
  + [PSR-7 Requests](https://laravel.com/docs/8.x/requests#psr7-requests)
* [Input](https://laravel.com/docs/8.x/requests#input)
  + [Retrieving Input](https://laravel.com/docs/8.x/requests#retrieving-input)
  + [Determining If Input Is Present](https://laravel.com/docs/8.x/requests#determining-if-input-is-present)
  + [Old Input](https://laravel.com/docs/8.x/requests#old-input)
  + [Cookies](https://laravel.com/docs/8.x/requests#cookies)
  + [Input Trimming & Normalization](https://laravel.com/docs/8.x/requests#input-trimming-and-normalization)
* [Files](https://laravel.com/docs/8.x/requests#files)
  + [Retrieving Uploaded Files](https://laravel.com/docs/8.x/requests#retrieving-uploaded-files)
  + [Storing Uploaded Files](https://laravel.com/docs/8.x/requests#storing-uploaded-files)
* [Configuring Trusted Proxies](https://laravel.com/docs/8.x/requests#configuring-trusted-proxies)
* [Configuring Trusted Hosts](https://laravel.com/docs/8.x/requests#configuring-trusted-hosts)

[Introduction](https://laravel.com/docs/8.x/requests#introduction)

Laravel's Illuminate\Http\Request class provides an object-oriented way to interact with the current HTTP request being handled by your application as well as retrieve the input, cookies, and files that were submitted with the request.

[Interacting With The Request](https://laravel.com/docs/8.x/requests#interacting-with-the-request)

[Accessing The Request](https://laravel.com/docs/8.x/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 route closure or controller method. The incoming request instance will automatically be injected by the Laravel [service container](https://laravel.com/docs/8.x/container):

**<?php**

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller

{

/\*\*

\* Store a new user.

\*

\* @param \Illuminate\Http\Request $request

\* @return \Illuminate\Http\Response

\*/

public function store(Request $request)

{

$name = $request->input('name');

//

}

}

As mentioned, you may also type-hint the Illuminate\Http\Request class on a route closure. The service container will automatically inject the incoming request into the closure when it is executed:

use Illuminate\Http\Request;

Route::get('/', function (Request $request) {

//

});

[Dependency Injection & Route Parameters](https://laravel.com/docs/8.x/requests#dependency-injection-route-parameters)

If your controller method is also expecting input from a route parameter you should list your route parameters after your other dependencies. For example, if your route is defined like so:

use App\Http\Controllers\UserController;

Route::put('/user/{id}', [UserController::class, 'update']);

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

**<?php**

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller

{

/\*\*

\* Update the specified user.

\*

\* @param \Illuminate\Http\Request $request

\* @param string $id

\* @return \Illuminate\Http\Response

\*/

public function update(Request $request, $id)

{

//

}

}

[Request Path & Method](https://laravel.com/docs/8.x/requests#request-path-and-method)

The Illuminate\Http\Request instance provides a variety of methods for examining the incoming HTTP request and extends the Symfony\Component\HttpFoundation\Request class. We will discuss a few of the most important methods below.

[Retrieving The Request Path](https://laravel.com/docs/8.x/requests#retrieving-the-request-path)

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

$uri = $request->path();

[Inspecting The Request Path / Route](https://laravel.com/docs/8.x/requests#inspecting-the-request-path)

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

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

//

}

Using the routeIs method, you may determine if the incoming request has matched a [named route](https://laravel.com/docs/8.x/routing#named-routes):

if ($request->routeIs('admin.\*')) {

//

}

[Retrieving The Request URL](https://laravel.com/docs/8.x/requests#retrieving-the-request-url)

To retrieve the full URL for the incoming request you may use the url or fullUrl methods. The url method will return the URL without the query string, while the fullUrl method includes the query string:

$url = $request->url();

$urlWithQueryString = $request->fullUrl();

If you would like to append query string data to the current URL, you may call the fullUrlWithQuery method. This method merges the given array of query string variables with the current query string:

$request->fullUrlWithQuery(['type' => 'phone']);

[Retrieving The Request Method](https://laravel.com/docs/8.x/requests#retrieving-the-request-method)

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

$method = $request->method();

if ($request->isMethod('post')) {

//

}

[Request Headers](https://laravel.com/docs/8.x/requests#request-headers)

You may retrieve a request header from the Illuminate\Http\Request instance using the header method. If the header is not present on the request, null will be returned. However, the header method accepts an optional second argument that will be returned if the header is not present on the request:

$value = $request->header('X-Header-Name');

$value = $request->header('X-Header-Name', 'default');

The hasHeader method may be used to determine if the request contains a given header:

if ($request->hasHeader('X-Header-Name')) {

//

}

For convenience, the bearerToken method may be used to retrieve a bearer token from the Authorization header. If no such header is present, an empty string will be returned:

$token = $request->bearerToken();

[Request IP Address](https://laravel.com/docs/8.x/requests#request-ip-address)

The ip method may be used to retrieve the IP address of the client that made the request to your application:

$ipAddress = $request->ip();

[Content Negotiation](https://laravel.com/docs/8.x/requests#content-negotiation)

Laravel provides several methods for inspecting the incoming request's requested content types via the Accept header. First, the getAcceptableContentTypes method will return an array containing all of the content types accepted by the request:

$contentTypes = $request->getAcceptableContentTypes();

The accepts method accepts an array of content types and returns true if any of the content types are accepted by the request. Otherwise, false will be returned:

if ($request->accepts(['text/html', 'application/json'])) {

// ...

}

You may use the prefers method to determine which content type out of a given array of content types is most preferred by the request. If none of the provided content types are accepted by the request, null will be returned:

$preferred = $request->prefers(['text/html', 'application/json']);

Since many applications only serve HTML or JSON, you may use the expectsJson method to quickly determine if the incoming request expects a JSON response:

if ($request->expectsJson()) {

// ...

}

[PSR-7 Requests](https://laravel.com/docs/8.x/requests#psr7-requests)

The [PSR-7 standard](https://www.php-fig.org/psr/psr-7/) specifies interfaces for HTTP messages, including requests and responses. If you would like to obtain an instance of a PSR-7 request instead of a Laravel request, you will first need to install a few libraries. Laravel uses the *Symfony HTTP Message Bridge* component to convert typical Laravel requests and responses into PSR-7 compatible implementations:

composer require symfony/psr-http-message-bridge

composer require nyholm/psr7

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

use Psr\Http\Message\ServerRequestInterface;

Route::get('/', function (ServerRequestInterface $request) {

//

});

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

[Input](https://laravel.com/docs/8.x/requests#input)

[Retrieving Input](https://laravel.com/docs/8.x/requests#retrieving-input)

[Retrieving All Input Data](https://laravel.com/docs/8.x/requests#retrieving-all-input-data)

You may retrieve all of the incoming request's input data as an array using the all method. This method may be used regardless of whether the incoming request is from an HTML form or is an XHR request:

$input = $request->all();

Using the collect method, you may retrieve all of the incoming request's input data as a [collection](https://laravel.com/docs/8.x/collections):

$input = $request->collect();

The collect method also allows you to retrieve a subset of the incoming request input as a collection:

$request->collect('users')->each(function ($user) {

// ...

});

[Retrieving An Input Value](https://laravel.com/docs/8.x/requests#retrieving-an-input-value)

Using a few simple methods, you may access all of the user input from your Illuminate\Http\Request instance without worrying about which HTTP verb was used for the request. Regardless of the HTTP verb, the input method may be used to retrieve user input:

$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 with forms that contain array inputs, use "dot" notation to access the arrays:

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

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

You may call the input method without any arguments in order to retrieve all of the input values as an associative array:

$input = $request->input();

[Retrieving Input From The Query String](https://laravel.com/docs/8.x/requests#retrieving-input-from-the-query-string)

While the input method retrieves values from the entire request payload (including the query string), the query method will only retrieve values from the query string:

$name = $request->query('name');

If the requested query string value data is not present, the second argument to this method will be returned:

$name = $request->query('name', 'Helen');

You may call the query method without any arguments in order to retrieve all of the query string values as an associative array:

$query = $request->query();

[Retrieving JSON Input Values](https://laravel.com/docs/8.x/requests#retrieving-json-input-values)

When sending JSON requests to your application, you may access the JSON data via the input method as long as the Content-Type header of the request is properly set to application/json. You may even use "dot" syntax to retrieve values that are nested within JSON arrays:

$name = $request->input('user.name');

[Retrieving Boolean Input Values](https://laravel.com/docs/8.x/requests#retrieving-boolean-input-values)

When dealing with HTML elements like checkboxes, your application may receive "truthy" values that are actually strings. For example, "true" or "on". For convenience, you may use the boolean method to retrieve these values as booleans. The boolean method returns true for 1, "1", true, "true", "on", and "yes". All other values will return false:

$archived = $request->boolean('archived');

[Retrieving Input Via Dynamic Properties](https://laravel.com/docs/8.x/requests#retrieving-input-via-dynamic-properties)

You may also access user input using dynamic properties on the Illuminate\Http\Request instance. For example, if one of your application's forms contains a name field, you may access the value of the field like so:

$name = $request->name;

When using dynamic properties, Laravel will first look for the parameter's value in the request payload. If it is not present, Laravel will search for the field in the matched route's parameters.

[Retrieving A Portion Of The Input Data](https://laravel.com/docs/8.x/requests#retrieving-a-portion-of-the-input-data)

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

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

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

$input = $request->except(['credit\_card']);

$input = $request->except('credit\_card');

The only method returns all of the key / value pairs that you request; however, it will not return key / value pairs that are not present on the request.

[Determining If Input Is Present](https://laravel.com/docs/8.x/requests#determining-if-input-is-present)

You may use the has method to determine if a value is present on the request. The has method returns true if the value is present on the request:

if ($request->has('name')) {

//

}

When given an array, the has method will determine if all of the specified values are present:

if ($request->has(['name', 'email'])) {

//

}

The whenHas method will execute the given closure if a value is present on the request:

$request->whenHas('name', function ($input) {

//

});

The hasAny method returns true if any of the specified values are present:

if ($request->hasAny(['name', 'email'])) {

//

}

If you would like to determine if a value is present on the request and is not empty, you may use the filled method:

if ($request->filled('name')) {

//

}

The whenFilled method will execute the given closure if a value is present on the request and is not empty:

$request->whenFilled('name', function ($input) {

//

});

To determine if a given key is absent from the request, you may use the missing method:

if ($request->missing('name')) {

//

}

[Old Input](https://laravel.com/docs/8.x/requests#old-input)

Laravel 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 Laravel's included [validation features](https://laravel.com/docs/8.x/validation), it is possible that you will not need to manually use these session input flashing methods directly, as some of Laravel's built-in validation facilities will call them automatically.

[Flashing Input To The Session](https://laravel.com/docs/8.x/requests#flashing-input-to-the-session)

The flash method on the Illuminate\Http\Request class will flash the current input to the [session](https://laravel.com/docs/8.x/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 subset of the request data to the session. These methods are useful for keeping sensitive information such as passwords out of the session:

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

$request->flashExcept('password');

[Flashing Input Then Redirecting](https://laravel.com/docs/8.x/requests#flashing-input-then-redirecting)

Since you often will want to flash input to the session and then redirect to the previous page, you may easily chain input flashing onto a redirect using the withInput method:

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

return redirect()->route('user.create')->withInput();

return redirect('form')->withInput(

$request->except('password')

);

[Retrieving Old Input](https://laravel.com/docs/8.x/requests#retrieving-old-input)

To retrieve flashed input from the previous request, invoke the old method on an instance of Illuminate\Http\Request. The old method will pull the previously flashed input data from the [session](https://laravel.com/docs/8.x/session):

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

Laravel also provides a global old helper. If you are displaying old input within a [Blade template](https://laravel.com/docs/8.x/blade), it is more convenient to use the old helper to repopulate the form. If no old input exists for the given field, null will be returned:

<input type="text" name="username" value="{{ old('username') }}">

[Cookies](https://laravel.com/docs/8.x/requests#cookies)

[Retrieving Cookies From Requests](https://laravel.com/docs/8.x/requests#retrieving-cookies-from-requests)

All cookies created by the Laravel framework are encrypted and signed with an authentication code, meaning they will be considered invalid if they have been changed by the client. To retrieve a cookie value from the request, use the cookie method on an Illuminate\Http\Request instance:

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

[Input Trimming & Normalization](https://laravel.com/docs/8.x/requests#input-trimming-and-normalization)

By default, Laravel includes the App\Http\Middleware\TrimStrings and App\Http\Middleware\ConvertEmptyStringsToNull middleware in your application's global middleware stack. These middleware are listed in the global middleware stack by the App\Http\Kernel class. These middleware will automatically trim all incoming string fields on the request, as well as convert any empty string fields to null. This allows you to not have to worry about these normalization concerns in your routes and controllers.

If you would like to disable this behavior, you may remove the two middleware from your application's middleware stack by removing them from the $middleware property of your App\Http\Kernel class.

[Files](https://laravel.com/docs/8.x/requests#files)

[Retrieving Uploaded Files](https://laravel.com/docs/8.x/requests#retrieving-uploaded-files)

You may retrieve uploaded files from an Illuminate\Http\Request instance using the file method or using dynamic properties. The file method returns an instance of the Illuminate\Http\UploadedFile class, which extends the PHP SplFileInfo class and provides a variety of methods for interacting with the file:

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

$file = $request->photo;

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

if ($request->hasFile('photo')) {

//

}

[Validating Successful Uploads](https://laravel.com/docs/8.x/requests#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()) {

//

}

[File Paths & Extensions](https://laravel.com/docs/8.x/requests#file-paths-extensions)

The UploadedFile class also contains methods for accessing the file's fully-qualified path and its extension. The extension method will attempt to guess the file's extension based on its contents. This extension may be different from the extension that was supplied by the client:

$path = $request->photo->path();

$extension = $request->photo->extension();

[Other File Methods](https://laravel.com/docs/8.x/requests#other-file-methods)

There are a variety of other methods available on UploadedFile instances. Check out the [API documentation for the class](https://api.symfony.com/master/Symfony/Component/HttpFoundation/File/UploadedFile.html) for more information regarding these methods.

[Storing Uploaded Files](https://laravel.com/docs/8.x/requests#storing-uploaded-files)

To store an uploaded file, you will typically use one of your configured [filesystems](https://laravel.com/docs/8.x/filesystem). The UploadedFile class has a store method that will move an uploaded file to one of your disks, which may be a location on your local filesystem or a cloud storage location like Amazon S3.

The store method accepts the path where the file should be stored relative to the filesystem's configured root directory. This path should not contain a filename, since a unique ID will automatically be generated to serve as the filename.

The store method also accepts an optional second argument for the name of the disk that should be used to store the file. The method will return the path of the file relative to the disk's root:

$path = $request->photo->store('images');

$path = $request->photo->store('images', 's3');

If you do not want a filename to be automatically generated, you may use the storeAs method, which accepts the path, filename, and disk name as its arguments:

$path = $request->photo->storeAs('images', 'filename.jpg');

$path = $request->photo->storeAs('images', 'filename.jpg', 's3');

For more information about file storage in Laravel, check out the complete [file storage documentation](https://laravel.com/docs/8.x/filesystem).

[Configuring Trusted Proxies](https://laravel.com/docs/8.x/requests#configuring-trusted-proxies)

When running your applications behind a load balancer that terminates TLS / SSL certificates, you may notice your application sometimes does not generate HTTPS links when using the url helper. Typically this is because your application is being forwarded traffic from your load balancer on port 80 and does not know it should generate secure links.

To solve this, you may use the App\Http\Middleware\TrustProxies middleware that is included in your Laravel application, which allows you to quickly customize the load balancers or proxies that should be trusted by your application. Your trusted proxies should be listed as an array on the $proxies property of this middleware. In addition to configuring the trusted proxies, you may configure the proxy $headers that should be trusted:

**<?php**

namespace App\Http\Middleware;

use Illuminate\Http\Middleware\TrustProxies as Middleware;

use Illuminate\Http\Request;

class TrustProxies extends Middleware

{

/\*\*

\* The trusted proxies for this application.

\*

\* @var string|array

\*/

protected $proxies = [

'192.168.1.1',

'192.168.1.2',

];

/\*\*

\* The headers that should be used to detect proxies.

\*

\* @var int

\*/

protected $headers = Request::HEADER\_X\_FORWARDED\_FOR | Request::HEADER\_X\_FORWARDED\_HOST | Request::HEADER\_X\_FORWARDED\_PORT | Request::HEADER\_X\_FORWARDED\_PROTO;

}

If you are using AWS Elastic Load Balancing, your $headers value should be Request::HEADER\_X\_FORWARDED\_AWS\_ELB. For more information on the constants that may be used in the $headers property, check out Symfony's documentation on [trusting proxies](https://symfony.com/doc/current/deployment/proxies.html).

[Trusting All Proxies](https://laravel.com/docs/8.x/requests#trusting-all-proxies)

If you are using Amazon AWS or another "cloud" load balancer provider, you may not know the IP addresses of your actual balancers. In this case, you may use \* to trust all proxies:

/\*\*

\* The trusted proxies for this application.

\*

\* @var string|array

\*/

protected $proxies = '\*';

[Configuring Trusted Hosts](https://laravel.com/docs/8.x/requests#configuring-trusted-hosts)

By default, Laravel will respond to all requests it receives regardless of the content of the HTTP request's Host header. In addition, the Host header's value will be used when generating absolute URLs to your application during a web request.

Typically, you should configure your web server, such as Nginx or Apache, to only send requests to your application that match a given host name. However, if you do not have the ability to customize your web server directly and need to instruct Laravel to only respond to certain host names, you may do so by enabling the App\Http\Middleware\TrustHosts middleware for your application.

The TrustHosts middleware is already included in the $middleware stack of your application; however, you should uncomment it so that it becomes active. Within this middleware's hosts method, you may specify the host names that your application should respond to. Incoming requests with other Host value headers will be rejected:

/\*\*

\* Get the host patterns that should be trusted.

\*

\* @return array

\*/

public function hosts()

{

return [

'laravel.test',

$this->allSubdomainsOfApplicationUrl(),

];

}

The allSubdomainsOfApplicationUrl helper method will return a regular expression matching all subdomains of your application's app.url configuration value. This helper method provides a convenient way to allow all of your application's subdomains when building an application that utilizes wildcard subdomains.

HTTP Responses

* [Creating Responses](https://laravel.com/docs/8.x/responses#creating-responses)
  + [Attaching Headers To Responses](https://laravel.com/docs/8.x/responses#attaching-headers-to-responses)
  + [Attaching Cookies To Responses](https://laravel.com/docs/8.x/responses#attaching-cookies-to-responses)
  + [Cookies & Encryption](https://laravel.com/docs/8.x/responses#cookies-and-encryption)
* [Redirects](https://laravel.com/docs/8.x/responses#redirects)
  + [Redirecting To Named Routes](https://laravel.com/docs/8.x/responses#redirecting-named-routes)
  + [Redirecting To Controller Actions](https://laravel.com/docs/8.x/responses#redirecting-controller-actions)
  + [Redirecting To External Domains](https://laravel.com/docs/8.x/responses#redirecting-external-domains)
  + [Redirecting With Flashed Session Data](https://laravel.com/docs/8.x/responses#redirecting-with-flashed-session-data)
* [Other Response Types](https://laravel.com/docs/8.x/responses#other-response-types)
  + [View Responses](https://laravel.com/docs/8.x/responses#view-responses)
  + [JSON Responses](https://laravel.com/docs/8.x/responses#json-responses)
  + [File Downloads](https://laravel.com/docs/8.x/responses#file-downloads)
  + [File Responses](https://laravel.com/docs/8.x/responses#file-responses)
* [Response Macros](https://laravel.com/docs/8.x/responses#response-macros)

[Creating Responses](https://laravel.com/docs/8.x/responses#creating-responses)

[Strings & Arrays](https://laravel.com/docs/8.x/responses#strings-arrays)

All routes and controllers should return a response to be sent back to the user's browser. Laravel provides several different ways to return responses. The most basic response is returning a string from a route or controller. The framework will automatically convert the string into a full HTTP response:

Route::get('/', function () {

return 'Hello World';

});

In addition to returning strings from your routes and controllers, you may also return arrays. The framework will automatically convert the array into a JSON response:

Route::get('/', function () {

return [1, 2, 3];

});

Did you know you can also return [Eloquent collections](https://laravel.com/docs/8.x/eloquent-collections) from your routes or controllers? They will automatically be converted to JSON. Give it a shot!

[Response Objects](https://laravel.com/docs/8.x/responses#response-objects)

Typically, you won't just be returning simple strings or arrays from your route actions. Instead, you will be returning full Illuminate\Http\Response instances or [views](https://laravel.com/docs/8.x/views).

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, which provides a variety of methods for building HTTP responses:

Route::get('/home', function () {

return response('Hello World', 200)

->header('Content-Type', 'text/plain');

});

[Eloquent Models & Collections](https://laravel.com/docs/8.x/responses#eloquent-models-and-collections)

You may also return [Eloquent ORM](https://laravel.com/docs/8.x/eloquent) models and collections directly from your routes and controllers. When you do, Laravel will automatically convert the models and collections to JSON responses while respecting the model's [hidden attributes](https://laravel.com/docs/8.x/eloquent-serialization#hiding-attributes-from-json):

use App\Models\User;

Route::get('/user/{user}', function (User $user) {

return $user;

});

[Attaching Headers To Responses](https://laravel.com/docs/8.x/responses#attaching-headers-to-responses)

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

return response($content)

->header('Content-Type', $type)

->header('X-Header-One', 'Header Value')

->header('X-Header-Two', 'Header Value');

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

return response($content)

->withHeaders([

'Content-Type' => $type,

'X-Header-One' => 'Header Value',

'X-Header-Two' => 'Header Value',

]);

[Cache Control Middleware](https://laravel.com/docs/8.x/responses#cache-control-middleware)

Laravel includes a cache.headers middleware, which may be used to quickly set the Cache-Control header for a group of routes. Directives should be provided using the "snake case" equivalent of the corresponding cache-control directive and should be separated by a semicolon. If etag is specified in the list of directives, an MD5 hash of the response content will automatically be set as the ETag identifier:

Route::middleware('cache.headers:public;max\_age=2628000;etag')->group(function () {

Route::get('/privacy', function () {

// ...

});

Route::get('/terms', function () {

// ...

});

});

[Attaching Cookies To Responses](https://laravel.com/docs/8.x/responses#attaching-cookies-to-responses)

You may attach a cookie to an outgoing Illuminate\Http\Response instance using the cookie method. You should pass the name, value, and the number of minutes the cookie should be considered valid to this method:

return response('Hello World')->cookie(

'name', 'value', $minutes

);

The cookie method also accepts a few more arguments which are used less frequently. Generally, these arguments have the same purpose and meaning as the arguments that would be given to PHP's native [setcookie](https://secure.php.net/manual/en/function.setcookie.php) method:

return response('Hello World')->cookie(

'name', 'value', $minutes, $path, $domain, $secure, $httpOnly

);

If you would like to ensure that a cookie is sent with the outgoing response but you do not yet have an instance of that response, you can use the Cookie facade to "queue" cookies for attachment to the response when it is sent. The queue method accepts the arguments needed to create a cookie instance. These cookies will be attached to the outgoing response before it is sent to the browser:

use Illuminate\Support\Facades\Cookie;

Cookie::queue('name', 'value', $minutes);

[Generating Cookie Instances](https://laravel.com/docs/8.x/responses#generating-cookie-instances)

If you would like to generate a Symfony\Component\HttpFoundation\Cookie instance that can be attached to a response instance at a later time, you may use the global cookie helper. This cookie will not be sent back to the client unless it is attached to a response instance:

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

return response('Hello World')->cookie($cookie);

[Expiring Cookies Early](https://laravel.com/docs/8.x/responses#expiring-cookies-early)

You may remove a cookie by expiring it via the withoutCookie method of an outgoing response:

return response('Hello World')->withoutCookie('name');

If you do not yet have an instance of the outgoing response, you may use the Cookie facade's expire method to expire a cookie:

Cookie::expire('name');

[Cookies & Encryption](https://laravel.com/docs/8.x/responses#cookies-and-encryption)

By default, all cookies generated by Laravel are encrypted and signed so that they can't be modified or read by the client. If you would like to disable encryption for a subset of cookies generated by your application, you may use the $except property of the App\Http\Middleware\EncryptCookies middleware, which is located in the app/Http/Middleware directory:

/\*\*

\* The names of the cookies that should not be encrypted.

\*

\* @var array

\*/

protected $except = [

'cookie\_name',

];

[Redirects](https://laravel.com/docs/8.x/responses#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:

Route::get('/dashboard', function () {

return redirect('home/dashboard');

});

Sometimes you may wish to redirect the user to their previous location, such as when a submitted form is invalid. You may do so by using the global back helper function. Since this feature utilizes the [session](https://laravel.com/docs/8.x/session), make sure the route calling the back function is using the web middleware group:

Route::post('/user/profile', function () {

// Validate the request...

return back()->withInput();

});

[Redirecting To Named Routes](https://laravel.com/docs/8.x/responses#redirecting-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', ['id' => 1]);

[Populating Parameters Via Eloquent Models](https://laravel.com/docs/8.x/responses#populating-parameters-via-eloquent-models)

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

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

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

If you would like to customize the value that is placed in the route parameter, you can specify the column in the route parameter definition (/profile/{id:slug}) or you can override the getRouteKey method on your Eloquent model:

/\*\*

\* Get the value of the model's route key.

\*

\* @return mixed

\*/

public function getRouteKey()

{

return $this->slug;

}

[Redirecting To Controller Actions](https://laravel.com/docs/8.x/responses#redirecting-controller-actions)

You may also generate redirects to [controller actions](https://laravel.com/docs/8.x/controllers). To do so, pass the controller and action name to the action method:

use App\Http\Controllers\UserController;

return redirect()->action([UserController::class, 'index']);

If your controller route requires parameters, you may pass them as the second argument to the action method:

return redirect()->action(

[UserController::class, 'profile'], ['id' => 1]

);

[Redirecting To External Domains](https://laravel.com/docs/8.x/responses#redirecting-external-domains)

Sometimes you may need to redirect to a domain outside of your application. You may do so by calling the away method, which creates a RedirectResponse without any additional URL encoding, validation, or verification:

return redirect()->away('https://www.google.com');

[Redirecting With Flashed Session Data](https://laravel.com/docs/8.x/responses#redirecting-with-flashed-session-data)

Redirecting to a new URL and [flashing data to the session](https://laravel.com/docs/8.x/session#flash-data) are usually done at the same time. Typically, this is done after successfully performing an action when you flash a success message to the session. For convenience, you may create a RedirectResponse instance and flash data to the session in a single, fluent method chain:

Route::post('/user/profile', function () {

// ...

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

});

After the user is redirected, you may display the flashed message from the [session](https://laravel.com/docs/8.x/session). For example, using [Blade syntax](https://laravel.com/docs/8.x/blade):

@if (session('status'))

<div class="alert alert-success">

{{ session('status') }}

</div>

@endif

[Redirecting With Input](https://laravel.com/docs/8.x/responses#redirecting-with-input)

You may use the withInput method provided by the RedirectResponse instance to flash the current request's input data to the session before redirecting the user to a new location. This is typically done if the user has encountered a validation error. Once the input has been flashed to the session, you may easily [retrieve it](https://laravel.com/docs/8.x/requests#retrieving-old-input) during the next request to repopulate the form:

return back()->withInput();

[Other Response Types](https://laravel.com/docs/8.x/responses#other-response-types)

The response helper may be used to generate other types of response instances. When the response helper is called without arguments, an implementation of the Illuminate\Contracts\Routing\ResponseFactory [contract](https://laravel.com/docs/8.x/contracts) is returned. This contract provides several helpful methods for generating responses.

[View Responses](https://laravel.com/docs/8.x/responses#view-responses)

If you need control over the response's status and headers but also need to return a [view](https://laravel.com/docs/8.x/views) as the response's content, you should use the view method:

return response()

->view('hello', $data, 200)

->header('Content-Type', $type);

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

[JSON Responses](https://laravel.com/docs/8.x/responses#json-responses)

The json method will automatically set the Content-Type header to application/json, as well as convert the given array to 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 combination with the withCallback method:

return response()

->json(['name' => 'Abigail', 'state' => 'CA'])

->withCallback($request->input('callback'));

[File Downloads](https://laravel.com/docs/8.x/responses#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 filename as the second argument to the method, which will determine the filename 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);

Symfony HttpFoundation, which manages file downloads, requires the file being downloaded to have an ASCII filename.

[Streamed Downloads](https://laravel.com/docs/8.x/responses#streamed-downloads)

Sometimes you may wish to turn the string response of a given operation into a downloadable response without having to write the contents of the operation to disk. You may use the streamDownload method in this scenario. This method accepts a callback, filename, and an optional array of headers as its arguments:

use App\Services\GitHub;

return response()->streamDownload(function () {

echo GitHub::api('repo')

->contents()

->readme('laravel', 'laravel')['contents'];

}, 'laravel-readme.md');

[File Responses](https://laravel.com/docs/8.x/responses#file-responses)

The file method may be used to display a file, such as an image or PDF, directly in the user's browser instead of initiating a download. This method accepts the path to the file as its first argument and an array of headers as its second argument:

return response()->file($pathToFile);

return response()->file($pathToFile, $headers);

[Response Macros](https://laravel.com/docs/8.x/responses#response-macros)

If you would like to define a custom response that you can re-use in a variety of your routes and controllers, you may use the macro method on the Response facade. Typically, you should call this method from the boot method of one of your application's [service providers](https://laravel.com/docs/8.x/providers), such as the App\Providers\AppServiceProvider service provider:

**<?php**

namespace App\Providers;

use Illuminate\Support\Facades\Response;

use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider

{

/\*\*

\* Bootstrap any application services.

\*

\* @return void

\*/

public function boot()

{

Response::macro('caps', function ($value) {

return Response::make(strtoupper($value));

});

}

}

The macro function accepts a name as its first argument and a closure as its second argument. The macro's closure will be executed when calling the macro name from a ResponseFactory implementation or the response helper:

return response()->caps('foo');

Views

* [Introduction](https://laravel.com/docs/8.x/views#introduction)
* [Creating & Rendering Views](https://laravel.com/docs/8.x/views#creating-and-rendering-views)
  + [Nested View Directories](https://laravel.com/docs/8.x/views#nested-view-directories)
  + [Creating The First Available View](https://laravel.com/docs/8.x/views#creating-the-first-available-view)
  + [Determining If A View Exists](https://laravel.com/docs/8.x/views#determining-if-a-view-exists)
* [Passing Data To Views](https://laravel.com/docs/8.x/views#passing-data-to-views)
  + [Sharing Data With All Views](https://laravel.com/docs/8.x/views#sharing-data-with-all-views)
* [View Composers](https://laravel.com/docs/8.x/views#view-composers)
  + [View Creators](https://laravel.com/docs/8.x/views#view-creators)
* [Optimizing Views](https://laravel.com/docs/8.x/views#optimizing-views)

[Introduction](https://laravel.com/docs/8.x/views#introduction)

Of course, it's not practical to return entire HTML documents strings directly from your routes and controllers. Thankfully, views provide a convenient way to place all of our HTML in separate files. Views separate your controller / application logic from your presentation logic and are stored in the resources/views directory. A simple view might look something like this:

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

<html>

<body>

<h1>Hello, {{ $name }}</h1>

</body>

</html>

Since this view is stored at resources/views/greeting.blade.php, we may return it using the global view helper like so:

Route::get('/', function () {

return view('greeting', ['name' => 'James']);

});

Looking for more information on how to write Blade templates? Check out the full [Blade documentation](https://laravel.com/docs/8.x/blade) to get started.

[Creating & Rendering Views](https://laravel.com/docs/8.x/views#creating-and-rendering-views)

You may create a view by placing a file with the .blade.php extension in your application's resources/views directory. The .blade.php extension informs the framework that the file contains a [Blade template](https://laravel.com/docs/8.x/blade). Blade templates contain HTML as well as Blade directives that allow you to easily echo values, create "if" statements, iterate over data, and more.

Once you have created a view, you may return it from one of your application's routes or controllers using the global view helper:

Route::get('/', function () {

return view('greeting', ['name' => 'James']);

});

Views may also be returned using the View facade:

use Illuminate\Support\Facades\View;

return View::make('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 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 using [Blade syntax](https://laravel.com/docs/8.x/blade).

[Nested View Directories](https://laravel.com/docs/8.x/views#nested-view-directories)

Views may also be nested within subdirectories 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.blade.php, you may return it from one of your application's routes / controllers like so:

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

View directory names should not contain the . character.

[Creating The First Available View](https://laravel.com/docs/8.x/views#creating-the-first-available-view)

Using the View facade's first method, you may create the first view that exists in a given array of views. This may be useful if your application or package allows views to be customized or overwritten:

use Illuminate\Support\Facades\View;

return View::first(['custom.admin', 'admin'], $data);

[Determining If A View Exists](https://laravel.com/docs/8.x/views#determining-if-a-view-exists)

If you need to determine if a view exists, you may use the View facade. The exists method will return true if the view exists:

use Illuminate\Support\Facades\View;

if (View::exists('emails.customer')) {

//

}

[Passing Data To Views](https://laravel.com/docs/8.x/views#passing-data-to-views)

As you saw in the previous examples, you may pass an array of data to views to make that data available to the view:

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

When passing information in this manner, the data should be an array with key / value pairs. After providing data to a view, you can then access each value within your view using the data's keys, such as <?php echo $name; ?>.

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. The with method returns an instance of the view object so that you can continue chaining methods before returning the view:

return view('greeting')

->with('name', 'Victoria')

->with('occupation', 'Astronaut');

[Sharing Data With All Views](https://laravel.com/docs/8.x/views#sharing-data-with-all-views)

Occasionally, you may need to share data with all views that are rendered by your application. You may do so using the View facade's share method. Typically, you should place calls to the share method within a service provider's boot method. You are free to add them to the App\Providers\AppServiceProvider class or generate a separate service provider to house them:

**<?php**

namespace App\Providers;

use Illuminate\Support\Facades\View;

class AppServiceProvider extends ServiceProvider

{

/\*\*

\* Register any application services.

\*

\* @return void

\*/

public function register()

{

//

}

/\*\*

\* Bootstrap any application services.

\*

\* @return void

\*/

public function boot()

{

View::share('key', 'value');

}

}

[View Composers](https://laravel.com/docs/8.x/views#view-composers)

View composers are callbacks or class methods that are called when a view is rendered. If you have data that you want to be bound to a view each time that view is rendered, a view composer can help you organize that logic into a single location. View composers may prove particularly useful if the same view is returned by multiple routes or controllers within your application and always needs a particular piece of data.

Typically, view composers will be registered within one of your application's [service providers](https://laravel.com/docs/8.x/providers). In this example, we'll assume that we have created a new App\Providers\ViewServiceProvider to house this logic.

We'll use the View facade's composer method to register the view composer. Laravel does not include a default directory for class based view composers, so you are free to organize them however you wish. For example, you could create an app/View/Composers directory to house all of your application's view composers:

**<?php**

namespace App\Providers;

use App\View\Composers\ProfileComposer;

use Illuminate\Support\Facades\View;

use Illuminate\Support\ServiceProvider;

class ViewServiceProvider extends ServiceProvider

{

/\*\*

\* Register any application services.

\*

\* @return void

\*/

public function register()

{

//

}

/\*\*

\* Bootstrap any application services.

\*

\* @return void

\*/

public function boot()

{

// Using class based composers...

View::composer('profile', ProfileComposer::class);

// Using closure based composers...

View::composer('dashboard', function ($view) {

//

});

}

}

Remember, if you create a new service provider to contain your view composer registrations, you will need to add the service provider to the providers array in the config/app.php configuration file.

Now that we have registered the composer, the compose method of the App\View\Composers\ProfileComposer class will be executed each time the profile view is being rendered. Let's take a look at an example of the composer class:

**<?php**

namespace App\View\Composers;

use App\Repositories\UserRepository;

use Illuminate\View\View;

class ProfileComposer

{

/\*\*

\* The user repository implementation.

\*

\* @var \App\Repositories\UserRepository

\*/

protected $users;

/\*\*

\* Create a new profile composer.

\*

\* @param \App\Repositories\UserRepository $users

\* @return void

\*/

public function \_\_construct(UserRepository $users)

{

// Dependencies are automatically resolved by the service container...

$this->users = $users;

}

/\*\*

\* Bind data to the view.

\*

\* @param \Illuminate\View\View $view

\* @return void

\*/

public function compose(View $view)

{

$view->with('count', $this->users->count());

}

}

As you can see, all view composers are resolved via the [service container](https://laravel.com/docs/8.x/container), so you may type-hint any dependencies you need within a composer's constructor.

[Attaching A Composer To Multiple Views](https://laravel.com/docs/8.x/views#attaching-a-composer-to-multiple-views)

You may attach a view composer to multiple views at once by passing an array of views as the first argument to the composer method:

use App\Views\Composers\MultiComposer;

View::composer(

['profile', 'dashboard'],

MultiComposer::class

);

The composer method also accepts the \* character as a wildcard, allowing you to attach a composer to all views:

View::composer('\*', function ($view) {

//

});

[View Creators](https://laravel.com/docs/8.x/views#view-creators)

View "creators" are very similar to view composers; however, they are executed immediately after the view is instantiated instead of waiting until the view is about to render. To register a view creator, use the creator method:

use App\View\Creators\ProfileCreator;

use Illuminate\Support\Facades\View;

View::creator('profile', ProfileCreator::class);

[Optimizing Views](https://laravel.com/docs/8.x/views#optimizing-views)

By default, Blade template views are compiled on demand. When a request is executed that renders a view, Laravel will determine if a compiled version of the view exists. If the file exists, Laravel will then determine if the uncompiled view has been modified more recently than the compiled view. If the compiled view either does not exist, or the uncompiled view has been modified, Laravel will recompile the view.

Compiling views during the request may have a small negative impact on performance, so Laravel provides the view:cache Artisan command to precompile all of the views utilized by your application. For increased performance, you may wish to run this command as part of your deployment process:

php artisan view:cache

You may use the view:clear command to clear the view cache:

php artisan view:clear

Blade Templates

* [Introduction](https://laravel.com/docs/8.x/blade#introduction)
* [Displaying Data](https://laravel.com/docs/8.x/blade#displaying-data)
  + [HTML Entity Encoding](https://laravel.com/docs/8.x/blade#html-entity-encoding)
  + [Blade & JavaScript Frameworks](https://laravel.com/docs/8.x/blade#blade-and-javascript-frameworks)
* [Blade Directives](https://laravel.com/docs/8.x/blade#blade-directives)
  + [If Statements](https://laravel.com/docs/8.x/blade#if-statements)
  + [Switch Statements](https://laravel.com/docs/8.x/blade#switch-statements)
  + [Loops](https://laravel.com/docs/8.x/blade#loops)
  + [The Loop Variable](https://laravel.com/docs/8.x/blade#the-loop-variable)
  + [Conditional Classes](https://laravel.com/docs/8.x/blade#conditional-classes)
  + [Including Subviews](https://laravel.com/docs/8.x/blade#including-subviews)
  + [The @once Directive](https://laravel.com/docs/8.x/blade#the-once-directive)
  + [Raw PHP](https://laravel.com/docs/8.x/blade#raw-php)
  + [Comments](https://laravel.com/docs/8.x/blade#comments)
* [Components](https://laravel.com/docs/8.x/blade#components)
  + [Rendering Components](https://laravel.com/docs/8.x/blade#rendering-components)
  + [Passing Data To Components](https://laravel.com/docs/8.x/blade#passing-data-to-components)
  + [Component Attributes](https://laravel.com/docs/8.x/blade#component-attributes)
  + [Reserved Keywords](https://laravel.com/docs/8.x/blade#reserved-keywords)
  + [Slots](https://laravel.com/docs/8.x/blade#slots)
  + [Inline Component Views](https://laravel.com/docs/8.x/blade#inline-component-views)
  + [Anonymous Components](https://laravel.com/docs/8.x/blade#anonymous-components)
  + [Dynamic Components](https://laravel.com/docs/8.x/blade#dynamic-components)
  + [Manually Registering Components](https://laravel.com/docs/8.x/blade#manually-registering-components)
* [Building Layouts](https://laravel.com/docs/8.x/blade#building-layouts)
  + [Layouts Using Components](https://laravel.com/docs/8.x/blade#layouts-using-components)
  + [Layouts Using Template Inheritance](https://laravel.com/docs/8.x/blade#layouts-using-template-inheritance)
* [Forms](https://laravel.com/docs/8.x/blade#forms)
  + [CSRF Field](https://laravel.com/docs/8.x/blade#csrf-field)
  + [Method Field](https://laravel.com/docs/8.x/blade#method-field)
  + [Validation Errors](https://laravel.com/docs/8.x/blade#validation-errors)
* [Stacks](https://laravel.com/docs/8.x/blade#stacks)
* [Service Injection](https://laravel.com/docs/8.x/blade#service-injection)
* [Extending Blade](https://laravel.com/docs/8.x/blade#extending-blade)
  + [Custom Echo Handlers](https://laravel.com/docs/8.x/blade#custom-echo-handlers)
  + [Custom If Statements](https://laravel.com/docs/8.x/blade#custom-if-statements)

[Introduction](https://laravel.com/docs/8.x/blade#introduction)

Blade is the simple, yet powerful templating engine that is included with Laravel. Unlike some PHP templating engines, Blade does not restrict you from using plain PHP code in your templates. In fact, all Blade templates are compiled into plain PHP code and cached until they are modified, meaning Blade adds essentially zero overhead to your application. Blade template files use the .blade.php file extension and are typically stored in the resources/views directory.

Blade views may be returned from routes or controller using the global view helper. Of course, as mentioned in the documentation on [views](https://laravel.com/docs/8.x/views), data may be passed to the Blade view using the view helper's second argument:

Route::get('/', function () {

return view('greeting', ['name' => 'Finn']);

});

Before digging deeper into Blade, make sure to read the Laravel [view documentation](https://laravel.com/docs/8.x/views).

[Displaying Data](https://laravel.com/docs/8.x/blade#displaying-data)

You may display data that is passed to your Blade views by wrapping the variable in curly braces. For example, given the following route:

Route::get('/', function () {

return view('welcome', ['name' => 'Samantha']);

});

You may display the contents of the name variable like so:

Hello, {{ $name }}.

Blade's {{ }} echo statements are automatically sent through PHP's htmlspecialchars function to prevent XSS attacks.

You are not limited to displaying the contents of the variables passed to the view. You may also echo the results of any PHP function. In fact, you can put any PHP code you wish inside of a Blade echo statement:

The current UNIX timestamp is {{ time() }}.

[Rendering JSON](https://laravel.com/docs/8.x/blade#rendering-json)

Sometimes you may pass an array to your view with the intention of rendering it as JSON in order to initialize a JavaScript variable. For example:

<script>

var app = **<?php** echo json\_encode($array); **?>**;

</script>

However, instead of manually calling json\_encode, you may use the @json Blade directive. The @json directive accepts the same arguments as PHP's json\_encode function. By default, the @json directive calls the json\_encode function with the JSON\_HEX\_TAG, JSON\_HEX\_APOS, JSON\_HEX\_AMP, and JSON\_HEX\_QUOT flags:

<script>

var app = @json($array);

var app = @json($array, JSON\_PRETTY\_PRINT);

</script>

You should only use the @json directive to render existing variables as JSON. The Blade templating is based on regular expressions and attempts to pass a complex expression to the directive may cause unexpected failures.

[HTML Entity Encoding](https://laravel.com/docs/8.x/blade#html-entity-encoding)

By default, Blade (and the Laravel e helper) will double encode HTML entities. If you would like to disable double encoding, call the Blade::withoutDoubleEncoding method from the boot method of your AppServiceProvider:

**<?php**

namespace App\Providers;

use Illuminate\Support\Facades\Blade;

use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider

{

/\*\*

\* Bootstrap any application services.

\*

\* @return void

\*/

public function boot()

{

Blade::withoutDoubleEncoding();

}

}

[Displaying Unescaped Data](https://laravel.com/docs/8.x/blade#displaying-unescaped-data)

By default, Blade {{ }} statements are automatically sent through PHP's htmlspecialchars function to prevent XSS attacks. If you do not want your data to be escaped, you may use the following syntax:

Hello, {!! $name !!}.

Be very careful when echoing content that is supplied by users of your application. You should typically use the escaped, double curly brace syntax to prevent XSS attacks when displaying user supplied data.

[Blade & JavaScript Frameworks](https://laravel.com/docs/8.x/blade#blade-and-javascript-frameworks)

Since many JavaScript frameworks also use "curly" braces to indicate a given expression should be displayed in the browser, you may use the @ symbol to inform the Blade rendering engine an expression should remain untouched. For example:

<h1>Laravel</h1>

Hello, @{{ name }}.

In this example, the @ symbol will be removed by Blade; however, {{ name }} expression will remain untouched by the Blade engine, allowing it to be rendered by your JavaScript framework.

The @ symbol may also be used to escape Blade directives:

{{-- Blade template --}}

@@json()

<!-- HTML output -->

@json()

[The @verbatim Directive](https://laravel.com/docs/8.x/blade#the-at-verbatim-directive)

If you are displaying JavaScript variables in a large portion of your template, you may wrap the HTML in the @verbatim directive so that you do not have to prefix each Blade echo statement with an @ symbol:

@verbatim

<div class="container">

Hello, {{ name }}.

</div>

@endverbatim

[Blade Directives](https://laravel.com/docs/8.x/blade#blade-directives)

In addition to template inheritance and displaying data, Blade also provides convenient shortcuts for common PHP control structures, such as conditional statements and loops. These shortcuts provide a very clean, terse way of working with PHP control structures while also remaining familiar to their PHP counterparts.

[If Statements](https://laravel.com/docs/8.x/blade#if-statements)

You may construct if statements using the @if, @elseif, @else, and @endif directives. These directives function identically to their PHP counterparts:

@if (count($records) === 1)

I have one record!

@elseif (count($records) > 1)

I have multiple records!

@else

I don't have any records!

@endif

For convenience, Blade also provides an @unless directive:

@unless (Auth::check())

You are not signed in.

@endunless

In addition to the conditional directives already discussed, the @isset and @empty directives may be used as convenient shortcuts for their respective PHP functions:

@isset($records)

// $records is defined and is not null...

@endisset

@empty($records)

// $records is "empty"...

@endempty

[Authentication Directives](https://laravel.com/docs/8.x/blade#authentication-directives)

The @auth and @guest directives may be used to quickly determine if the current user is [authenticated](https://laravel.com/docs/8.x/authentication) or is a guest:

@auth

// The user is authenticated...

@endauth

@guest

// The user is not authenticated...

@endguest

If needed, you may specify the authentication guard that should be checked when using the @auth and @guest directives:

@auth('admin')

// The user is authenticated...

@endauth

@guest('admin')

// The user is not authenticated...

@endguest

[Environment Directives](https://laravel.com/docs/8.x/blade#environment-directives)

You may check if the application is running in the production environment using the @production directive:

@production

// Production specific content...

@endproduction

Or, you may determine if the application is running in a specific environment using the @env directive:

@env('staging')

// The application is running in "staging"...

@endenv

@env(['staging', 'production'])

// The application is running in "staging" or "production"...

@endenv

[Section Directives](https://laravel.com/docs/8.x/blade#section-directives)

You may determine if a template inheritance section has content using the @hasSection directive:

@hasSection('navigation')

<div class="pull-right">

@yield('navigation')

</div>

<div class="clearfix"></div>

@endif

You may use the sectionMissing directive to determine if a section does not have content:

@sectionMissing('navigation')

<div class="pull-right">

@include('default-navigation')

</div>

@endif

[Switch Statements](https://laravel.com/docs/8.x/blade#switch-statements)

Switch statements can be constructed using the @switch, @case, @break, @default and @endswitch directives:

@switch($i)

@case(1)

First case...

@break

@case(2)

Second case...

@break

@default

Default case...

@endswitch

[Loops](https://laravel.com/docs/8.x/blade#loops)

In addition to conditional statements, Blade provides simple directives for working with PHP's loop structures. Again, each of these directives functions identically to their PHP counterparts:

@for ($i = 0; $i < 10; $i++)

The current value is {{ $i }}

@endfor

@foreach ($users as $user)

<p>This is user {{ $user->id }}</p>

@endforeach

@forelse ($users as $user)

<li>{{ $user->name }}</li>

@empty

<p>No users</p>

@endforelse

@while (true)

<p>I'm looping forever.</p>

@endwhile

When looping, you may use the [loop variable](https://laravel.com/docs/8.x/blade#the-loop-variable) to gain valuable information about the loop, such as whether you are in the first or last iteration through the loop.

When using loops you may also end the loop or skip the current iteration using the @continue and @break directives:

@foreach ($users as $user)

@if ($user->type == 1)

@continue

@endif

<li>{{ $user->name }}</li>

@if ($user->number == 5)

@break

@endif

@endforeach

You may also include the continuation or break condition within the directive declaration:

@foreach ($users as $user)

@continue($user->type == 1)

<li>{{ $user->name }}</li>

@break($user->number == 5)

@endforeach

[The Loop Variable](https://laravel.com/docs/8.x/blade#the-loop-variable)

When looping, a $loop variable will be available inside of your loop. This variable provides access to some useful bits of information such as the current loop index and whether this is the first or last iteration through the loop:

@foreach ($users as $user)

@if ($loop->first)

This is the first iteration.

@endif

@if ($loop->last)

This is the last iteration.

@endif

<p>This is user {{ $user->id }}</p>

@endforeach

If you are in a nested loop, you may access the parent loop's $loop variable via the parent property:

@foreach ($users as $user)

@foreach ($user->posts as $post)

@if ($loop->parent->first)

This is the first iteration of the parent loop.

@endif

@endforeach

@endforeach

The $loop variable also contains a variety of other useful properties:

| **Property** | **Description** |
| --- | --- |
| $loop->index | The index of the current loop iteration (starts at 0). |
| $loop->iteration | The current loop iteration (starts at 1). |
| $loop->remaining | The iterations remaining in the loop. |
| $loop->count | The total number of items in the array being iterated. |
| $loop->first | Whether this is the first iteration through the loop. |
| $loop->last | Whether this is the last iteration through the loop. |
| $loop->even | Whether this is an even iteration through the loop. |
| $loop->odd | Whether this is an odd iteration through the loop. |
| $loop->depth | The nesting level of the current loop. |
| $loop->parent | When in a nested loop, the parent's loop variable. |

[Conditional Classes](https://laravel.com/docs/8.x/blade#conditional-classes)

The @class directive conditionally compiles a CSS class string. The directive accepts an array of classes where the array key contains the class or classes you wish to add, while the value is a boolean expression. If the array element has a numeric key, it will always be included in the rendered class list:

@php

$isActive = false;

$hasError = true;

@endphp

<span @class([

'p-4',

'font-bold' => $isActive,

'text-gray-500' => ! $isActive,

'bg-red' => $hasError,

])></span>

<span class="p-4 text-gray-500 bg-red"></span>

[Including Subviews](https://laravel.com/docs/8.x/blade#including-subviews)

While you're free to use the @include directive, Blade [components](https://laravel.com/docs/8.x/blade#components) provide similar functionality and offer several benefits over the @include directive such as data and attribute binding.

Blade's @include directive allows you to include a Blade view from within another view. All variables that are available to the parent view will be made available to the included view:

<div>

@include('shared.errors')

<form>

<!-- Form Contents -->

</form>

</div>

Even though the included view will inherit all data available in the parent view, you may also pass an array of additional data that should be made available to the included view:

@include('view.name', ['status' => 'complete'])

If you attempt to @include a view which does not exist, Laravel will throw an error. If you would like to include a view that may or may not be present, you should use the @includeIf directive:

@includeIf('view.name', ['status' => 'complete'])

If you would like to @include a view if a given boolean expression evaluates to true or false, you may use the @includeWhen and @includeUnless directives:

@includeWhen($boolean, 'view.name', ['status' => 'complete'])

@includeUnless($boolean, 'view.name', ['status' => 'complete'])

To include the first view that exists from a given array of views, you may use the includeFirst directive:

@includeFirst(['custom.admin', 'admin'], ['status' => 'complete'])

You should avoid using the \_\_DIR\_\_ and \_\_FILE\_\_ constants in your Blade views, since they will refer to the location of the cached, compiled view.

[Rendering Views For Collections](https://laravel.com/docs/8.x/blade#rendering-views-for-collections)

You may combine loops and includes into one line with Blade's @each directive:

@each('view.name', $jobs, 'job')

The @each directive's first argument is the view to render for each element in the array or collection. The second argument is the array or collection you wish to iterate over, while the third argument is the variable name that will be assigned to the current iteration within the view. So, for example, if you are iterating over an array of jobs, typically you will want to access each job as a job variable within the view. The array key for the current iteration will be available as the key variable within the view.

You may also pass a fourth argument to the @each directive. This argument determines the view that will be rendered if the given array is empty.

@each('view.name', $jobs, 'job', 'view.empty')

Views rendered via @each do not inherit the variables from the parent view. If the child view requires these variables, you should use the @foreach and @include directives instead.

[The @once Directive](https://laravel.com/docs/8.x/blade#the-once-directive)

The @once directive allows you to define a portion of the template that will only be evaluated once per rendering cycle. This may be useful for pushing a given piece of JavaScript into the page's header using [stacks](https://laravel.com/docs/8.x/blade#stacks). For example, if you are rendering a given [component](https://laravel.com/docs/8.x/blade#components) within a loop, you may wish to only push the JavaScript to the header the first time the component is rendered:

@once

@push('scripts')

<script>

// Your custom JavaScript...

</script>

@endpush

@endonce

[Raw PHP](https://laravel.com/docs/8.x/blade#raw-php)

In some situations, it's useful to embed PHP code into your views. You can use the Blade @php directive to execute a block of plain PHP within your template:

@php

$counter = 1;

@endphp

[Comments](https://laravel.com/docs/8.x/blade#comments)

Blade also allows you to define comments in your views. However, unlike HTML comments, Blade comments are not included in the HTML returned by your application:

{{-- This comment will not be present in the rendered HTML --}}

[Components](https://laravel.com/docs/8.x/blade#components)

Components and slots provide similar benefits to sections, layouts, and includes; however, some may find the mental model of components and slots easier to understand. There are two approaches to writing components: class based components and anonymous components.

To create a class based component, you may use the make:component Artisan command. To illustrate how to use components, we will create a simple Alert component. The make:component command will place the component in the App\View\Components directory:

php artisan make:component Alert

The make:component command will also create a view template for the component. The view will be placed in the resources/views/components directory. When writing components for your own application, components are automatically discovered within the app/View/Components directory and resources/views/components directory, so no further component registration is typically required.

You may also create components within subdirectories:

php artisan make:component Forms/Input

The command above will create an Input component in the App\View\Components\Forms directory and the view will be placed in the resources/views/components/forms directory.

[Manually Registering Package Components](https://laravel.com/docs/8.x/blade#manually-registering-package-components)

When writing components for your own application, components are automatically discovered within the app/View/Components directory and resources/views/components directory.

However, if you are building a package that utilizes Blade components, you will need to manually register your component class and its HTML tag alias. You should typically register your components in the boot method of your package's service provider:

use Illuminate\Support\Facades\Blade;

/\*\*

\* Bootstrap your package's services.

\*/

public function boot()

{

Blade::component('package-alert', Alert::class);

}

Once your component has been registered, it may be rendered using its tag alias:

<x-package-alert/>

Alternatively, you may use the componentNamespace method to autoload component classes by convention. For example, a Nightshade package might have Calendar and ColorPicker components that reside within the Package\Views\Components namespace:

use Illuminate\Support\Facades\Blade;

/\*\*

\* Bootstrap your package's services.

\*

\* @return void

\*/

public function boot()

{

Blade::componentNamespace('Nightshade\\Views\\Components', 'nightshade');

}

This will allow the usage of package components by their vendor namespace using the package-name:: syntax:

<x-nightshade::calendar />

<x-nightshade::color-picker />

Blade will automatically detect the class that's linked to this component by pascal-casing the component name. Subdirectories are also supported using "dot" notation.

[Rendering Components](https://laravel.com/docs/8.x/blade#rendering-components)

To display a component, you may use a Blade component tag within one of your Blade templates. Blade component tags start with the string x- followed by the kebab case name of the component class:

<x-alert/>

<x-user-profile/>

If the component class is nested deeper within the App\View\Components directory, you may use the . character to indicate directory nesting. For example, if we assume a component is located at App\View\Components\Inputs\Button.php, we may render it like so:

<x-inputs.button/>

[Passing Data To Components](https://laravel.com/docs/8.x/blade#passing-data-to-components)

You may pass data to Blade components using HTML attributes. Hard-coded, primitive values may be passed to the component using simple HTML attribute strings. PHP expressions and variables should be passed to the component via attributes that use the : character as a prefix:

<x-alert type="error" :message="$message"/>

You should define the component's required data in its class constructor. All public properties on a component will automatically be made available to the component's view. It is not necessary to pass the data to the view from the component's render method:

**<?php**

namespace App\View\Components;

use Illuminate\View\Component;

class Alert extends Component

{

/\*\*

\* The alert type.

\*

\* @var string

\*/

public $type;

/\*\*

\* The alert message.

\*

\* @var string

\*/

public $message;

/\*\*

\* Create the component instance.

\*

\* @param string $type

\* @param string $message

\* @return void

\*/

public function \_\_construct($type, $message)

{

$this->type = $type;

$this->message = $message;

}

/\*\*

\* Get the view / contents that represent the component.

\*

\* @return \Illuminate\View\View|\Closure|string

\*/

public function render()

{

return view('components.alert');

}

}

When your component is rendered, you may display the contents of your component's public variables by echoing the variables by name:

<div class="alert alert-{{ $type }}">

{{ $message }}

</div>

[Casing](https://laravel.com/docs/8.x/blade#casing)

Component constructor arguments should be specified using camelCase, while kebab-case should be used when referencing the argument names in your HTML attributes. For example, given the following component constructor:

/\*\*

\* Create the component instance.

\*

\* @param string $alertType

\* @return void

\*/

public function \_\_construct($alertType)

{

$this->alertType = $alertType;

}

The $alertType argument may be provided to the component like so:

<x-alert alert-type="danger" />

[Escaping Attribute Rendering](https://laravel.com/docs/8.x/blade#escaping-attribute-rendering)

Since some JavaScript frameworks such as Alpine.js also use colon-prefixed attributes, you may use a double colon (::) prefix to inform Blade that the attribute is not a PHP expression. For example, given the following component:

<x-button ::class="{ danger: isDeleting }">

Submit

</x-button>

The following HTML will be rendered by Blade:

<button :class="{ danger: isDeleting }">

Submit

</button>

[Component Methods](https://laravel.com/docs/8.x/blade#component-methods)

In addition to public variables being available to your component template, any public methods on the component may be invoked. For example, imagine a component that has an isSelected method:

/\*\*

\* Determine if the given option is the currently selected option.

\*

\* @param string $option

\* @return bool

\*/

public function isSelected($option)

{

return $option === $this->selected;

}

You may execute this method from your component template by invoking the variable matching the name of the method:

<option {{ $isSelected($value) ? 'selected="selected"' : '' }} value="{{ $value }}">

{{ $label }}

</option>

[Accessing Attributes & Slots Within Component Classes](https://laravel.com/docs/8.x/blade#using-attributes-slots-within-component-class)

Blade components also allow you to access the component name, attributes, and slot inside the class's render method. However, in order to access this data, you should return a closure from your component's render method. The closure will receive a $data array as its only argument. This array will contain several elements that provide information about the component:

/\*\*

\* Get the view / contents that represent the component.

\*

\* @return \Illuminate\View\View|\Closure|string

\*/

public function render()

{

return function (array $data) {

// $data['componentName'];

// $data['attributes'];

// $data['slot'];

return '<div>Components content</div>';

};

}

The componentName is equal to the name used in the HTML tag after the x- prefix. So <x-alert />'s componentName will be alert. The attributes element will contain all of the attributes that were present on the HTML tag. The slot element is an Illuminate\Support\HtmlString instance with the contents of the component's slot.

The closure should return a string. If the returned string corresponds to an existing view, that view will be rendered; otherwise, the returned string will be evaluated as an inline Blade view.

[Additional Dependencies](https://laravel.com/docs/8.x/blade#additional-dependencies)

If your component requires dependencies from Laravel's [service container](https://laravel.com/docs/8.x/container), you may list them before any of the component's data attributes and they will automatically be injected by the container:

use App\Services\AlertCreator

/\*\*

\* Create the component instance.

\*

\* @param \App\Services\AlertCreator $creator

\* @param string $type

\* @param string $message

\* @return void

\*/

public function \_\_construct(AlertCreator $creator, $type, $message)

{

$this->creator = $creator;

$this->type = $type;

$this->message = $message;

}

[Hiding Attributes / Methods](https://laravel.com/docs/8.x/blade#hiding-attributes-and-methods)

If you would like to prevent some public methods or properties from being exposed as variables to your component template, you may add them to an $except array property on your component:

**<?php**

namespace App\View\Components;

use Illuminate\View\Component;

class Alert extends Component

{

/\*\*

\* The alert type.

\*

\* @var string

\*/

public $type;

/\*\*

\* The properties / methods that should not be exposed to the component template.

\*

\* @var array

\*/

protected $except = ['type'];

}

[Component Attributes](https://laravel.com/docs/8.x/blade#component-attributes)

We've already examined how to pass data attributes to a component; however, sometimes you may need to specify additional HTML attributes, such as class, that are not part of the data required for a component to function. Typically, you want to pass these additional attributes down to the root element of the component template. For example, imagine we want to render an alert component like so:

<x-alert type="error" :message="$message" class="mt-4"/>

All of the attributes that are not part of the component's constructor will automatically be added to the component's "attribute bag". This attribute bag is automatically made available to the component via the $attributes variable. All of the attributes may be rendered within the component by echoing this variable:

<div {{ $attributes }}>

<!-- Component content -->

</div>

Using directives such as @env within component tags is not supported at this time. For example, <x-alert :live="@env('production')"/> will not be compiled.

[Default / Merged Attributes](https://laravel.com/docs/8.x/blade#default-merged-attributes)

Sometimes you may need to specify default values for attributes or merge additional values into some of the component's attributes. To accomplish this, you may use the attribute bag's merge method. This method is particularly useful for defining a set of default CSS classes that should always be applied to a component:

<div {{ $attributes->merge(['class' => 'alert alert-'.$type]) }}>

{{ $message }}

</div>

If we assume this component is utilized like so:

<x-alert type="error" :message="$message" class="mb-4"/>

The final, rendered HTML of the component will appear like the following:

<div class="alert alert-error mb-4">

<!-- Contents of the $message variable -->

</div>

[Conditionally Merge Classes](https://laravel.com/docs/8.x/blade#conditionally-merge-classes)

Sometimes you may wish to merge classes if a given condition is true. You can accomplish this via the class method, which accepts an array of classes where the array key contains the class or classes you wish to add, while the value is a boolean expression. If the array element has a numeric key, it will always be included in the rendered class list:

<div {{ $attributes->class(['p-4', 'bg-red' => $hasError]) }}>

{{ $message }}

</div>

If you need to merge other attributes onto your component, you can chain the merge method onto the class method:

<button {{ $attributes->class(['p-4'])->merge(['type' => 'button']) }}>

{{ $slot }}

</button>

If you need to conditionally compile classes on other HTML elements that shouldn't receive merged attributes, you can use the [@class directive](https://laravel.com/docs/8.x/blade#conditional-classes).

[Non-Class Attribute Merging](https://laravel.com/docs/8.x/blade#non-class-attribute-merging)

When merging attributes that are not class attributes, the values provided to the merge method will be considered the "default" values of the attribute. However, unlike the class attribute, these attributes will not be merged with injected attribute values. Instead, they will be overwritten. For example, a button component's implementation may look like the following:

<button {{ $attributes->merge(['type' => 'button']) }}>

{{ $slot }}

</button>

To render the button component with a custom type, it may be specified when consuming the component. If no type is specified, the button type will be used:

<x-button type="submit">

Submit

</x-button>

The rendered HTML of the button component in this example would be:

<button type="submit">

Submit

</button>

If you would like an attribute other than class to have its default value and injected values joined together, you may use the prepends method. In this example, the data-controller attribute will always begin with profile-controller and any additional injected data-controller values will be placed after this default value:

<div {{ $attributes->merge(['data-controller' => $attributes->prepends('profile-controller')]) }}>

{{ $slot }}

</div>

[Retrieving & Filtering Attributes](https://laravel.com/docs/8.x/blade#filtering-attributes)

You may filter attributes using the filter method. This method accepts a closure which should return true if you wish to retain the attribute in the attribute bag:

{{ $attributes->filter(fn ($value, $key) => $key == 'foo') }}

For convenience, you may use the whereStartsWith method to retrieve all attributes whose keys begin with a given string:

{{ $attributes->whereStartsWith('wire:model') }}

Conversely, the whereDoesntStartWith method may be used to exclude all attributes whose keys begin with a given string:

{{ $attributes->whereDoesntStartWith('wire:model') }}

Using the first method, you may render the first attribute in a given attribute bag:

{{ $attributes->whereStartsWith('wire:model')->first() }}

If you would like to check if an attribute is present on the component, you may use the has method. This method accepts the attribute name as its only argument and returns a boolean indicating whether or not the attribute is present:

@if ($attributes->has('class'))

<div>Class attribute is present</div>

@endif

You may retrieve a specific attribute's value using the get method:

{{ $attributes->get('class') }}

[Reserved Keywords](https://laravel.com/docs/8.x/blade#reserved-keywords)

By default, some keywords are reserved for Blade's internal use in order to render components. The following keywords cannot be defined as public properties or method names within your components:

* data
* render
* resolveView
* shouldRender
* view
* withAttributes
* withName

[Slots](https://laravel.com/docs/8.x/blade#slots)

You will often need to pass additional content to your component via "slots". Component slots are rendered by echoing the $slot variable. To explore this concept, let's imagine that an alert component has the following markup:

<!-- /resources/views/components/alert.blade.php -->

<div class="alert alert-danger">

{{ $slot }}

</div>

We may pass content to the slot by injecting content into the component:

<x-alert>

<strong>Whoops!</strong> Something went wrong!

</x-alert>

Sometimes a component may need to render multiple different slots in different locations within the component. Let's modify our alert component to allow for the injection of a "title" slot:

<!-- /resources/views/components/alert.blade.php -->

<span class="alert-title">{{ $title }}</span>

<div class="alert alert-danger">

{{ $slot }}

</div>

You may define the content of the named slot using the x-slot tag. Any content not within an explicit x-slot tag will be passed to the component in the $slot variable:

<x-alert>

<x-slot name="title">

Server Error

</x-slot>

<strong>Whoops!</strong> Something went wrong!

</x-alert>

[Scoped Slots](https://laravel.com/docs/8.x/blade#scoped-slots)

If you have used a JavaScript framework such as Vue, you may be familiar with "scoped slots", which allow you to access data or methods from the component within your slot. You may achieve similar behavior in Laravel by defining public methods or properties on your component and accessing the component within your slot via the $component variable. In this example, we will assume that the x-alert component has a public formatAlert method defined on its component class:

<x-alert>

<x-slot name="title">

{{ $component->formatAlert('Server Error') }}

</x-slot>

<strong>Whoops!</strong> Something went wrong!

</x-alert>

[Slot Attributes](https://laravel.com/docs/8.x/blade#slot-attributes)

Like Blade components, you may assign additional [attributes](https://laravel.com/docs/8.x/blade#component-attributes) to slots such as CSS class names:

<x-card class="shadow-sm">

<x-slot name="heading" class="font-bold">

Heading

</x-slot>

Content

<x-slot name="footer" class="text-sm">

Footer

</x-slot>

</x-card>

To interact with slot attributes, you may access the attributes property of the slot's variable. For more information on how to interact with attributes, please consult the documentation on [component attributes](https://laravel.com/docs/8.x/blade#component-attributes):

@props([

'heading',

'footer',

])

<div {{ $attributes->class(['border']) }}>

<h1 {{ $heading->attributes->class(['text-lg']) }}>

{{ $heading }}

</h1>

{{ $slot }}

<footer {{ $footer->attributes->class(['text-gray-700']) }}>

{{ $footer }}

</footer>

</div>

[Inline Component Views](https://laravel.com/docs/8.x/blade#inline-component-views)

For very small components, it may feel cumbersome to manage both the component class and the component's view template. For this reason, you may return the component's markup directly from the render method:

/\*\*

\* Get the view / contents that represent the component.

\*

\* @return \Illuminate\View\View|\Closure|string

\*/

public function render()

{

return <<<'blade'

<div class="alert alert-danger">

{{ $slot }}

</div>

blade;

}

[Generating Inline View Components](https://laravel.com/docs/8.x/blade#generating-inline-view-components)

To create a component that renders an inline view, you may use the inline option when executing the make:component command:

php artisan make:component Alert --inline

[Anonymous Components](https://laravel.com/docs/8.x/blade#anonymous-components)

Similar to inline components, anonymous components provide a mechanism for managing a component via a single file. However, anonymous components utilize a single view file and have no associated class. To define an anonymous component, you only need to place a Blade template within your resources/views/components directory. For example, assuming you have defined a component at resources/views/components/alert.blade.php, you may simply render it like so:

<x-alert/>

You may use the . character to indicate if a component is nested deeper inside the components directory. For example, assuming the component is defined at resources/views/components/inputs/button.blade.php, you may render it like so:

<x-inputs.button/>

[Anonymous Index Components](https://laravel.com/docs/8.x/blade#anonymous-index-components)

Sometimes, when a component is made up of many Blade templates, you may wish to group the given component's templates within a single directory. For example, imagine an "accordion" component with the following directory structure:

/resources/views/components/accordion.blade.php

/resources/views/components/accordion/item.blade.php

This directory structure allows you to render the accordion component and its item like so:

<x-accordion>

<x-accordion.item>

...

</x-accordion.item>

</x-accordion>

However, in order to render the accordion component via x-accordion, we were forced to place the "index" accordion component template in the resources/views/components directory instead of nesting it within the accordion directory with the other accordion related templates.

Thankfully, Blade allows you to place an index.blade.php file within a component's template directory. When an index.blade.php template exists for the component, it will be rendered as the "root" node of the component. So, we can continue to use the same Blade syntax given in the example above; however, we will adjust our directory structure like so:

/resources/views/components/accordion/index.blade.php

/resources/views/components/accordion/item.blade.php

[Data Properties / Attributes](https://laravel.com/docs/8.x/blade#data-properties-attributes)

Since anonymous components do not have any associated class, you may wonder how you may differentiate which data should be passed to the component as variables and which attributes should be placed in the component's [attribute bag](https://laravel.com/docs/8.x/blade#component-attributes).

You may specify which attributes should be considered data variables using the @props directive at the top of your component's Blade template. All other attributes on the component will be available via the component's attribute bag. If you wish to give a data variable a default value, you may specify the variable's name as the array key and the default value as the array value:

<!-- /resources/views/components/alert.blade.php -->

@props(['type' => 'info', 'message'])

<div {{ $attributes->merge(['class' => 'alert alert-'.$type]) }}>

{{ $message }}

</div>

Given the component definition above, we may render the component like so:

<x-alert type="error" :message="$message" class="mb-4"/>

[Dynamic Components](https://laravel.com/docs/8.x/blade#dynamic-components)

Sometimes you may need to render a component but not know which component should be rendered until runtime. In this situation, you may use Laravel's built-in dynamic-component component to render the component based on a runtime value or variable:

<x-dynamic-component :component="$componentName" class="mt-4" />

[Manually Registering Components](https://laravel.com/docs/8.x/blade#manually-registering-components)

The following documentation on manually registering components is primarily applicable to those who are writing Laravel packages that include view components. If you are not writing a package, this portion of the component documentation may not be relevant to you.

When writing components for your own application, components are automatically discovered within the app/View/Components directory and resources/views/components directory.

However, if you are building a package that utilizes Blade components or placing components in non-conventional directories, you will need to manually register your component class and its HTML tag alias so that Laravel knows where to find the component. You should typically register your components in the boot method of your package's service provider:

use Illuminate\Support\Facades\Blade;

use VendorPackage\View\Components\AlertComponent;

/\*\*

\* Bootstrap your package's services.

\*

\* @return void

\*/

public function boot()

{

Blade::component('package-alert', AlertComponent::class);

}

Once your component has been registered, it may be rendered using its tag alias:

<x-package-alert/>

Autoloading Package Components

Alternatively, you may use the componentNamespace method to autoload component classes by convention. For example, a Nightshade package might have Calendar and ColorPicker components that reside within the Package\Views\Components namespace:

use Illuminate\Support\Facades\Blade;

/\*\*

\* Bootstrap your package's services.

\*

\* @return void

\*/

public function boot()

{

Blade::componentNamespace('Nightshade\\Views\\Components', 'nightshade');

}

This will allow the usage of package components by their vendor namespace using the package-name:: syntax:

<x-nightshade::calendar />

<x-nightshade::color-picker />

Blade will automatically detect the class that's linked to this component by pascal-casing the component name. Subdirectories are also supported using "dot" notation.

[Building Layouts](https://laravel.com/docs/8.x/blade#building-layouts)

[Layouts Using Components](https://laravel.com/docs/8.x/blade#layouts-using-components)

Most web applications maintain the same general layout across various pages. It would be incredibly cumbersome and hard to maintain our application if we had to repeat the entire layout HTML in every view we create. Thankfully, it's convenient to define this layout as a single [Blade component](https://laravel.com/docs/8.x/blade#components) and then use it throughout our application.

[Defining The Layout Component](https://laravel.com/docs/8.x/blade#defining-the-layout-component)

For example, imagine we are building a "todo" list application. We might define a layout component that looks like the following:

<!-- resources/views/components/layout.blade.php -->

<html>

<head>

<title>{{ $title ?? 'Todo Manager' }}</title>

</head>

<body>

<h1>Todos</h1>

<hr/>

{{ $slot }}

</body>

</html>

[Applying The Layout Component](https://laravel.com/docs/8.x/blade#applying-the-layout-component)

Once the layout component has been defined, we may create a Blade view that utilizes the component. In this example, we will define a simple view that displays our task list:

<!-- resources/views/tasks.blade.php -->

<x-layout>

@foreach ($tasks as $task)

{{ $task }}

@endforeach

</x-layout>

Remember, content that is injected into a component will be supplied to the default $slot variable within our layout component. As you may have noticed, our layout also respects a $title slot if one is provided; otherwise, a default title is shown. We may inject a custom title from our task list view using the standard slot syntax discussed in the [component documentation](https://laravel.com/docs/8.x/blade#components):

<!-- resources/views/tasks.blade.php -->

<x-layout>

<x-slot name="title">

Custom Title

</x-slot>

@foreach ($tasks as $task)

{{ $task }}

@endforeach

</x-layout>

Now that we have defined our layout and task list views, we just need to return the task view from a route:

use App\Models\Task;

Route::get('/tasks', function () {

return view('tasks', ['tasks' => Task::all()]);

});

[Layouts Using Template Inheritance](https://laravel.com/docs/8.x/blade#layouts-using-template-inheritance)

[Defining A Layout](https://laravel.com/docs/8.x/blade#defining-a-layout)

Layouts may also be created via "template inheritance". This was the primary way of building applications prior to the introduction of [components](https://laravel.com/docs/8.x/blade#components).

To get started, let's take a look at a simple example. First, we will examine a page layout. Since most web applications maintain the same general layout across various pages, it's convenient to define this layout as a single Blade view:

<!-- resources/views/layouts/app.blade.php -->

<html>

<head>

<title>App Name - @yield('title')</title>

</head>

<body>

@section('sidebar')

This is the master sidebar.

@show

<div class="container">

@yield('content')

</div>

</body>

</html>

As you can see, this file contains typical HTML mark-up. However, take note of the @section and @yield directives. The @section directive, as the name implies, defines a section of content, while the @yield directive is used to display the contents of a given section.

Now that we have defined a layout for our application, let's define a child page that inherits the layout.

[Extending A Layout](https://laravel.com/docs/8.x/blade#extending-a-layout)

When defining a child view, use the @extends Blade directive to specify which layout the child view should "inherit". Views which extend a Blade layout may inject content into the layout's sections using @section directives. Remember, as seen in the example above, the contents of these sections will be displayed in the layout using @yield:

<!-- resources/views/child.blade.php -->

@extends('layouts.app')

@section('title', 'Page Title')

@section('sidebar')

@parent

<p>This is appended to the master sidebar.</p>

@endsection

@section('content')

<p>This is my body content.</p>

@endsection

In this example, the sidebar section is utilizing the @parent directive to append (rather than overwriting) content to the layout's sidebar. The @parent directive will be replaced by the content of the layout when the view is rendered.

Contrary to the previous example, this sidebar section ends with @endsection instead of @show. The @endsection directive will only define a section while @show will define and immediately yield the section.

The @yield directive also accepts a default value as its second parameter. This value will be rendered if the section being yielded is undefined:

@yield('content', 'Default content')

[Forms](https://laravel.com/docs/8.x/blade#forms)

[CSRF Field](https://laravel.com/docs/8.x/blade#csrf-field)

Anytime you define an HTML form in your application, you should include a hidden CSRF token field in the form so that [the CSRF protection](https://laravel.com/docs/8.x/csrf) middleware can validate the request. You may use the @csrf Blade directive to generate the token field:

<form method="POST" action="/profile">

@csrf

...

</form>

[Method Field](https://laravel.com/docs/8.x/blade#method-field)

Since HTML forms can't make PUT, PATCH, or DELETE requests, you will need to add a hidden \_method field to spoof these HTTP verbs. The @method Blade directive can create this field for you:

<form action="/foo/bar" method="POST">

@method('PUT')

...

</form>

[Validation Errors](https://laravel.com/docs/8.x/blade#validation-errors)

The @error directive may be used to quickly check if [validation error messages](https://laravel.com/docs/8.x/validation#quick-displaying-the-validation-errors) exist for a given attribute. Within an @error directive, you may echo the $message variable to display the error message:

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

<label for="title">Post Title</label>

<input id="title" type="text" class="@error('title') is-invalid @enderror">

@error('title')

<div class="alert alert-danger">{{ $message }}</div>

@enderror

Since the @error directive compiles to an "if" statement, you may use the @else directive to render content when there is not an error for an attribute:

<!-- /resources/views/auth.blade.php -->

<label for="email">Email address</label>

<input id="email" type="email" class="@error('email') is-invalid @else is-valid @enderror">

You may pass [the name of a specific error bag](https://laravel.com/docs/8.x/validation#named-error-bags) as the second parameter to the @error directive to retrieve validation error messages on pages containing multiple forms:

<!-- /resources/views/auth.blade.php -->

<label for="email">Email address</label>

<input id="email" type="email" class="@error('email', 'login') is-invalid @enderror">

@error('email', 'login')

<div class="alert alert-danger">{{ $message }}</div>

@enderror

[Stacks](https://laravel.com/docs/8.x/blade#stacks)

Blade allows you to push to named stacks which can be rendered somewhere else in another view or layout. This can be particularly useful for specifying any JavaScript libraries required by your child views:

@push('scripts')

<script src="/example.js"></script>

@endpush

You may push to a stack as many times as needed. To render the complete stack contents, pass the name of the stack to the @stack directive:

<head>

<!-- Head Contents -->

@stack('scripts')

</head>

If you would like to prepend content onto the beginning of a stack, you should use the @prepend directive:

@push('scripts')

This will be second...

@endpush

// Later...

@prepend('scripts')

This will be first...

@endprepend

[Service Injection](https://laravel.com/docs/8.x/blade#service-injection)

The @inject directive may be used to retrieve a service from the Laravel [service container](https://laravel.com/docs/8.x/container). The first argument passed to @inject is the name of the variable the service will be placed into, while the second argument is the class or interface name of the service you wish to resolve:

@inject('metrics', 'App\Services\MetricsService')

<div>

Monthly Revenue: {{ $metrics->monthlyRevenue() }}.

</div>

[Extending Blade](https://laravel.com/docs/8.x/blade#extending-blade)

Blade allows you to define your own custom directives using the directive method. When the Blade compiler encounters the custom directive, it will call the provided callback with the expression that the directive contains.

The following example creates a @datetime($var) directive which formats a given $var, which should be an instance of DateTime:

**<?php**

namespace App\Providers;

use Illuminate\Support\Facades\Blade;

use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider

{

/\*\*

\* Register any application services.

\*

\* @return void

\*/

public function register()

{

//

}

/\*\*

\* Bootstrap any application services.

\*

\* @return void

\*/

public function boot()

{

Blade::directive('datetime', function ($expression) {

return "<?php echo ($expression)->format('m/d/Y H:i'); ?>";

});

}

}

As you can see, we will chain the format method onto whatever expression is passed into the directive. So, in this example, the final PHP generated by this directive will be:

**<?php** echo ($var)->format('m/d/Y H:i'); **?>**

After updating the logic of a Blade directive, you will need to delete all of the cached Blade views. The cached Blade views may be removed using the view:clear Artisan command.

[Custom Echo Handlers](https://laravel.com/docs/8.x/blade#custom-echo-handlers)

If you attempt to "echo" an object using Blade, the object's \_\_toString method will be invoked. The [\_\_toString](https://www.php.net/manual/en/language.oop5.magic.php#object.tostring) method is one of PHP's built-in "magic methods". However, sometimes you may not have control over the \_\_toString method of a given class, such as when the class that you are interacting with belongs to a third-party library.

In these cases, Blade allows you to register a custom echo handler for that particular type of object. To accomplish this, you should invoke Blade's stringable method. The stringable method accepts a closure. This closure should type-hint the type of object that it is responsible for rendering. Typically, the stringable method should be invoked within the boot method of your application's AppServiceProvider class:

use Illuminate\Support\Facades\Blade;

use Money\Money;

/\*\*

\* Bootstrap any application services.

\*

\* @return void

\*/

public function boot()

{

Blade::stringable(function (Money $money) {

return $money->formatTo('en\_GB');

});

}

Once your custom echo handler has been defined, you may simply echo the object in your Blade template:

Cost: {{ $money }}

[Custom If Statements](https://laravel.com/docs/8.x/blade#custom-if-statements)

Programming a custom directive is sometimes more complex than necessary when defining simple, custom conditional statements. For that reason, Blade provides a Blade::if method which allows you to quickly define custom conditional directives using closures. For example, let's define a custom conditional that checks the configured default "disk" for the application. We may do this in the boot method of our AppServiceProvider:

use Illuminate\Support\Facades\Blade;

/\*\*

\* Bootstrap any application services.

\*

\* @return void

\*/

public function boot()

{

Blade::if('disk', function ($value) {

return config('filesystems.default') === $value;

});

}

Once the custom conditional has been defined, you can use it within your templates:

@disk('local')

<!-- The application is using the local disk... -->

@elsedisk('s3')

<!-- The application is using the s3 disk... -->

@else

<!-- The application is using some other disk... -->

@enddisk

@unlessdisk('local')

<!-- The application is not using the local disk... -->

@enddisk

URL Generation

* [Introduction](https://laravel.com/docs/8.x/urls#introduction)
* [The Basics](https://laravel.com/docs/8.x/urls#the-basics)
  + [Generating URLs](https://laravel.com/docs/8.x/urls#generating-urls)
  + [Accessing The Current URL](https://laravel.com/docs/8.x/urls#accessing-the-current-url)
* [URLs For Named Routes](https://laravel.com/docs/8.x/urls#urls-for-named-routes)
  + [Signed URLs](https://laravel.com/docs/8.x/urls#signed-urls)
* [URLs For Controller Actions](https://laravel.com/docs/8.x/urls#urls-for-controller-actions)
* [Default Values](https://laravel.com/docs/8.x/urls#default-values)

[Introduction](https://laravel.com/docs/8.x/urls#introduction)

Laravel provides several helpers to assist you in generating URLs for your application. These helpers are primarily helpful when building links in your templates and API responses, or when generating redirect responses to another part of your application.

[The Basics](https://laravel.com/docs/8.x/urls#the-basics)

[Generating URLs](https://laravel.com/docs/8.x/urls#generating-urls)

The url helper may be used to generate arbitrary URLs for your application. The generated URL will automatically use the scheme (HTTP or HTTPS) and host from the current request being handled by the application:

$post = App\Models\Post::find(1);

echo url("/posts/{$post->id}");

// http://example.com/posts/1

[Accessing The Current URL](https://laravel.com/docs/8.x/urls#accessing-the-current-url)

If no path is provided to the url helper, an Illuminate\Routing\UrlGenerator instance is returned, allowing you to access information about the current URL:

// Get the current URL without the query string...

echo url()->current();

// Get the current URL including the query string...

echo url()->full();

// Get the full URL for the previous request...

echo url()->previous();

Each of these methods may also be accessed via the URL [facade](https://laravel.com/docs/8.x/facades):

use Illuminate\Support\Facades\URL;

echo URL::current();

[URLs For Named Routes](https://laravel.com/docs/8.x/urls#urls-for-named-routes)

The route helper may be used to generate URLs to [named routes](https://laravel.com/docs/8.x/routing#named-routes). Named routes allow you to generate URLs without being coupled to the actual URL defined on the route. Therefore, if the route's URL changes, no changes need to be made to your calls to the route function. For example, imagine your application contains a route defined like the following:

Route::get('/post/{post}', function (Post $post) {

//

})->name('post.show');

To generate a URL to this route, you may use the route helper like so:

echo route('post.show', ['post' => 1]);

// http://example.com/post/1

Of course, the route helper may also be used to generate URLs for routes with multiple parameters:

Route::get('/post/{post}/comment/{comment}', function (Post $post, Comment $comment) {

//

})->name('comment.show');

echo route('comment.show', ['post' => 1, 'comment' => 3]);

// http://example.com/post/1/comment/3

Any additional array elements that do not correspond to the route's definition parameters will be added to the URL's query string:

echo route('post.show', ['post' => 1, 'search' => 'rocket']);

// http://example.com/post/1?search=rocket

[Eloquent Models](https://laravel.com/docs/8.x/urls#eloquent-models)

You will often be generating URLs using the route key (typically the primary key) of [Eloquent models](https://laravel.com/docs/8.x/eloquent). For this reason, you may pass Eloquent models as parameter values. The route helper will automatically extract the model's route key:

echo route('post.show', ['post' => $post]);

[Signed URLs](https://laravel.com/docs/8.x/urls#signed-urls)

Laravel allows you to easily create "signed" URLs to named routes. These URLs have a "signature" hash appended to the query string which allows Laravel to verify that the URL has not been modified since it was created. Signed URLs are especially useful for routes that are publicly accessible yet need a layer of protection against URL manipulation.

For example, you might use signed URLs to implement a public "unsubscribe" link that is emailed to your customers. To create a signed URL to a named route, use the signedRoute method of the URL facade:

use Illuminate\Support\Facades\URL;

return URL::signedRoute('unsubscribe', ['user' => 1]);

If you would like to generate a temporary signed route URL that expires after a specified amount of time, you may use the temporarySignedRoute method. When Laravel validates a temporary signed route URL, it will ensure that the expiration timestamp that is encoded into the signed URL has not elapsed:

use Illuminate\Support\Facades\URL;

return URL::temporarySignedRoute(

'unsubscribe', now()->addMinutes(30), ['user' => 1]

);

[Validating Signed Route Requests](https://laravel.com/docs/8.x/urls#validating-signed-route-requests)

To verify that an incoming request has a valid signature, you should call the hasValidSignature method on the incoming Request:

use Illuminate\Http\Request;

Route::get('/unsubscribe/{user}', function (Request $request) {

if (! $request->hasValidSignature()) {

abort(401);

}

// ...

})->name('unsubscribe');

Alternatively, you may assign the Illuminate\Routing\Middleware\ValidateSignature [middleware](https://laravel.com/docs/8.x/middleware) to the route. If it is not already present, you should assign this middleware a key in your HTTP kernel's routeMiddleware array:

/\*\*

\* The application's route middleware.

\*

\* These middleware may be assigned to groups or used individually.

\*

\* @var array

\*/

protected $routeMiddleware = [

'signed' => \Illuminate\Routing\Middleware\ValidateSignature::class,

];

Once you have registered the middleware in your kernel, you may attach it to a route. If the incoming request does not have a valid signature, the middleware will automatically return a 403 HTTP response:

Route::post('/unsubscribe/{user}', function (Request $request) {

// ...

})->name('unsubscribe')->middleware('signed');

[Responding To Invalid Signed Routes](https://laravel.com/docs/8.x/urls#responding-to-invalid-signed-routes)

When someone visits a signed URL that has expired, they will receive a generic error page for the 403 HTTP status code. However, you can customize this behavior by defining a custom "renderable" closure for the InvalidSignatureException exception in your exception handler. This closure should return an HTTP response:

use Illuminate\Routing\Exceptions\InvalidSignatureException;

/\*\*

\* Register the exception handling callbacks for the application.

\*

\* @return void

\*/

public function register()

{

$this->renderable(function (InvalidSignatureException $e) {

return response()->view('error.link-expired', [], 403);

});

}

[URLs For Controller Actions](https://laravel.com/docs/8.x/urls#urls-for-controller-actions)

The action function generates a URL for the given controller action:

use App\Http\Controllers\HomeController;

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

If the controller method accepts route parameters, you may pass an associative array of route parameters as the second argument to the function:

$url = action([UserController::class, 'profile'], ['id' => 1]);

[Default Values](https://laravel.com/docs/8.x/urls#default-values)

For some applications, you may wish to specify request-wide default values for certain URL parameters. For example, imagine many of your routes define a {locale} parameter:

Route::get('/{locale}/posts', function () {

//

})->name('post.index');

It is cumbersome to always pass the locale every time you call the route helper. So, you may use the URL::defaults method to define a default value for this parameter that will always be applied during the current request. You may wish to call this method from a [route middleware](https://laravel.com/docs/8.x/middleware#assigning-middleware-to-routes) so that you have access to the current request:

**<?php**

namespace App\Http\Middleware;

use Closure;

use Illuminate\Support\Facades\URL;

class SetDefaultLocaleForUrls

{

/\*\*

\* Handle the incoming request.

\*

\* @param \Illuminate\Http\Request $request

\* @param \Closure $next

\* @return \Illuminate\Http\Response

\*/

public function handle($request, Closure $next)

{

URL::defaults(['locale' => $request->user()->locale]);

return $next($request);

}

}

Once the default value for the locale parameter has been set, you are no longer required to pass its value when generating URLs via the route helper.

[URL Defaults & Middleware Priority](https://laravel.com/docs/8.x/urls#url-defaults-middleware-priority)

Setting URL default values can interfere with Laravel's handling of implicit model bindings. Therefore, you should [prioritize your middleware](https://laravel.com/docs/8.x/middleware#sorting-middleware) that set URL defaults to be executed before Laravel's own SubstituteBindings middleware. You can accomplish this by making sure your middleware occurs before the SubstituteBindings middleware within the $middlewarePriority property of your application's HTTP kernel.

The $middlewarePriority property is defined in the base Illuminate\Foundation\Http\Kernel class. You may copy its definition from that class and overwrite it in your application's HTTP kernel in order to modify it:

/\*\*

\* The priority-sorted list of middleware.

\*

\* This forces non-global middleware to always be in the given order.

\*

\* @var array

\*/

protected $middlewarePriority = [

// ...

\App\Http\Middleware\SetDefaultLocaleForUrls::class,

\Illuminate\Routing\Middleware\SubstituteBindings::class,

// ...

];

HTTP Session

* [Introduction](https://laravel.com/docs/8.x/session#introduction)
  + [Configuration](https://laravel.com/docs/8.x/session#configuration)
  + [Driver Prerequisites](https://laravel.com/docs/8.x/session#driver-prerequisites)
* [Interacting With The Session](https://laravel.com/docs/8.x/session#interacting-with-the-session)
  + [Retrieving Data](https://laravel.com/docs/8.x/session#retrieving-data)
  + [Storing Data](https://laravel.com/docs/8.x/session#storing-data)
  + [Flash Data](https://laravel.com/docs/8.x/session#flash-data)
  + [Deleting Data](https://laravel.com/docs/8.x/session#deleting-data)
  + [Regenerating The Session ID](https://laravel.com/docs/8.x/session#regenerating-the-session-id)
* [Session Blocking](https://laravel.com/docs/8.x/session#session-blocking)
* [Adding Custom Session Drivers](https://laravel.com/docs/8.x/session#adding-custom-session-drivers)
  + [Implementing The Driver](https://laravel.com/docs/8.x/session#implementing-the-driver)
  + [Registering The Driver](https://laravel.com/docs/8.x/session#registering-the-driver)

[Introduction](https://laravel.com/docs/8.x/session#introduction)

Since HTTP driven applications are stateless, sessions provide a way to store information about the user across multiple requests. That user information is typically placed in a persistent store / backend that can be accessed from subsequent requests.

Laravel ships with a variety of session backends that are accessed through an expressive, unified API. Support for popular backends such as [Memcached](https://memcached.org/), [Redis](https://redis.io/), and databases is included.

[Configuration](https://laravel.com/docs/8.x/session#configuration)

Your application's session configuration file is stored at config/session.php. Be sure to review the options available to you in this file. By default, Laravel is configured to use the file session driver, which will work well for many applications. If your application will be load balanced across multiple web servers, you should choose a centralized store that all servers can access, such as Redis or a database.

The session driver configuration option defines where session data will be stored for each request. Laravel 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 relational database.
* memcached / redis - sessions are stored in one of these fast, cache based stores.
* dynamodb - sessions are stored in AWS DynamoDB.
* array - sessions are stored in a PHP array and will not be persisted.

The array driver is primarily used during [testing](https://laravel.com/docs/8.x/testing) and prevents the data stored in the session from being persisted.

[Driver Prerequisites](https://laravel.com/docs/8.x/session#driver-prerequisites)

[Database](https://laravel.com/docs/8.x/session#database)

When using the database session driver, you will need to create a table to contain the session records. An example Schema declaration for the table may be found below:

Schema::create('sessions', function ($table) {

$table->string('id')->primary();

$table->foreignId('user\_id')->nullable()->index();

$table->string('ip\_address', 45)->nullable();

$table->text('user\_agent')->nullable();

$table->text('payload');

$table->integer('last\_activity')->index();

});

You may use the session:table Artisan command to generate this migration. To learn more about database migrations, you may consult the complete [migration documentation](https://laravel.com/docs/8.x/migrations):

php artisan session:table

php artisan migrate

[Redis](https://laravel.com/docs/8.x/session#redis)

Before using Redis sessions with Laravel, you will need to either install the PhpRedis PHP extension via PECL or install the predis/predis package (~1.0) via Composer. For more information on configuring Redis, consult Laravel's [Redis documentation](https://laravel.com/docs/8.x/redis#configuration).

In the session configuration file, the connection option may be used to specify which Redis connection is used by the session.

[Interacting With The Session](https://laravel.com/docs/8.x/session#interacting-with-the-session)

[Retrieving Data](https://laravel.com/docs/8.x/session#retrieving-data)

There are two primary ways of working with session data in Laravel: the global session helper and via a Request instance. First, let's look at accessing the session via a Request instance, which can be type-hinted on a route closure or controller method. Remember, controller method dependencies are automatically injected via the Laravel [service container](https://laravel.com/docs/8.x/container):

**<?php**

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;

use Illuminate\Http\Request;

class UserController extends Controller

{

/\*\*

\* Show the profile for the given user.

\*

\* @param Request $request

\* @param int $id

\* @return Response

\*/

public function show(Request $request, $id)

{

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

//

}

}

When you retrieve an item 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 and the requested key does not exist, the closure will be executed and its result returned:

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

$value = $request->session()->get('key', function () {

return 'default';

});

[The Global Session Helper](https://laravel.com/docs/8.x/session#the-global-session-helper)

You may also use the global session PHP function to retrieve and store data in the session. When the session helper is called with a single, string argument, it will return the value of that session key. When the helper is called with an array of key / value pairs, those values will be stored in the session:

Route::get('/home', function () {

// Retrieve a piece of data from the session...

$value = session('key');

// Specifying a default value...

$value = session('key', 'default');

// Store a piece of data in the session...

session(['key' => 'value']);

});

There is little practical difference between using the session via an HTTP request instance versus using the global session helper. Both methods are [testable](https://laravel.com/docs/8.x/testing) via the assertSessionHas method which is available in all of your test cases.

[Retrieving All Session Data](https://laravel.com/docs/8.x/session#retrieving-all-session-data)

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

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

[Determining If An Item Exists In The Session](https://laravel.com/docs/8.x/session#determining-if-an-item-exists-in-the-session)

To determine if an item is present in the session, you may use the has method. The has method returns true if the item is present and is not null:

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

//

}

To determine if an item is present in the session, even if its value is null, you may use the exists method:

if ($request->session()->exists('users')) {

//

}

To determine if an item is not present in the session, you may use the missing method. The missing method returns true if the item is null or if the item is not present:

if ($request->session()->missing('users')) {

//

}

[Storing Data](https://laravel.com/docs/8.x/session#storing-data)

To store data in the session, you will typically use the request instance's put method or the global session helper:

// Via a request instance...

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

// Via the global "session" helper...

session(['key' => 'value']);

[Pushing To Array Session Values](https://laravel.com/docs/8.x/session#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 & Deleting An Item](https://laravel.com/docs/8.x/session#retrieving-deleting-an-item)

The pull method will retrieve and delete an item from the session in a single statement:

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

[Incrementing & Decrementing Session Values](https://laravel.com/docs/8.x/session##incrementing-and-decrementing-session-values)

If your session data contains an integer you wish to increment or decrement, you may use the increment and decrement methods:

$request->session()->increment('count');

$request->session()->increment('count', $incrementBy = 2);

$request->session()->decrement('count');

$request->session()->decrement('count', $decrementBy = 2);

[Flash Data](https://laravel.com/docs/8.x/session#flash-data)

Sometimes you may wish to store items in the session for the next request. You may do so using the flash method. Data stored in the session using this method will be available immediately and during the subsequent HTTP request. After the subsequent HTTP request, the flashed data will be deleted. Flash data is primarily useful for short-lived status messages:

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

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

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

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

To persist your flash data only for the current request, you may use the now method:

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

[Deleting Data](https://laravel.com/docs/8.x/session#deleting-data)

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:

// Forget a single key...

$request->session()->forget('name');

// Forget multiple keys...

$request->session()->forget(['name', 'status']);

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

[Regenerating The Session ID](https://laravel.com/docs/8.x/session#regenerating-the-session-id)

Regenerating the session ID is often done in order to prevent malicious users from exploiting a [session fixation](https://owasp.org/www-community/attacks/Session_fixation) attack on your application.

Laravel automatically regenerates the session ID during authentication if you are using one of the Laravel [application starter kits](https://laravel.com/docs/8.x/starter-kits) or [Laravel Fortify](https://laravel.com/docs/8.x/fortify); however, if you need to manually regenerate the session ID, you may use the regenerate method:

$request->session()->regenerate();

If you need to regenerate the session ID and remove all data from the session in a single statement, you may use the invalidate method:

$request->session()->invalidate();

[Session Blocking](https://laravel.com/docs/8.x/session#session-blocking)

To utilize session blocking, your application must be using a cache driver that supports [atomic locks](https://laravel.com/docs/8.x/cache#atomic-locks). Currently, those cache drivers include the memcached, dynamodb, redis, and database drivers. In addition, you may not use the cookie session driver.

By default, Laravel allows requests using the same session to execute concurrently. So, for example, if you use a JavaScript HTTP library to make two HTTP requests to your application, they will both execute at the same time. For many applications, this is not a problem; however, session data loss can occur in a small subset of applications that make concurrent requests to two different application endpoints which both write data to the session.

To mitigate this, Laravel provides functionality that allows you to limit concurrent requests for a given session. To get started, you may simply chain the block method onto your route definition. In this example, an incoming request to the /profile endpoint would acquire a session lock. While this lock is being held, any incoming requests to the /profile or /order endpoints which share the same session ID will wait for the first request to finish executing before continuing their execution:

Route::post('/profile', function () {

//

})->block($lockSeconds = 10, $waitSeconds = 10)

Route::post('/order', function () {

//

})->block($lockSeconds = 10, $waitSeconds = 10)

The block method accepts two optional arguments. The first argument accepted by the block method is the maximum number of seconds the session lock should be held for before it is released. Of course, if the request finishes executing before this time the lock will be released earlier.

The second argument accepted by the block method is the number of seconds a request should wait while attempting to obtain a session lock. An Illuminate\Contracts\Cache\LockTimeoutException will be thrown if the request is unable to obtain a session lock within the given number of seconds.

If neither of these arguments is passed, the lock will be obtained for a maximum of 10 seconds and requests will wait a maximum of 10 seconds while attempting to obtain a lock:

Route::post('/profile', function () {

//

})->block()

[Adding Custom Session Drivers](https://laravel.com/docs/8.x/session#adding-custom-session-drivers)

[Implementing The Driver](https://laravel.com/docs/8.x/session#implementing-the-driver)

If none of the existing session drivers fit your application's needs, Laravel makes it possible to write your own session handler. Your custom session driver should implement PHP's built-in SessionHandlerInterface. This interface contains just a few simple methods. A stubbed MongoDB implementation looks like the following:

**<?php**

namespace App\Extensions;

class MongoSessionHandler implements \SessionHandlerInterface

{

public function open($savePath, $sessionName) {}

public function close() {}

public function read($sessionId) {}

public function write($sessionId, $data) {}

public function destroy($sessionId) {}

public function gc($lifetime) {}

}

Laravel does not ship with a directory to contain your extensions. You are free to place them anywhere you like. In this example, we have created an Extensions directory to house the MongoSessionHandler.

Since the purpose of these methods is not readily understandable, let's quickly cover what each of the methods do:

* The open method would typically be used in file based session store systems. Since Laravel ships with a file session driver, you will rarely need to put anything in this method. You can simply leave this method empty.
* The close method, like the open method, can also usually be disregarded. For most drivers, it is not needed.
* The read method should return the string version of the session data associated with the given $sessionId. There is no need to do any serialization or other encoding when retrieving or storing session data in your driver, as Laravel will perform the serialization for you.
* The write method should write the given $data string associated with the $sessionId to some persistent storage system, such as MongoDB or another storage system of your choice. Again, you should not perform any serialization - Laravel will have already handled that for you.
* The destroy method should remove the data associated with the $sessionId from persistent storage.
* The gc method should destroy all session data that is older than the given $lifetime, which is a UNIX timestamp. For self-expiring systems like Memcached and Redis, this method may be left empty.

[Registering The Driver](https://laravel.com/docs/8.x/session#registering-the-driver)

Once your driver has been implemented, you are ready to register it with Laravel. To add additional drivers to Laravel's session backend, you may use the extend method provided by the Session [facade](https://laravel.com/docs/8.x/facades). You should call the extend method from the boot method of a [service provider](https://laravel.com/docs/8.x/providers). You may do this from the existing App\Providers\AppServiceProvider or create an entirely new provider:

**<?php**

namespace App\Providers;

use App\Extensions\MongoSessionHandler;

use Illuminate\Support\Facades\Session;

use Illuminate\Support\ServiceProvider;

class SessionServiceProvider extends ServiceProvider

{

/\*\*

\* Register any application services.

\*

\* @return void

\*/

public function register()

{

//

}

/\*\*

\* Bootstrap any application services.

\*

\* @return void

\*/

public function boot()

{

Session::extend('mongo', function ($app) {

// Return an implementation of SessionHandlerInterface...

return new MongoSessionHandler;

});

}

}

Once the session driver has been registered, you may use the mongo driver in your config/session.php configuration file.

Validation

* [Introduction](https://laravel.com/docs/8.x/validation#introduction)
* [Validation Quickstart](https://laravel.com/docs/8.x/validation#validation-quickstart)
  + [Defining The Routes](https://laravel.com/docs/8.x/validation#quick-defining-the-routes)
  + [Creating The Controller](https://laravel.com/docs/8.x/validation#quick-creating-the-controller)
  + [Writing The Validation Logic](https://laravel.com/docs/8.x/validation#quick-writing-the-validation-logic)
  + [Displaying The Validation Errors](https://laravel.com/docs/8.x/validation#quick-displaying-the-validation-errors)
  + [Repopulating Forms](https://laravel.com/docs/8.x/validation#repopulating-forms)
  + [A Note On Optional Fields](https://laravel.com/docs/8.x/validation#a-note-on-optional-fields)
* [Form Request Validation](https://laravel.com/docs/8.x/validation#form-request-validation)
  + [Creating Form Requests](https://laravel.com/docs/8.x/validation#creating-form-requests)
  + [Authorizing Form Requests](https://laravel.com/docs/8.x/validation#authorizing-form-requests)
  + [Customizing The Error Messages](https://laravel.com/docs/8.x/validation#customizing-the-error-messages)
  + [Preparing Input For Validation](https://laravel.com/docs/8.x/validation#preparing-input-for-validation)
* [Manually Creating Validators](https://laravel.com/docs/8.x/validation#manually-creating-validators)
  + [Automatic Redirection](https://laravel.com/docs/8.x/validation#automatic-redirection)
  + [Named Error Bags](https://laravel.com/docs/8.x/validation#named-error-bags)
  + [Customizing The Error Messages](https://laravel.com/docs/8.x/validation#manual-customizing-the-error-messages)
  + [After Validation Hook](https://laravel.com/docs/8.x/validation#after-validation-hook)
* [Working With Validated Input](https://laravel.com/docs/8.x/validation#working-with-validated-input)
* [Working With Error Messages](https://laravel.com/docs/8.x/validation#working-with-error-messages)
  + [Specifying Custom Messages In Language Files](https://laravel.com/docs/8.x/validation#specifying-custom-messages-in-language-files)
  + [Specifying Attributes In Language Files](https://laravel.com/docs/8.x/validation#specifying-attribute-in-language-files)
  + [Specifying Values In Language Files](https://laravel.com/docs/8.x/validation#specifying-values-in-language-files)
* [Available Validation Rules](https://laravel.com/docs/8.x/validation#available-validation-rules)
* [Conditionally Adding Rules](https://laravel.com/docs/8.x/validation#conditionally-adding-rules)
* [Validating Arrays](https://laravel.com/docs/8.x/validation#validating-arrays)
  + [Excluding Unvalidated Array Keys](https://laravel.com/docs/8.x/validation#excluding-unvalidated-array-keys)
  + [Validating Nested Array Input](https://laravel.com/docs/8.x/validation#validating-nested-array-input)
* [Validating Passwords](https://laravel.com/docs/8.x/validation#validating-passwords)
* [Custom Validation Rules](https://laravel.com/docs/8.x/validation#custom-validation-rules)
  + [Using Rule Objects](https://laravel.com/docs/8.x/validation#using-rule-objects)
  + [Using Closures](https://laravel.com/docs/8.x/validation#using-closures)
  + [Implicit Rules](https://laravel.com/docs/8.x/validation#implicit-rules)

[Introduction](https://laravel.com/docs/8.x/validation#introduction)

Laravel provides several different approaches to validate your application's incoming data. It is most common to use the validate method available on all incoming HTTP requests. However, we will discuss other approaches to validation as well.

Laravel includes a wide variety of convenient validation rules that you may apply to data, even providing the ability to validate if values are unique in a given database table. We'll cover each of these validation rules in detail so that you are familiar with all of Laravel's validation features.

[Validation Quickstart](https://laravel.com/docs/8.x/validation#validation-quickstart)

To learn about Laravel's powerful validation features, let's look at a complete example of validating a form and displaying the error messages back to the user. By reading this high-level overview, you'll be able to gain a good general understanding of how to validate incoming request data using Laravel:

[Defining The Routes](https://laravel.com/docs/8.x/validation#quick-defining-the-routes)

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

use App\Http\Controllers\PostController;

Route::get('/post/create', [PostController::class, 'create']);

Route::post('/post', [PostController::class, 'store']);

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](https://laravel.com/docs/8.x/validation#quick-creating-the-controller)

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

**<?php**

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;

use Illuminate\Http\Request;

class PostController extends Controller

{

/\*\*

\* Show the form to create a new blog post.

\*

\* @return \Illuminate\View\View

\*/

public function create()

{

return view('post.create');

}

/\*\*

\* Store a new blog post.

\*

\* @param \Illuminate\Http\Request $request

\* @return \Illuminate\Http\Response

\*/

public function store(Request $request)

{

// Validate and store the blog post...

}

}

[Writing The Validation Logic](https://laravel.com/docs/8.x/validation#quick-writing-the-validation-logic)

Now we are ready to fill in our store method with the logic to validate the new blog post. To do this, we will use the validate method provided by the Illuminate\Http\Request object. 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.

If validation fails during a traditional HTTP request, a redirect response to the previous URL will be generated. If the incoming request is an XHR request, a JSON response containing the validation error messages will be returned.

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

/\*\*

\* Store a new blog post.

\*

\* @param \Illuminate\Http\Request $request

\* @return \Illuminate\Http\Response

\*/

public function store(Request $request)

{

$validated = $request->validate([

'title' => 'required|unique:posts|max:255',

'body' => 'required',

]);

// The blog post is valid...

}

As you can see, the validation rules are passed into the validate method. Don't worry - all available validation rules are [documented](https://laravel.com/docs/8.x/validation#available-validation-rules). Again, if the validation fails, the proper response will automatically be generated. If the validation passes, our controller will continue executing normally.

Alternatively, validation rules may be specified as arrays of rules instead of a single | delimited string:

$validatedData = $request->validate([

'title' => ['required', 'unique:posts', 'max:255'],

'body' => ['required'],

]);

In addition, you may use the validateWithBag method to validate a request and store any error messages within a [named error bag](https://laravel.com/docs/8.x/validation#named-error-bags):

$validatedData = $request->validateWithBag('post', [

'title' => ['required', 'unique:posts', 'max:255'],

'body' => ['required'],

]);

[Stopping On First Validation Failure](https://laravel.com/docs/8.x/validation#stopping-on-first-validation-failure)

Sometimes you may wish to stop running validation rules on an attribute after the first validation failure. To do so, assign the bail rule to the attribute:

$request->validate([

'title' => 'bail|required|unique:posts|max:255',

'body' => 'required',

]);

In this example, if the unique rule on the title attribute fails, the max rule will not be checked. Rules will be validated in the order they are assigned.

[A Note On Nested Attributes](https://laravel.com/docs/8.x/validation#a-note-on-nested-attributes)

If the incoming HTTP request contains "nested" field data, you may specify these fields in your validation rules using "dot" syntax:

$request->validate([

'title' => 'required|unique:posts|max:255',

'author.name' => 'required',

'author.description' => 'required',

]);

On the other hand, if your field name contains a literal period, you can explicitly prevent this from being interpreted as "dot" syntax by escaping the period with a backslash:

$request->validate([

'title' => 'required|unique:posts|max:255',

'v1\.0' => 'required',

]);

[Displaying The Validation Errors](https://laravel.com/docs/8.x/validation#quick-displaying-the-validation-errors)

So, what if the incoming request fields do not pass the given validation rules? As mentioned previously, Laravel will automatically redirect the user back to their previous location. In addition, all of the validation errors and [request input](https://laravel.com/docs/8.x/requests#retrieving-old-input) will automatically be [flashed to the session](https://laravel.com/docs/8.x/session#flash-data).

An $errors variable is shared with all of your application's views by the Illuminate\View\Middleware\ShareErrorsFromSession middleware, which is provided by the web middleware group. When this middleware is applied an $errors variable will always be available in your views, 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](https://laravel.com/docs/8.x/validation#working-with-error-messages).

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 ($errors->any())

<div class="alert alert-danger">

<ul>

@foreach ($errors->all() as $error)

<li>{{ $error }}</li>

@endforeach

</ul>

</div>

@endif

<!-- Create Post Form -->

[Customizing The Error Messages](https://laravel.com/docs/8.x/validation#quick-customizing-the-error-messages)

Laravel's built-in validation rules each has an error message that is located in your application's resources/lang/en/validation.php file. Within this file, you will find a translation entry for each validation rule. You are free to change or modify these messages based on the needs of your application.

In addition, you may copy this file to another translation language directory to translate the messages for your application's language. To learn more about Laravel localization, check out the complete [localization documentation](https://laravel.com/docs/8.x/localization).

[XHR Requests & Validation](https://laravel.com/docs/8.x/validation#quick-xhr-requests-and-validation)

In this example, we used a traditional form to send data to the application. However, many applications receive XHR requests from a JavaScript powered frontend. When using the validate method during an XHR request, Laravel will not generate a redirect response. Instead, Laravel generates a JSON response containing all of the validation errors. This JSON response will be sent with a 422 HTTP status code.

[The @error Directive](https://laravel.com/docs/8.x/validation#the-at-error-directive)

You may use the @error [Blade](https://laravel.com/docs/8.x/blade) directive to quickly determine if validation error messages exist for a given attribute. Within an @error directive, you may echo the $message variable to display the error message:

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

<label for="title">Post Title</label>

<input id="title" type="text" name="title" class="@error('title') is-invalid @enderror">

@error('title')

<div class="alert alert-danger">{{ $message }}</div>

@enderror

If you are using [named error bags](https://laravel.com/docs/8.x/validation#named-error-bags), you may pass the name of the error bag as the second argument to the @error directive:

<input ... class="@error('title', 'post') is-invalid @enderror">

[Repopulating Forms](https://laravel.com/docs/8.x/validation#repopulating-forms)

When Laravel generates a redirect response due to a validation error, the framework will automatically [flash all of the request's input to the session](https://laravel.com/docs/8.x/session#flash-data). This is done so that you may conveniently access the input during the next request and repopulate the form that the user attempted to submit.

To retrieve flashed input from the previous request, invoke the old method on an instance of Illuminate\Http\Request. The old method will pull the previously flashed input data from the [session](https://laravel.com/docs/8.x/session):

$title = $request->old('title');

Laravel also provides a global old helper. If you are displaying old input within a [Blade template](https://laravel.com/docs/8.x/blade), it is more convenient to use the old helper to repopulate the form. If no old input exists for the given field, null will be returned:

<input type="text" name="title" value="{{ old('title') }}">

[A Note On Optional Fields](https://laravel.com/docs/8.x/validation#a-note-on-optional-fields)

By default, Laravel includes the TrimStrings and ConvertEmptyStringsToNull middleware in your application's global middleware stack. These middleware are listed in the stack by the App\Http\Kernel class. Because of this, you will often need to mark your "optional" request fields as nullable if you do not want the validator to consider null values as invalid. For example:

$request->validate([

'title' => 'required|unique:posts|max:255',

'body' => 'required',

'publish\_at' => 'nullable|date',

]);

In this example, we are specifying that the publish\_at field may be either null or a valid date representation. If the nullable modifier is not added to the rule definition, the validator would consider null an invalid date.

[Form Request Validation](https://laravel.com/docs/8.x/validation#form-request-validation)

[Creating Form Requests](https://laravel.com/docs/8.x/validation#creating-form-requests)

For more complex validation scenarios, you may wish to create a "form request". Form requests are custom request classes that encapsulate their own validation and authorization logic. To create a form request class, you may use the make:request Artisan CLI command:

php artisan make:request StorePostRequest

The generated form request class will be placed in the app/Http/Requests directory. If this directory does not exist, it will be created when you run the make:request command. Each form request generated by Laravel has two methods: authorize and rules.

As you might have guessed, the authorize method is responsible for determining if the currently authenticated user can perform the action represented by the request, while the rules method returns the validation rules that should apply to the request's data:

/\*\*

\* Get the validation rules that apply to the request.

\*

\* @return array

\*/

public function rules()

{

return [

'title' => 'required|unique:posts|max:255',

'body' => 'required',

];

}

You may type-hint any dependencies you require within the rules method's signature. They will automatically be resolved via the Laravel [service container](https://laravel.com/docs/8.x/container).

So, how are the validation rules evaluated? All you need to do is type-hint the request on your controller method. The incoming form request is validated before the controller method is called, meaning you do not need to clutter your controller with any validation logic:

/\*\*

\* Store a new blog post.

\*

\* @param \App\Http\Requests\StorePostRequest $request

\* @return Illuminate\Http\Response

\*/

public function store(StorePostRequest $request)

{

// The incoming request is valid...

// Retrieve the validated input data...

$validated = $request->validated();

// Retrieve a portion of the validated input data...

$validated = $request->safe()->only(['name', 'email']);

$validated = $request->safe()->except(['name', 'email']);

}

If validation fails, a redirect response will be generated to send the user back to their previous location. The errors will also be flashed to the session so they are available for display. If the request was an XHR request, an HTTP response with a 422 status code will be returned to the user including a JSON representation of the validation errors.

[Adding After Hooks To Form Requests](https://laravel.com/docs/8.x/validation#adding-after-hooks-to-form-requests)

If you would like to add an "after" validation hook to a form request, you may use the withValidator method. This method receives the fully constructed validator, allowing you to call any of its methods before the validation rules are actually evaluated:

/\*\*

\* Configure the validator instance.

\*

\* @param \Illuminate\Validation\Validator $validator

\* @return void

\*/

public function withValidator($validator)

{

$validator->after(function ($validator) {

if ($this->somethingElseIsInvalid()) {

$validator->errors()->add('field', 'Something is wrong with this field!');

}

});

}

[Stopping On First Validation Failure Attribute](https://laravel.com/docs/8.x/validation#request-stopping-on-first-validation-rule-failure)

By adding a stopOnFirstFailure property to your request class, you may inform the validator that it should stop validating all attributes once a single validation failure has occurred:

/\*\*

\* Indicates if the validator should stop on the first rule failure.

\*

\* @var bool

\*/

protected $stopOnFirstFailure = true;

[Customizing The Redirect Location](https://laravel.com/docs/8.x/validation#customizing-the-redirect-location)

As previously discussed, a redirect response will be generated to send the user back to their previous location when form request validation fails. However, you are free to customize this behavior. To do so, define a $redirect property on your form request:

/\*\*

\* The URI that users should be redirected to if validation fails.

\*

\* @var string

\*/

protected $redirect = '/dashboard';

Or, if you would like to redirect users to a named route, you may define a $redirectRoute property instead:

/\*\*

\* The route that users should be redirected to if validation fails.

\*

\* @var string

\*/

protected $redirectRoute = 'dashboard';

[Authorizing Form Requests](https://laravel.com/docs/8.x/validation#authorizing-form-requests)

The form request class also contains an authorize method. Within this method, you may determine if the authenticated user actually has the authority to update a given resource. For example, you may determine if a user actually owns a blog comment they are attempting to update. Most likely, you will interact with your [authorization gates and policies](https://laravel.com/docs/8.x/authorization) within this method:

use App\Models\Comment;

/\*\*

\* Determine if the user is authorized to make this request.

\*

\* @return bool

\*/

public function authorize()

{

$comment = Comment::find($this->route('comment'));

return $comment && $this->user()->can('update', $comment);

}

Since all form requests extend the base Laravel request class, we may use the user method to access the currently authenticated user. Also, note the call to the route method in the example above. This method grants you access to the URI parameters defined on the route being called, such as the {comment} parameter in the example below:

Route::post('/comment/{comment}');

Therefore, if your application is taking advantage of [route model binding](https://laravel.com/docs/8.x/routing#route-model-binding), your code may be made even more succinct by accessing the resolved model as a property of the request:

return $this->user()->can('update', $this->comment);

If the authorize method returns false, an HTTP response with a 403 status code will automatically be returned and your controller method will not execute.

If you plan to handle authorization logic for the request in another part of your application, you may simply return true from the authorize method:

/\*\*

\* Determine if the user is authorized to make this request.

\*

\* @return bool

\*/

public function authorize()

{

return true;

}

You may type-hint any dependencies you need within the authorize method's signature. They will automatically be resolved via the Laravel [service container](https://laravel.com/docs/8.x/container).

[Customizing The Error Messages](https://laravel.com/docs/8.x/validation#customizing-the-error-messages)

You may customize the error messages used by the form request by overriding the messages method. This method should return an array of attribute / rule pairs and their corresponding error messages:

/\*\*

\* Get the error messages for the defined validation rules.

\*

\* @return array

\*/

public function messages()

{

return [

'title.required' => 'A title is required',

'body.required' => 'A message is required',

];

}

[Customizing The Validation Attributes](https://laravel.com/docs/8.x/validation#customizing-the-validation-attributes)

Many of Laravel's built-in validation rule error messages contain an :attribute placeholder. If you would like the :attribute placeholder of your validation message to be replaced with a custom attribute name, you may specify the custom names by overriding the attributes method. This method should return an array of attribute / name pairs:

/\*\*

\* Get custom attributes for validator errors.

\*

\* @return array

\*/

public function attributes()

{

return [

'email' => 'email address',

];

}

[Preparing Input For Validation](https://laravel.com/docs/8.x/validation#preparing-input-for-validation)

If you need to prepare or sanitize any data from the request before you apply your validation rules, you may use the prepareForValidation method:

use Illuminate\Support\Str;

/\*\*

\* Prepare the data for validation.

\*

\* @return void

\*/

protected function prepareForValidation()

{

$this->merge([

'slug' => Str::slug($this->slug),

]);

}

[Manually Creating Validators](https://laravel.com/docs/8.x/validation#manually-creating-validators)

If you do not want to use the validate method on the request, you may create a validator instance manually using the Validator [facade](https://laravel.com/docs/8.x/facades). The make method on the facade generates a new validator instance:

**<?php**

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;

use Illuminate\Http\Request;

use Illuminate\Support\Facades\Validator;

class PostController extends Controller

{

/\*\*

\* Store a new blog post.

\*

\* @param Request $request

\* @return Response

\*/

public function store(Request $request)

{

$validator = Validator::make($request->all(), [

'title' => 'required|unique:posts|max:255',

'body' => 'required',

]);

if ($validator->fails()) {

return redirect('post/create')

->withErrors($validator)

->withInput();

}

// Retrieve the validated input...

$validated = $validator->validated();

// Retrieve a portion of the validated input...

$validated = $validator->safe()->only(['name', 'email']);

$validated = $validator->safe()->except(['name', 'email']);

// Store the blog post...

}

}

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

After determining whether the request validation failed, you may use the withErrors method to flash the error messages to the session. When using this method, the $errors variable will automatically be shared with your views after redirection, allowing you to easily display them back to the user. The withErrors method accepts a validator, a MessageBag, or a PHP array.

Stopping On First Validation Failure

The stopOnFirstFailure method will inform the validator that it should stop validating all attributes once a single validation failure has occurred:

if ($validator->stopOnFirstFailure()->fails()) {

// ...

}

[Automatic Redirection](https://laravel.com/docs/8.x/validation#automatic-redirection)

If you would like to create a validator instance manually but still take advantage of the automatic redirection offered by the HTTP request's validate method, you may call the validate method on an existing validator instance. If validation fails, the user will automatically be redirected or, in the case of an XHR request, a JSON response will be returned:

Validator::make($request->all(), [

'title' => 'required|unique:posts|max:255',

'body' => 'required',

])->validate();

You may use the validateWithBag method to store the error messages in a [named error bag](https://laravel.com/docs/8.x/validation#named-error-bags) if validation fails:

Validator::make($request->all(), [

'title' => 'required|unique:posts|max:255',

'body' => 'required',

])->validateWithBag('post');

[Named Error Bags](https://laravel.com/docs/8.x/validation#named-error-bags)

If you have multiple forms on a single page, you may wish to name the MessageBag containing the validation errors, allowing you to retrieve the error messages for a specific form. To achieve this, pass a name as the second argument to withErrors:

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

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

{{ $errors->login->first('email') }}

[Customizing The Error Messages](https://laravel.com/docs/8.x/validation#manual-customizing-the-error-messages)

If needed, you may provide custom error messages that a validator instance should use instead of the default error messages provided by Laravel. There are several ways to specify custom messages. First, you may pass the custom messages as the third argument to the Validator::make method:

$validator = Validator::make($input, $rules, $messages = [

'required' => 'The :attribute field is required.',

]);

In this example, the :attribute placeholder will be replaced by the actual name of the field under validation. You may also utilize other placeholders in validation messages. For example:

$messages = [

'same' => 'The :attribute and :other must match.',

'size' => 'The :attribute must be exactly :size.',

'between' => 'The :attribute value :input is not between :min - :max.',

'in' => 'The :attribute must be one of the following types: :values',

];

[Specifying A Custom Message For A Given Attribute](https://laravel.com/docs/8.x/validation#specifying-a-custom-message-for-a-given-attribute)

Sometimes you may wish to specify a custom error message only for a specific attribute. You may do so using "dot" notation. Specify the attribute's name first, followed by the rule:

$messages = [

'email.required' => 'We need to know your email address!',

];

[Specifying Custom Attribute Values](https://laravel.com/docs/8.x/validation#specifying-custom-attribute-values)

Many of Laravel's built-in error messages include an :attribute placeholder that is replaced with the name of the field or attribute under validation. To customize the values used to replace these placeholders for specific fields, you may pass an array of custom attributes as the fourth argument to the Validator::make method:

$validator = Validator::make($input, $rules, $messages, [

'email' => 'email address',

]);

[After Validation Hook](https://laravel.com/docs/8.x/validation#after-validation-hook)

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

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

$validator->after(function ($validator) {

if ($this->somethingElseIsInvalid()) {

$validator->errors()->add(

'field', 'Something is wrong with this field!'

);

}

});

if ($validator->fails()) {

//

}

[Working With Validated Input](https://laravel.com/docs/8.x/validation#working-with-validated-input)

After validating incoming request data using a form request or a manually created validator instance, you may wish to retrieve the incoming request data that actually underwent validation. This can be accomplished in several ways. First, you may call the validated method on a form request or validator instance. This method returns an array of the data that was validated:

$validated = $request->validated();

$validated = $validator->validated();

Alternatively, you may call the safe method on a form request or validator instance. This method returns an instance of Illuminate\Support\ValidatedInput. This object exposes only, except, and all methods to retrieve a subset of the validated data or the entire array of validated data:

$validated = $request->safe()->only(['name', 'email']);

$validated = $request->safe()->except(['name', 'email']);

$validated = $request->safe()->all();

In addition, the Illuminate\Support\ValidatedInput instance may be iterated over and accessed like an array:

// Validated data may be iterated...

foreach ($request->safe() as $key => $value) {

//

}

// Validated data may be accessed as an array...

$validated = $request->safe();

$email = $validated['email'];

If you would like to add additional fields to the validated data, you may call the merge method:

$validated = $request->safe()->merge(['name' => 'Taylor Otwell']);

If you would like to retrieve the validated data as a [collection](https://laravel.com/docs/8.x/collections) instance, you may call the collect method:

$collection = $request->safe()->collect();

[Working With Error Messages](https://laravel.com/docs/8.x/validation#working-with-error-messages)

After calling the errors method on a Validator instance, you will receive an Illuminate\Support\MessageBag instance, which has a variety of convenient methods for working with error messages. The $errors variable that is automatically made available to all views is also an instance of the MessageBag class.

[Retrieving The First Error Message For A Field](https://laravel.com/docs/8.x/validation#retrieving-the-first-error-message-for-a-field)

To retrieve the first error message for a given field, use the first method:

$errors = $validator->errors();

echo $errors->first('email');

[Retrieving All Error Messages For A Field](https://laravel.com/docs/8.x/validation#retrieving-all-error-messages-for-a-field)

If you need to retrieve an array of all the messages for a given field, use the get method:

foreach ($errors->get('email') as $message) {

//

}

If you are validating an array form field, you may retrieve all of the messages for each of the array elements using the \* character:

foreach ($errors->get('attachments.\*') as $message) {

//

}

[Retrieving All Error Messages For All Fields](https://laravel.com/docs/8.x/validation#retrieving-all-error-messages-for-all-fields)

To retrieve an array of all messages for all fields, use the all method:

foreach ($errors->all() as $message) {

//

}

[Determining If Messages Exist For A Field](https://laravel.com/docs/8.x/validation#determining-if-messages-exist-for-a-field)

The has method may be used to determine if any error messages exist for a given field:

if ($errors->has('email')) {

//

}

[Specifying Custom Messages In Language Files](https://laravel.com/docs/8.x/validation#specifying-custom-messages-in-language-files)

Laravel's built-in validation rules each has an error message that is located in your application's resources/lang/en/validation.php file. Within this file, you will find a translation entry for each validation rule. You are free to change or modify these messages based on the needs of your application.

In addition, you may copy this file to another translation language directory to translate the messages for your application's language. To learn more about Laravel localization, check out the complete [localization documentation](https://laravel.com/docs/8.x/localization).

[Custom Messages For Specific Attributes](https://laravel.com/docs/8.x/validation#custom-messages-for-specific-attributes)

You may customize the error messages used for specified attribute and rule combinations within your application's validation language files. To do so, add your message customizations to the custom array of your application's resources/lang/xx/validation.php language file:

'custom' => [

'email' => [

'required' => 'We need to know your email address!',

'max' => 'Your email address is too long!'

],

],

[Specifying Attributes In Language Files](https://laravel.com/docs/8.x/validation#specifying-attribute-in-language-files)

Many of Laravel's built-in error messages include an :attribute placeholder that is replaced with the name of the field or attribute under validation. If you would like the :attribute portion of your validation message to be replaced with a custom value, you may specify the custom attribute name in the attributes array of your resources/lang/xx/validation.php language file:

'attributes' => [

'email' => 'email address',

],

[Specifying Values In Language Files](https://laravel.com/docs/8.x/validation#specifying-values-in-language-files)

Some of Laravel's built-in validation rule error messages contain a :value placeholder that is replaced with the current value of the request attribute. However, you may occasionally need the :value portion of your validation message to be replaced with a custom representation of the value. For example, consider the following rule that specifies that a credit card number is required if the payment\_type has a value of cc:

Validator::make($request->all(), [

'credit\_card\_number' => 'required\_if:payment\_type,cc'

]);

If this validation rule fails, it will produce the following error message:

The credit card number field is required when payment type is cc.

Instead of displaying cc as the payment type value, you may specify a more user-friendly value representation in your resources/lang/xx/validation.php language file by defining a values array:

'values' => [

'payment\_type' => [

'cc' => 'credit card'

],

],

After defining this value, the validation rule will produce the following error message:

The credit card number field is required when payment type is credit card.

[Available Validation Rules](https://laravel.com/docs/8.x/validation#available-validation-rules)

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

[Accepted](https://laravel.com/docs/8.x/validation#rule-accepted)[Accepted If](https://laravel.com/docs/8.x/validation#rule-accepted-if)[Active URL](https://laravel.com/docs/8.x/validation#rule-active-url)[After (Date)](https://laravel.com/docs/8.x/validation#rule-after)[After Or Equal (Date)](https://laravel.com/docs/8.x/validation#rule-after-or-equal)[Alpha](https://laravel.com/docs/8.x/validation#rule-alpha)[Alpha Dash](https://laravel.com/docs/8.x/validation#rule-alpha-dash)[Alpha Numeric](https://laravel.com/docs/8.x/validation#rule-alpha-num)[Array](https://laravel.com/docs/8.x/validation#rule-array)[Bail](https://laravel.com/docs/8.x/validation#rule-bail)[Before (Date)](https://laravel.com/docs/8.x/validation#rule-before)[Before Or Equal (Date)](https://laravel.com/docs/8.x/validation#rule-before-or-equal)[Between](https://laravel.com/docs/8.x/validation#rule-between)[Boolean](https://laravel.com/docs/8.x/validation#rule-boolean)[Confirmed](https://laravel.com/docs/8.x/validation#rule-confirmed)[Current Password](https://laravel.com/docs/8.x/validation#rule-current-password)[Date](https://laravel.com/docs/8.x/validation#rule-date)[Date Equals](https://laravel.com/docs/8.x/validation#rule-date-equals)[Date Format](https://laravel.com/docs/8.x/validation#rule-date-format)[Different](https://laravel.com/docs/8.x/validation#rule-different)[Digits](https://laravel.com/docs/8.x/validation#rule-digits)[Digits Between](https://laravel.com/docs/8.x/validation#rule-digits-between)[Dimensions (Image Files)](https://laravel.com/docs/8.x/validation#rule-dimensions)[Distinct](https://laravel.com/docs/8.x/validation#rule-distinct)[Email](https://laravel.com/docs/8.x/validation#rule-email)[Ends With](https://laravel.com/docs/8.x/validation#rule-ends-with)[Exclude](https://laravel.com/docs/8.x/validation#rule-exclude)[Exclude If](https://laravel.com/docs/8.x/validation#rule-exclude-if)[Exclude Unless](https://laravel.com/docs/8.x/validation#rule-exclude-unless)[Exists (Database)](https://laravel.com/docs/8.x/validation#rule-exists)[File](https://laravel.com/docs/8.x/validation#rule-file)[Filled](https://laravel.com/docs/8.x/validation#rule-filled)[Greater Than](https://laravel.com/docs/8.x/validation#rule-gt)[Greater Than Or Equal](https://laravel.com/docs/8.x/validation#rule-gte)[Image (File)](https://laravel.com/docs/8.x/validation#rule-image)[In](https://laravel.com/docs/8.x/validation#rule-in)[In Array](https://laravel.com/docs/8.x/validation#rule-in-array)[Integer](https://laravel.com/docs/8.x/validation#rule-integer)[IP Address](https://laravel.com/docs/8.x/validation#rule-ip)[JSON](https://laravel.com/docs/8.x/validation#rule-json)[Less Than](https://laravel.com/docs/8.x/validation#rule-lt)[Less Than Or Equal](https://laravel.com/docs/8.x/validation#rule-lte)[Max](https://laravel.com/docs/8.x/validation#rule-max)[MIME Types](https://laravel.com/docs/8.x/validation#rule-mimetypes)[MIME Type By File Extension](https://laravel.com/docs/8.x/validation#rule-mimes)[Min](https://laravel.com/docs/8.x/validation#rule-min)[Multiple Of](https://laravel.com/docs/8.x/validation#multiple-of)[Not In](https://laravel.com/docs/8.x/validation#rule-not-in)[Not Regex](https://laravel.com/docs/8.x/validation#rule-not-regex)[Nullable](https://laravel.com/docs/8.x/validation#rule-nullable)[Numeric](https://laravel.com/docs/8.x/validation#rule-numeric)[Password](https://laravel.com/docs/8.x/validation#rule-password)[Present](https://laravel.com/docs/8.x/validation#rule-present)[Prohibited](https://laravel.com/docs/8.x/validation#rule-prohibited)[Prohibited If](https://laravel.com/docs/8.x/validation#rule-prohibited-if)[Prohibited Unless](https://laravel.com/docs/8.x/validation#rule-prohibited-unless)[Prohibits](https://laravel.com/docs/8.x/validation#rule-prohibits)[Regular Expression](https://laravel.com/docs/8.x/validation#rule-regex)[Required](https://laravel.com/docs/8.x/validation#rule-required)[Required If](https://laravel.com/docs/8.x/validation#rule-required-if)[Required Unless](https://laravel.com/docs/8.x/validation#rule-required-unless)[Required With](https://laravel.com/docs/8.x/validation#rule-required-with)[Required With All](https://laravel.com/docs/8.x/validation#rule-required-with-all)[Required Without](https://laravel.com/docs/8.x/validation#rule-required-without)[Required Without All](https://laravel.com/docs/8.x/validation#rule-required-without-all)[Same](https://laravel.com/docs/8.x/validation#rule-same)[Size](https://laravel.com/docs/8.x/validation#rule-size)[Sometimes](https://laravel.com/docs/8.x/validation#validating-when-present)[Starts With](https://laravel.com/docs/8.x/validation#rule-starts-with)[String](https://laravel.com/docs/8.x/validation#rule-string)[Timezone](https://laravel.com/docs/8.x/validation#rule-timezone)[Unique (Database)](https://laravel.com/docs/8.x/validation#rule-unique)[URL](https://laravel.com/docs/8.x/validation#rule-url)[UUID](https://laravel.com/docs/8.x/validation#rule-uuid)

[accepted](https://laravel.com/docs/8.x/validation#rule-accepted)

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

[accepted\_if:anotherfield,value,...](https://laravel.com/docs/8.x/validation#rule-accepted-if)

The field under validation must be "yes", "on", 1, or true if another field under validation is equal to a specified value. This is useful for validating "Terms of Service" acceptance or similar fields.

[active\_url](https://laravel.com/docs/8.x/validation#rule-active-url)

The field under validation must have a valid A or AAAA record according to the dns\_get\_record PHP function. The hostname of the provided URL is extracted using the parse\_url PHP function before being passed to dns\_get\_record.

[after:*date*](https://laravel.com/docs/8.x/validation#rule-after)

The field under validation must be a value after a given date. The dates will be passed into the strtotime PHP function in order to be converted to a valid DateTime instance:

'start\_date' => 'required|date|after:tomorrow'

Instead of passing a date string to be evaluated by strtotime, you may specify another field to compare against the date:

'finish\_date' => 'required|date|after:start\_date'

[after\_or\_equal:*date*](https://laravel.com/docs/8.x/validation#rule-after-or-equal)

The field under validation must be a value after or equal to the given date. For more information, see the [after](https://laravel.com/docs/8.x/validation#rule-after) rule.

[alpha](https://laravel.com/docs/8.x/validation#rule-alpha)

The field under validation must be entirely alphabetic characters.

[alpha\_dash](https://laravel.com/docs/8.x/validation#rule-alpha-dash)

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

[alpha\_num](https://laravel.com/docs/8.x/validation#rule-alpha-num)

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

[array](https://laravel.com/docs/8.x/validation#rule-array)

The field under validation must be a PHP array.

When additional values are provided to the array rule, each key in the input array must be present within the list of values provided to the rule. In the following example, the admin key in the input array is invalid since it is not contained in the list of values provided to the array rule:

use Illuminate\Support\Facades\Validator;

$input = [

'user' => [

'name' => 'Taylor Otwell',

'username' => 'taylorotwell',

'admin' => true,

],

];

Validator::make($input, [

'user' => 'array:username,locale',

]);

In general, you should always specify the array keys that are allowed to be present within your array. Otherwise, the validator's validate and validated methods will return all of the validated data, including the array and all of its keys, even if those keys were not validated by other nested array validation rules.

If you would like, you may instruct Laravel's validator to never include unvalidated array keys in the "validated" data it returns, even if you use the array rule without specifying a list of allowed keys. To accomplish this, you may call the validator's excludeUnvalidatedArrayKeys method in the boot method of your application's AppServiceProvider. After doing so, the validator will include array keys in the "validated" data it returns only when those keys were specifically validated by [nested array rules](https://laravel.com/docs/8.x/validation#validating-arrays):

use Illuminate\Support\Facades\Validator;

/\*\*

\* Register any application services.

\*

\* @return void

\*/

public function boot()

{

Validator::excludeUnvalidatedArrayKeys();

}

[bail](https://laravel.com/docs/8.x/validation#rule-bail)

Stop running validation rules for the field after the first validation failure.

While the bail rule will only stop validating a specific field when it encounters a validation failure, the stopOnFirstFailure method will inform the validator that it should stop validating all attributes once a single validation failure has occurred:

if ($validator->stopOnFirstFailure()->fails()) {

// ...

}

[before:*date*](https://laravel.com/docs/8.x/validation#rule-before)

The field under validation must be a value preceding the given date. The dates will be passed into the PHP strtotime function in order to be converted into a valid DateTime instance. In addition, like the [after](https://laravel.com/docs/8.x/validation#rule-after) rule, the name of another field under validation may be supplied as the value of date.

[before\_or\_equal:*date*](https://laravel.com/docs/8.x/validation#rule-before-or-equal)

The field under validation must be a value preceding or equal to the given date. The dates will be passed into the PHP strtotime function in order to be converted into a valid DateTime instance. In addition, like the [after](https://laravel.com/docs/8.x/validation#rule-after) rule, the name of another field under validation may be supplied as the value of date.

[between:*min*,*max*](https://laravel.com/docs/8.x/validation#rule-between)

The field under validation must have a size between the given *min* and *max*. Strings, numerics, arrays, and files are evaluated in the same fashion as the [size](https://laravel.com/docs/8.x/validation#rule-size) rule.

[boolean](https://laravel.com/docs/8.x/validation#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](https://laravel.com/docs/8.x/validation#rule-confirmed)

The field under validation must have a matching field of {field}\_confirmation. For example, if the field under validation is password, a matching password\_confirmation field must be present in the input.

[current\_password](https://laravel.com/docs/8.x/validation#rule-current-password)

The field under validation must match the authenticated user's password. You may specify an [authentication guard](https://laravel.com/docs/8.x/authentication) using the rule's first parameter:

'password' => 'current\_password:api'

[date](https://laravel.com/docs/8.x/validation#rule-date)

The field under validation must be a valid, non-relative date according to the strtotime PHP function.

[date\_equals:*date*](https://laravel.com/docs/8.x/validation#rule-date-equals)

The field under validation must be equal to the given date. The dates will be passed into the PHP strtotime function in order to be converted into a valid DateTime instance.

[date\_format:*format*](https://laravel.com/docs/8.x/validation#rule-date-format)

The field under validation must match the given *format*. You should use either date or date\_format when validating a field, not both. This validation rule supports all formats supported by PHP's [DateTime](https://www.php.net/manual/en/class.datetime.php) class.

[different:*field*](https://laravel.com/docs/8.x/validation#rule-different)

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

[digits:*value*](https://laravel.com/docs/8.x/validation#rule-digits)

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

[digits\_between:*min*,*max*](https://laravel.com/docs/8.x/validation#rule-digits-between)

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

[dimensions](https://laravel.com/docs/8.x/validation#rule-dimensions)

The file under validation must be an image meeting the dimension constraints as specified by the rule's parameters:

'avatar' => 'dimensions:min\_width=100,min\_height=200'

Available constraints are: *min\_width*, *max\_width*, *min\_height*, *max\_height*, *width*, *height*, *ratio*.

A *ratio* constraint should be represented as width divided by height. This can be specified either by a fraction like 3/2 or a float like 1.5:

'avatar' => 'dimensions:ratio=3/2'

Since this rule requires several arguments, you may use the Rule::dimensions method to fluently construct the rule:

use Illuminate\Support\Facades\Validator;

use Illuminate\Validation\Rule;

Validator::make($data, [

'avatar' => [

'required',

Rule::dimensions()->maxWidth(1000)->maxHeight(500)->ratio(3 / 2),

],

]);

[distinct](https://laravel.com/docs/8.x/validation#rule-distinct)

When validating arrays, the field under validation must not have any duplicate values:

'foo.\*.id' => 'distinct'

Distinct uses loose variable comparisons by default. To use strict comparisons, you may add the strict parameter to your validation rule definition:

'foo.\*.id' => 'distinct:strict'

You may add ignore\_case to the validation rule's arguments to make the rule ignore capitalization differences:

'foo.\*.id' => 'distinct:ignore\_case'

[email](https://laravel.com/docs/8.x/validation#rule-email)

The field under validation must be formatted as an email address. This validation rule utilizes the [egulias/email-validator](https://github.com/egulias/EmailValidator) package for validating the email address. By default, the RFCValidation validator is applied, but you can apply other validation styles as well:

'email' => 'email:rfc,dns'

The example above will apply the RFCValidation and DNSCheckValidation validations. Here's a full list of validation styles you can apply:

* rfc: RFCValidation
* strict: NoRFCWarningsValidation
* dns: DNSCheckValidation
* spoof: SpoofCheckValidation
* filter: FilterEmailValidation

The filter validator, which uses PHP's filter\_var function, ships with Laravel and was Laravel's default email validation behavior prior to Laravel version 5.8.

The dns and spoof validators require the PHP intl extension.

[ends\_with:*foo*,*bar*,...](https://laravel.com/docs/8.x/validation#rule-ends-with)

The field under validation must end with one of the given values.

[exclude](https://laravel.com/docs/8.x/validation#rule-exclude)

The field under validation will be excluded from the request data returned by the validate and validated methods.

[exclude\_if:*anotherfield*,*value*](https://laravel.com/docs/8.x/validation#rule-exclude-if)

The field under validation will be excluded from the request data returned by the validate and validated methods if the *anotherfield* field is equal to *value*.

[exclude\_unless:*anotherfield*,*value*](https://laravel.com/docs/8.x/validation#rule-exclude-unless)

The field under validation will be excluded from the request data returned by the validate and validated methods unless *anotherfield*'s field is equal to *value*. If *value* is null (exclude\_unless:name,null), the field under validation will be excluded unless the comparison field is null or the comparison field is missing from the request data.

[exists:*table*,*column*](https://laravel.com/docs/8.x/validation#rule-exists)

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

[Basic Usage Of Exists Rule](https://laravel.com/docs/8.x/validation#basic-usage-of-exists-rule)

'state' => 'exists:states'

If the column option is not specified, the field name will be used. So, in this case, the rule will validate that the states database table contains a record with a state column value matching the request's state attribute value.

[Specifying A Custom Column Name](https://laravel.com/docs/8.x/validation#specifying-a-custom-column-name)

You may explicitly specify the database column name that should be used by the validation rule by placing it after the database table name:

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

Occasionally, you may need to specify a specific database connection to be used for the exists query. You can accomplish this by prepending the connection name to the table name:

'email' => 'exists:connection.staff,email'

Instead of specifying the table name directly, you may specify the Eloquent model which should be used to determine the table name:

'user\_id' => 'exists:App\Models\User,id'

If you would like to customize the query executed by the validation rule, you may use the Rule class to fluently define the rule. In this example, we'll also specify the validation rules as an array instead of using the | character to delimit them:

use Illuminate\Support\Facades\Validator;

use Illuminate\Validation\Rule;

Validator::make($data, [

'email' => [

'required',

Rule::exists('staff')->where(function ($query) {

return $query->where('account\_id', 1);

}),

],

]);

[file](https://laravel.com/docs/8.x/validation#rule-file)

The field under validation must be a successfully uploaded file.

[filled](https://laravel.com/docs/8.x/validation#rule-filled)

The field under validation must not be empty when it is present.

[gt:*field*](https://laravel.com/docs/8.x/validation#rule-gt)

The field under validation must be greater than the given *field*. The two fields must be of the same type. Strings, numerics, arrays, and files are evaluated using the same conventions as the [size](https://laravel.com/docs/8.x/validation#rule-size) rule.

[gte:*field*](https://laravel.com/docs/8.x/validation#rule-gte)

The field under validation must be greater than or equal to the given *field*. The two fields must be of the same type. Strings, numerics, arrays, and files are evaluated using the same conventions as the [size](https://laravel.com/docs/8.x/validation#rule-size) rule.

[image](https://laravel.com/docs/8.x/validation#rule-image)

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

[in:*foo*,*bar*,...](https://laravel.com/docs/8.x/validation#rule-in)

The field under validation must be included in the given list of values. Since this rule often requires you to implode an array, the Rule::in method may be used to fluently construct the rule:

use Illuminate\Support\Facades\Validator;

use Illuminate\Validation\Rule;

Validator::make($data, [

'zones' => [

'required',

Rule::in(['first-zone', 'second-zone']),

],

]);

When the in rule is combined with the array rule, each value in the input array must be present within the list of values provided to the in rule. In the following example, the LAS airport code in the input array is invalid since it is not contained in the list of airports provided to the in rule:

use Illuminate\Support\Facades\Validator;

use Illuminate\Validation\Rule;

$input = [

'airports' => ['NYC', 'LAS'],

];

Validator::make($input, [

'airports' => [

'required',

'array',

Rule::in(['NYC', 'LIT']),

],

]);

[in\_array:*anotherfield*.\*](https://laravel.com/docs/8.x/validation#rule-in-array)

The field under validation must exist in *anotherfield*'s values.

[integer](https://laravel.com/docs/8.x/validation#rule-integer)

The field under validation must be an integer.

This validation rule does not verify that the input is of the "integer" variable type, only that the input is of a type accepted by PHP's FILTER\_VALIDATE\_INT rule. If you need to validate the input as being a number please use this rule in combination with [the numeric validation rule](https://laravel.com/docs/8.x/validation#rule-numeric).

[ip](https://laravel.com/docs/8.x/validation#rule-ip)

The field under validation must be an IP address.

[ipv4](https://laravel.com/docs/8.x/validation#ipv4)

The field under validation must be an IPv4 address.

[ipv6](https://laravel.com/docs/8.x/validation#ipv6)

The field under validation must be an IPv6 address.

[json](https://laravel.com/docs/8.x/validation#rule-json)

The field under validation must be a valid JSON string.

[lt:*field*](https://laravel.com/docs/8.x/validation#rule-lt)

The field under validation must be less than the given *field*. The two fields must be of the same type. Strings, numerics, arrays, and files are evaluated using the same conventions as the [size](https://laravel.com/docs/8.x/validation#rule-size) rule.

[lte:*field*](https://laravel.com/docs/8.x/validation#rule-lte)

The field under validation must be less than or equal to the given *field*. The two fields must be of the same type. Strings, numerics, arrays, and files are evaluated using the same conventions as the [size](https://laravel.com/docs/8.x/validation#rule-size) rule.

[max:*value*](https://laravel.com/docs/8.x/validation#rule-max)

The field under validation must be less than or equal to a maximum *value*. Strings, numerics, arrays, and files are evaluated in the same fashion as the [size](https://laravel.com/docs/8.x/validation#rule-size) rule.

[mimetypes:*text/plain*,...](https://laravel.com/docs/8.x/validation#rule-mimetypes)

The file under validation must match one of the given MIME types:

'video' => 'mimetypes:video/avi,video/mpeg,video/quicktime'

To determine the MIME type of the uploaded file, the file's contents will be read and the framework will attempt to guess the MIME type, which may be different from the client's provided MIME type.

[mimes:*foo*,*bar*,...](https://laravel.com/docs/8.x/validation#rule-mimes)

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

[Basic Usage Of MIME Rule](https://laravel.com/docs/8.x/validation#basic-usage-of-mime-rule)

'photo' => 'mimes:jpg,bmp,png'

Even though you only need to specify the extensions, this rule actually validates the MIME type of the file by reading the file's contents and guessing its MIME type. A full listing of MIME types and their corresponding extensions may be found at the following location:

<https://svn.apache.org/repos/asf/httpd/httpd/trunk/docs/conf/mime.types>

[min:*value*](https://laravel.com/docs/8.x/validation#rule-min)

The field under validation must have a minimum *value*. Strings, numerics, arrays, and files are evaluated in the same fashion as the [size](https://laravel.com/docs/8.x/validation#rule-size) rule.

[multiple\_of:*value*](https://laravel.com/docs/8.x/validation#multiple-of)

The field under validation must be a multiple of *value*.

[not\_in:*foo*,*bar*,...](https://laravel.com/docs/8.x/validation#rule-not-in)

The field under validation must not be included in the given list of values. The Rule::notIn method may be used to fluently construct the rule:

use Illuminate\Validation\Rule;

Validator::make($data, [

'toppings' => [

'required',

Rule::notIn(['sprinkles', 'cherries']),

],

]);

[not\_regex:*pattern*](https://laravel.com/docs/8.x/validation#rule-not-regex)

The field under validation must not match the given regular expression.

Internally, this rule uses the PHP preg\_match function. The pattern specified should obey the same formatting required by preg\_match and thus also include valid delimiters. For example: 'email' => 'not\_regex:/^.+$/i'.

When using the regex / not\_regex patterns, it may be necessary to specify your validation rules using an array instead of using | delimiters, especially if the regular expression contains a | character.

[nullable](https://laravel.com/docs/8.x/validation#rule-nullable)

The field under validation may be null.

[numeric](https://laravel.com/docs/8.x/validation#rule-numeric)

The field under validation must be [numeric](https://www.php.net/manual/en/function.is-numeric.php).

[password](https://laravel.com/docs/8.x/validation#rule-password)

The field under validation must match the authenticated user's password.

This rule was renamed to current\_password with the intention of removing it in Laravel 9. Please use the [Current Password](https://laravel.com/docs/8.x/validation#rule-current-password) rule instead.

[present](https://laravel.com/docs/8.x/validation#rule-present)

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

[prohibited](https://laravel.com/docs/8.x/validation#rule-prohibited)

The field under validation must be empty or not present.

[prohibited\_if:*anotherfield*,*value*,...](https://laravel.com/docs/8.x/validation#rule-prohibited-if)

The field under validation must be empty or not present if the *anotherfield* field is equal to any *value*.

[prohibited\_unless:*anotherfield*,*value*,...](https://laravel.com/docs/8.x/validation#rule-prohibited-unless)

The field under validation must be empty or not present unless the *anotherfield* field is equal to any *value*.

[prohibits:*anotherfield*,...](https://laravel.com/docs/8.x/validation#rule-prohibits)

If the field under validation is present, no fields in *anotherfield* can be present, even if empty.

[regex:*pattern*](https://laravel.com/docs/8.x/validation#rule-regex)

The field under validation must match the given regular expression.

Internally, this rule uses the PHP preg\_match function. The pattern specified should obey the same formatting required by preg\_match and thus also include valid delimiters. For example: 'email' => 'regex:/^.+@.+$/i'.

When using the regex / not\_regex patterns, it may be necessary to specify rules in an array instead of using | delimiters, especially if the regular expression contains a | character.

[required](https://laravel.com/docs/8.x/validation#rule-required)

The field under validation must be present in the input data and not empty. A field is considered "empty" if one of the following conditions are true:

* The value is null.
* The value is an empty string.
* The value is an empty array or empty Countable object.
* The value is an uploaded file with no path.

[required\_if:*anotherfield*,*value*,...](https://laravel.com/docs/8.x/validation#rule-required-if)

The field under validation must be present and not empty if the *anotherfield* field is equal to any *value*.

If you would like to construct a more complex condition for the required\_if rule, you may use the Rule::requiredIf method. This method accepts a boolean or a closure. When passed a closure, the closure should return true or false to indicate if the field under validation is required:

use Illuminate\Support\Facades\Validator;

use Illuminate\Validation\Rule;

Validator::make($request->all(), [

'role\_id' => Rule::requiredIf($request->user()->is\_admin),

]);

Validator::make($request->all(), [

'role\_id' => Rule::requiredIf(function () use ($request) {

return $request->user()->is\_admin;

}),

]);

[required\_unless:*anotherfield*,*value*,...](https://laravel.com/docs/8.x/validation#rule-required-unless)

The field under validation must be present and not empty unless the *anotherfield* field is equal to any *value*. This also means *anotherfield* must be present in the request data unless *value* is null. If *value* is null (required\_unless:name,null), the field under validation will be required unless the comparison field is null or the comparison field is missing from the request data.

[required\_with:*foo*,*bar*,...](https://laravel.com/docs/8.x/validation#rule-required-with)

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

[required\_with\_all:*foo*,*bar*,...](https://laravel.com/docs/8.x/validation#rule-required-with-all)

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

[required\_without:*foo*,*bar*,...](https://laravel.com/docs/8.x/validation#rule-required-without)

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

[required\_without\_all:*foo*,*bar*,...](https://laravel.com/docs/8.x/validation#rule-required-without-all)

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

[same:*field*](https://laravel.com/docs/8.x/validation#rule-same)

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

[size:*value*](https://laravel.com/docs/8.x/validation#rule-size)

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 (the attribute must also have the numeric or integer rule). For an array, *size* corresponds to the count of the array. For files, *size* corresponds to the file size in kilobytes. Let's look at some examples:

// Validate that a string is exactly 12 characters long...

'title' => 'size:12';

// Validate that a provided integer equals 10...

'seats' => 'integer|size:10';

// Validate that an array has exactly 5 elements...

'tags' => 'array|size:5';

// Validate that an uploaded file is exactly 512 kilobytes...

'image' => 'file|size:512';

[starts\_with:*foo*,*bar*,...](https://laravel.com/docs/8.x/validation#rule-starts-with)

The field under validation must start with one of the given values.

[string](https://laravel.com/docs/8.x/validation#rule-string)

The field under validation must be a string. If you would like to allow the field to also be null, you should assign the nullable rule to the field.

[timezone](https://laravel.com/docs/8.x/validation#rule-timezone)

The field under validation must be a valid timezone identifier according to the timezone\_identifiers\_list PHP function.

[unique:*table*,*column*,*except*,*idColumn*](https://laravel.com/docs/8.x/validation#rule-unique)

The field under validation must not exist within the given database table.

Specifying A Custom Table / Column Name:

Instead of specifying the table name directly, you may specify the Eloquent model which should be used to determine the table name:

'email' => 'unique:App\Models\User,email\_address'

The column option may be used to specify the field's corresponding database column. If the column option is not specified, the name of the field under validation will be used.

'email' => 'unique:users,email\_address'

Specifying A Custom Database Connection

Occasionally, you may need to set a custom connection for database queries made by the Validator. To accomplish this, you may prepend the connection name to the table name:

'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 unique validation. For example, consider an "update profile" screen that includes the user's name, email address, and location. You will probably want to verify that the email address is unique. However, if the user only changes the name field and not the email field, you do not want a validation error to be thrown because the user is already the owner of the email address in question.

To instruct the validator to ignore the user's ID, we'll use the Rule class to fluently define the rule. In this example, we'll also specify the validation rules as an array instead of using the | character to delimit the rules:

use Illuminate\Support\Facades\Validator;

use Illuminate\Validation\Rule;

Validator::make($data, [

'email' => [

'required',

Rule::unique('users')->ignore($user->id),

],

]);

You should never pass any user controlled request input into the ignore method. Instead, you should only pass a system generated unique ID such as an auto-incrementing ID or UUID from an Eloquent model instance. Otherwise, your application will be vulnerable to an SQL injection attack.

Instead of passing the model key's value to the ignore method, you may also pass the entire model instance. Laravel will automatically extract the key from the model:

Rule::unique('users')->ignore($user)

If your table uses a primary key column name other than id, you may specify the name of the column when calling the ignore method:

Rule::unique('users')->ignore($user->id, 'user\_id')

By default, the unique rule will check the uniqueness of the column matching the name of the attribute being validated. However, you may pass a different column name as the second argument to the unique method:

Rule::unique('users', 'email\_address')->ignore($user->id),

Adding Additional Where Clauses:

You may specify additional query conditions by customizing the query using the where method. For example, let's add a query condition that scopes the query to only search records that have an account\_id column value of 1:

'email' => Rule::unique('users')->where(function ($query) {

return $query->where('account\_id', 1);

})

[url](https://laravel.com/docs/8.x/validation#rule-url)

The field under validation must be a valid URL.

[uuid](https://laravel.com/docs/8.x/validation#rule-uuid)

The field under validation must be a valid RFC 4122 (version 1, 3, 4, or 5) universally unique identifier (UUID).

[Conditionally Adding Rules](https://laravel.com/docs/8.x/validation#conditionally-adding-rules)

[Skipping Validation When Fields Have Certain Values](https://laravel.com/docs/8.x/validation#skipping-validation-when-fields-have-certain-values)

You may occasionally wish to not validate a given field if another field has a given value. You may accomplish this using the exclude\_if validation rule. In this example, the appointment\_date and doctor\_name fields will not be validated if the has\_appointment field has a value of false:

use Illuminate\Support\Facades\Validator;

$validator = Validator::make($data, [

'has\_appointment' => 'required|boolean',

'appointment\_date' => 'exclude\_if:has\_appointment,false|required|date',

'doctor\_name' => 'exclude\_if:has\_appointment,false|required|string',

]);

Alternatively, you may use the exclude\_unless rule to not validate a given field unless another field has a given value:

$validator = Validator::make($data, [

'has\_appointment' => 'required|boolean',

'appointment\_date' => 'exclude\_unless:has\_appointment,true|required|date',

'doctor\_name' => 'exclude\_unless:has\_appointment,true|required|string',

]);

[Validating When Present](https://laravel.com/docs/8.x/validation#validating-when-present)

In some situations, you may wish to run validation checks against a field only if that field is present in the data being validated. 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.

If you are attempting to validate a field that should always be present but may be empty, check out [this note on optional fields](https://laravel.com/docs/8.x/validation#a-note-on-optional-fields).

[Complex Conditional Validation](https://laravel.com/docs/8.x/validation#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:

use Illuminate\Support\Facades\Validator;

$validator = Validator::make($request->all(), [

'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 resale shop, or maybe they just enjoy collecting games. To conditionally add this requirement, we can use the sometimes method on the Validator instance.

$validator->sometimes('reason', 'required|max:500', function ($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 a list of 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:

$validator->sometimes(['reason', 'cost'], 'required', function ($input) {

return $input->games >= 100;

});

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 under validation.

[Complex Conditional Array Validation](https://laravel.com/docs/8.x/validation#complex-conditional-array-validation)

Sometimes you may want to validate a field based on another field in the same nested array whose index you do not know. In these situations, you may allow your closure to receive a second argument which will be the current individual item in the array being validated:

$input = [

'channels' => [

[

'type' => 'email',

'address' => 'abigail@example.com',

],

[

'type' => 'url',

'address' => 'https://example.com',

],

],

];

$validator->sometimes('channels.\*.address', 'email', function($input, $item) {

return $item->type === 'email';

});

$validator->sometimes('channels.\*.address', 'url', function($input, $item) {

return $item->type !== 'email';

});

Like the $input parameter passed to the closure, the $item parameter is an instance of Illuminate\Support\Fluent when the attribute data is an array; otherwise, it is a string.

[Validating Arrays](https://laravel.com/docs/8.x/validation#validating-arrays)

As discussed in the [array validation rule documentation](https://laravel.com/docs/8.x/validation#rule-array), the array rule accepts a list of allowed array keys. If any additional keys are present within the array, validation will fail:

use Illuminate\Support\Facades\Validator;

$input = [

'user' => [

'name' => 'Taylor Otwell',

'username' => 'taylorotwell',

'admin' => true,

],

];

Validator::make($input, [

'user' => 'array:username,locale',

]);

In general, you should always specify the array keys that are allowed to be present within your array. Otherwise, the validator's validate and validated methods will return all of the validated data, including the array and all of its keys, even if those keys were not validated by other nested array validation rules.

[Excluding Unvalidated Array Keys](https://laravel.com/docs/8.x/validation#excluding-unvalidated-array-keys)

If you would like, you may instruct Laravel's validator to never include unvalidated array keys in the "validated" data it returns, even if you use the array rule without specifying a list of allowed keys. To accomplish this, you may call the validator's excludeUnvalidatedArrayKeys method in the boot method of your application's AppServiceProvider. After doing so, the validator will include array keys in the "validated" data it returns only when those keys were specifically validated by [nested array rules](https://laravel.com/docs/8.x/validation#validating-arrays):

use Illuminate\Support\Facades\Validator;

/\*\*

\* Register any application services.

\*

\* @return void

\*/

public function boot()

{

Validator::excludeUnvalidatedArrayKeys();

}

[Validating Nested Array Input](https://laravel.com/docs/8.x/validation#validating-nested-array-input)

Validating nested array based form input fields doesn't have to be a pain. You may use "dot notation" to validate attributes within an array. For example, if the incoming HTTP request contains a photos[profile] field, you may validate it like so:

use Illuminate\Support\Facades\Validator;

$validator = Validator::make($request->all(), [

'photos.profile' => 'required|image',

]);

You may also validate each element of an array. For example, to validate that each email in a given array input field is unique, you may do the following:

$validator = Validator::make($request->all(), [

'person.\*.email' => 'email|unique:users',

'person.\*.first\_name' => 'required\_with:person.\*.last\_name',

]);

Likewise, you may use the \* character when specifying [custom validation messages in your language files](https://laravel.com/docs/8.x/validation#custom-messages-for-specific-attributes), making it a breeze to use a single validation message for array based fields:

'custom' => [

'person.\*.email' => [

'unique' => 'Each person must have a unique email address',

]

],

[Validating Passwords](https://laravel.com/docs/8.x/validation#validating-passwords)

To ensure that passwords have an adequate level of complexity, you may use Laravel's Password rule object:

use Illuminate\Support\Facades\Validator;

use Illuminate\Validation\Rules\Password;

$validator = Validator::make($request->all(), [

'password' => ['required', 'confirmed', Password::min(8)],

]);

The Password rule object allows you to easily customize the password complexity requirements for your application, such as specifying that passwords require at least one letter, number, symbol, or characters with mixed casing:

// Require at least 8 characters...

Password::min(8)

// Require at least one letter...

Password::min(8)->letters()

// Require at least one uppercase and one lowercase letter...

Password::min(8)->mixedCase()

// Require at least one number...

Password::min(8)->numbers()

// Require at least one symbol...

Password::min(8)->symbols()

In addition, you may ensure that a password has not been compromised in a public password data breach leak using the uncompromised method:

Password::min(8)->uncompromised()

Internally, the Password rule object uses the [k-Anonymity](https://en.wikipedia.org/wiki/K-anonymity) model to determine if a password has been leaked via the [haveibeenpwned.com](https://haveibeenpwned.com/) service without sacrificing the user's privacy or security.

By default, if a password appears at least once in a data leak, it will be considered compromised. You can customize this threshold using the first argument of the uncompromised method:

// Ensure the password appears less than 3 times in the same data leak...

Password::min(8)->uncompromised(3);

Of course, you may chain all the methods in the examples above:

Password::min(8)

->letters()

->mixedCase()

->numbers()

->symbols()

->uncompromised()

[Defining Default Password Rules](https://laravel.com/docs/8.x/validation#defining-default-password-rules)

You may find it convenient to specify the default validation rules for passwords in a single location of your application. You can easily accomplish this using the Password::defaults method, which accepts a closure. The closure given to the defaults method should return the default configuration of the Password rule. Typically, the defaults rule should be called within the boot method of one of your application's service providers:

use Illuminate\Validation\Rules\Password;

/\*\*

\* Bootstrap any application services.

\*

\* @return void

\*/

public function boot()

{

Password::defaults(function () {

$rule = Password::min(8);

return $this->app->isProduction()

? $rule->mixedCase()->uncompromised()

: $rule;

});

}

Then, when you would like to apply the default rules to a particular password undergoing validation, you may invoke the defaults method with no arguments:

'password' => ['required', Password::defaults()],

[Custom Validation Rules](https://laravel.com/docs/8.x/validation#custom-validation-rules)

[Using Rule Objects](https://laravel.com/docs/8.x/validation#using-rule-objects)

Laravel 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 rule objects. To generate a new rule object, you may use the make:rule Artisan command. Let's use this command to generate a rule that verifies a string is uppercase. Laravel will place the new rule in the app/Rules directory. If this directory does not exist, Laravel will create it when you execute the Artisan command to create your rule:

php artisan make:rule Uppercase

Once the rule has been created, we are ready to define its behavior. A rule object contains two methods: passes and message. The passes method receives the attribute value and name, and should return true or false depending on whether the attribute value is valid or not. The message method should return the validation error message that should be used when validation fails:

**<?php**

namespace App\Rules;

use Illuminate\Contracts\Validation\Rule;

class Uppercase implements Rule

{

/\*\*

\* Determine if the validation rule passes.

\*

\* @param string $attribute

\* @param mixed $value

\* @return bool

\*/

public function passes($attribute, $value)

{

return strtoupper($value) === $value;

}

/\*\*

\* Get the validation error message.

\*

\* @return string

\*/

public function message()

{

return 'The :attribute must be uppercase.';

}

}

You may call the trans helper from your message method if you would like to return an error message from your translation files:

/\*\*

\* Get the validation error message.

\*

\* @return string

\*/

public function message()

{

return trans('validation.uppercase');

}

Once the rule has been defined, you may attach it to a validator by passing an instance of the rule object with your other validation rules:

use App\Rules\Uppercase;

$request->validate([

'name' => ['required', 'string', new Uppercase],

]);

[Using Closures](https://laravel.com/docs/8.x/validation#using-closures)

If you only need the functionality of a custom rule once throughout your application, you may use a closure instead of a rule object. The closure receives the attribute's name, the attribute's value, and a $fail callback that should be called if validation fails:

use Illuminate\Support\Facades\Validator;

$validator = Validator::make($request->all(), [

'title' => [

'required',

'max:255',

function ($attribute, $value, $fail) {

if ($value === 'foo') {

$fail('The '.$attribute.' is invalid.');

}

},

],

]);

[Implicit Rules](https://laravel.com/docs/8.x/validation#implicit-rules)

By default, when an attribute being validated is not present or contains an empty string, normal validation rules, including custom rules, are not run. For example, the [unique](https://laravel.com/docs/8.x/validation#rule-unique) rule will not be run against an empty string:

use Illuminate\Support\Facades\Validator;

$rules = ['name' => 'unique:users,name'];

$input = ['name' => ''];

Validator::make($input, $rules)->passes(); // true

For a custom rule to run even when an attribute is empty, the rule must imply that the attribute is required. To create an "implicit" rule, implement the Illuminate\Contracts\Validation\ImplicitRule interface. This interface serves as a "marker interface" for the validator; therefore, it does not contain any additional methods you need to implement beyond the methods required by the typical Rule interface.

To generate a new implicit rule object, you may use the make:rule Artisan command with the --implicit option :

php artisan make:rule Uppercase --implicit

Error Handling

* [Introduction](https://laravel.com/docs/8.x/errors#introduction)
* [Configuration](https://laravel.com/docs/8.x/errors#configuration)
* [The Exception Handler](https://laravel.com/docs/8.x/errors#the-exception-handler)
  + [Reporting Exceptions](https://laravel.com/docs/8.x/errors#reporting-exceptions)
  + [Ignoring Exceptions By Type](https://laravel.com/docs/8.x/errors#ignoring-exceptions-by-type)
  + [Rendering Exceptions](https://laravel.com/docs/8.x/errors#rendering-exceptions)
  + [Reportable & Renderable Exceptions](https://laravel.com/docs/8.x/errors#renderable-exceptions)
* [HTTP Exceptions](https://laravel.com/docs/8.x/errors#http-exceptions)
  + [Custom HTTP Error Pages](https://laravel.com/docs/8.x/errors#custom-http-error-pages)

[Introduction](https://laravel.com/docs/8.x/errors#introduction)

When you start a new Laravel project, error and exception handling is already configured for you. The App\Exceptions\Handler class is where all exceptions thrown by your application are logged and then rendered to the user. We'll dive deeper into this class throughout this documentation.

[Configuration](https://laravel.com/docs/8.x/errors#configuration)

The debug option in your config/app.php configuration file determines how much information about an error is actually displayed to the user. By default, this option is set to respect the value of the APP\_DEBUG environment variable, which is stored in your .env file.

During local development, you should set the APP\_DEBUG environment variable to true. In your production environment, this value should always be false. If the value is set to true in production, you risk exposing sensitive configuration values to your application's end users.

[The Exception Handler](https://laravel.com/docs/8.x/errors#the-exception-handler)

[Reporting Exceptions](https://laravel.com/docs/8.x/errors#reporting-exceptions)

All exceptions are handled by the App\Exceptions\Handler class. This class contains a register method where you may register custom exception reporting and rendering callbacks. We'll examine each of these concepts in detail. Exception reporting is used to log exceptions or send them to an external service like [Flare](https://flareapp.io/), [Bugsnag](https://bugsnag.com/) or [Sentry](https://github.com/getsentry/sentry-laravel). By default, exceptions will be logged based on your [logging](https://laravel.com/docs/8.x/logging) configuration. 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 reportable method to register a closure that should be executed when an exception of a given type needs to be reported. Laravel will deduce what type of exception the closure reports by examining the type-hint of the closure:

use App\Exceptions\InvalidOrderException;

/\*\*

\* Register the exception handling callbacks for the application.

\*

\* @return void

\*/

public function register()

{

$this->reportable(function (InvalidOrderException $e) {

//

});

}

When you register a custom exception reporting callback using the reportable method, Laravel will still log the exception using the default logging configuration for the application. If you wish to stop the propagation of the exception to the default logging stack, you may use the stop method when defining your reporting callback or return false from the callback:

$this->reportable(function (InvalidOrderException $e) {

//

})->stop();

$this->reportable(function (InvalidOrderException $e) {

return false;

});

To customize the exception reporting for a given exception, you may also utilize [reportable exceptions](https://laravel.com/docs/8.x/errors#renderable-exceptions).

[Global Log Context](https://laravel.com/docs/8.x/errors#global-log-context)

If available, Laravel automatically adds the current user's ID to every exception's log message as contextual data. You may define your own global contextual data by overriding the context method of your application's App\Exceptions\Handler class. This information will be included in every exception's log message written by your application:

/\*\*

\* Get the default context variables for logging.

\*

\* @return array

\*/

protected function context()

{

return array\_merge(parent::context(), [

'foo' => 'bar',

]);

}

[Exception Log Context](https://laravel.com/docs/8.x/errors#exception-log-context)

While adding context to every log message can be useful, sometimes a particular exception may have unique context that you would like to include in your logs. By defining a context method on one of your application's custom exceptions, you may specify any data relevant to that exception that should be added to the exception's log entry:

**<?php**

namespace App\Exceptions;

use Exception;

class InvalidOrderException extends Exception

{

// ...

/\*\*

\* Get the exception's context information.

\*

\* @return array

\*/

public function context()

{

return ['order\_id' => $this->orderId];

}

}

[The report Helper](https://laravel.com/docs/8.x/errors#the-report-helper)

Sometimes you may need to report an exception but continue handling the current request. The report helper function allows you to quickly report an exception via the exception handler without rendering an error page to the user:

public function isValid($value)

{

try {

// Validate the value...

} catch (Throwable $e) {

report($e);

return false;

}

}

[Ignoring Exceptions By Type](https://laravel.com/docs/8.x/errors#ignoring-exceptions-by-type)

When building your application, there will be some types of exceptions you simply want to ignore and never report. Your application's exception handler contains a $dontReport property which is initialized to an empty array. Any classes that you add to this property will never be reported; however, they may still have custom rendering logic:

use App\Exceptions\InvalidOrderException;

/\*\*

\* A list of the exception types that should not be reported.

\*

\* @var array

\*/

protected $dontReport = [

InvalidOrderException::class,

];

Behind the scenes, Laravel already ignores some types of errors for you, such as exceptions resulting from 404 HTTP "not found" errors or 419 HTTP responses generated by invalid CSRF tokens.

[Rendering Exceptions](https://laravel.com/docs/8.x/errors#rendering-exceptions)

By default, the Laravel exception handler will convert exceptions into an HTTP response for you. However, you are free to register a custom rendering closure for exceptions of a given type. You may accomplish this via the renderable method of your exception handler.

The closure passed to the renderable method should return an instance of Illuminate\Http\Response, which may be generated via the response helper. Laravel will deduce what type of exception the closure renders by examining the type-hint of the closure:

use App\Exceptions\InvalidOrderException;

/\*\*

\* Register the exception handling callbacks for the application.

\*

\* @return void

\*/

public function register()

{

$this->renderable(function (InvalidOrderException $e, $request) {

return response()->view('errors.invalid-order', [], 500);

});

}

You may also use the renderable method to override the rendering behavior for built-in Laravel or Symfony exceptions such as NotFoundHttpException. If the closure given to the renderable method does not return a value, Laravel's default exception rendering will be utilized:

use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;

/\*\*

\* Register the exception handling callbacks for the application.

\*

\* @return void

\*/

public function register()

{

$this->renderable(function (NotFoundHttpException $e, $request) {

if ($request->is('api/\*')) {

return response()->json([

'message' => 'Record not found.'

], 404);

}

});

}

[Reportable & Renderable Exceptions](https://laravel.com/docs/8.x/errors#renderable-exceptions)

Instead of type-checking exceptions in the exception handler's register method, you may define report and render methods directly on your custom exceptions. When these methods exist, they will be automatically called by the framework:

**<?php**

namespace App\Exceptions;

use Exception;

class InvalidOrderException extends Exception

{

/\*\*

\* Report the exception.

\*

\* @return bool|null

\*/

public function report()

{

//

}

/\*\*

\* Render the exception into an HTTP response.

\*

\* @param \Illuminate\Http\Request $request

\* @return \Illuminate\Http\Response

\*/

public function render($request)

{

return response(...);

}

}

If your exception extends an exception that is already renderable, such as a built-in Laravel or Symfony exception, you may return false from the exception's render method to render the exception's default HTTP response:

/\*\*

\* Render the exception into an HTTP response.

\*

\* @param \Illuminate\Http\Request $request

\* @return \Illuminate\Http\Response

\*/

public function render($request)

{

// Determine if the exception needs custom rendering...

return false;

}

If your exception contains custom reporting logic that is only necessary when certain conditions are met, you may need to instruct Laravel to sometimes report the exception using the default exception handling configuration. To accomplish this, you may return false from the exception's report method:

/\*\*

\* Report the exception.

\*

\* @return bool|null

\*/

public function report()

{

// Determine if the exception needs custom reporting...

return false;

}

You may type-hint any required dependencies of the report method and they will automatically be injected into the method by Laravel's [service container](https://laravel.com/docs/8.x/container).

[HTTP Exceptions](https://laravel.com/docs/8.x/errors#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, you may use the abort helper:

abort(404);

[Custom HTTP Error Pages](https://laravel.com/docs/8.x/errors#custom-http-error-pages)

Laravel makes it easy to display custom error pages for various HTTP status codes. For example, if you wish to customize the error page for 404 HTTP status codes, create a resources/views/errors/404.blade.php view template. This view will be rendered on all 404 errors generated by your application. The views within this directory should be named to match the HTTP status code they correspond to. The Symfony\Component\HttpKernel\Exception\HttpException instance raised by the abort function will be passed to the view as an $exception variable:

<h2>{{ $exception->getMessage() }}</h2>

You may publish Laravel's default error page templates using the vendor:publish Artisan command. Once the templates have been published, you may customize them to your liking:

php artisan vendor:publish --tag=laravel-errors

Logging

* [Introduction](https://laravel.com/docs/8.x/logging#introduction)
* [Configuration](https://laravel.com/docs/8.x/logging#configuration)
  + [Available Channel Drivers](https://laravel.com/docs/8.x/logging#available-channel-drivers)
  + [Channel Prerequisites](https://laravel.com/docs/8.x/logging#channel-prerequisites)
* [Building Log Stacks](https://laravel.com/docs/8.x/logging#building-log-stacks)
* [Writing Log Messages](https://laravel.com/docs/8.x/logging#writing-log-messages)
  + [Contextual Information](https://laravel.com/docs/8.x/logging#contextual-information)
  + [Writing To Specific Channels](https://laravel.com/docs/8.x/logging#writing-to-specific-channels)
* [Monolog Channel Customization](https://laravel.com/docs/8.x/logging#monolog-channel-customization)
  + [Customizing Monolog For Channels](https://laravel.com/docs/8.x/logging#customizing-monolog-for-channels)
  + [Creating Monolog Handler Channels](https://laravel.com/docs/8.x/logging#creating-monolog-handler-channels)
  + [Creating Custom Channels Via Factories](https://laravel.com/docs/8.x/logging#creating-custom-channels-via-factories)

[Introduction](https://laravel.com/docs/8.x/logging#introduction)

To help you learn more about what's happening within your application, Laravel provides robust logging services that allow you to log messages to files, the system error log, and even to Slack to notify your entire team.

Laravel logging is based on "channels". Each channel represents a specific way of writing log information. For example, the single channel writes log files to a single log file, while the slack channel sends log messages to Slack. Log messages may be written to multiple channels based on their severity.

Under the hood, Laravel utilizes the [Monolog](https://github.com/Seldaek/monolog) library, which provides support for a variety of powerful log handlers. Laravel makes it a cinch to configure these handlers, allowing you to mix and match them to customize your application's log handling.

[Configuration](https://laravel.com/docs/8.x/logging#configuration)

All of the configuration options for your application's logging behavior is housed in the config/logging.php configuration file. This file allows you to configure your application's log channels, so be sure to review each of the available channels and their options. We'll review a few common options below.

By default, Laravel will use the stack channel when logging messages. The stack channel is used to aggregate multiple log channels into a single channel. For more information on building stacks, check out the [documentation below](https://laravel.com/docs/8.x/logging#building-log-stacks).

[Configuring The Channel Name](https://laravel.com/docs/8.x/logging#configuring-the-channel-name)

By default, Monolog is instantiated with a "channel name" that matches the current environment, such as production or local. To change this value, add a name option to your channel's configuration:

'stack' => [

'driver' => 'stack',

'name' => 'channel-name',

'channels' => ['single', 'slack'],

],

[Available Channel Drivers](https://laravel.com/docs/8.x/logging#available-channel-drivers)

Each log channel is powered by a "driver". The driver determines how and where the log message is actually recorded. The following log channel drivers are available in every Laravel application. An entry for most of these drivers is already present in your application's config/logging.php configuration file, so be sure to review this file to become familiar with its contents:

| **Name** | **Description** |
| --- | --- |
| custom | A driver that calls a specified factory to create a channel |
| daily | A RotatingFileHandler based Monolog driver which rotates daily |
| errorlog | A ErrorLogHandler based Monolog driver |
| monolog | A Monolog factory driver that may use any supported Monolog handler |
| null | A driver that discards all log messages |
| papertrail | A SyslogUdpHandler based Monolog driver |
| single | A single file or path based logger channel (StreamHandler) |
| slack | A SlackWebhookHandler based Monolog driver |
| stack | A wrapper to facilitate creating "multi-channel" channels |
| syslog | A SyslogHandler based Monolog driver |

Check out the documentation on [advanced channel customization](https://laravel.com/docs/8.x/logging#monolog-channel-customization) to learn more about the monolog and custom drivers.

[Channel Prerequisites](https://laravel.com/docs/8.x/logging#channel-prerequisites)

[Configuring The Single and Daily Channels](https://laravel.com/docs/8.x/logging#configuring-the-single-and-daily-channels)

The single and daily channels have three optional configuration options: bubble, permission, and locking.

| **Name** | **Description** | **Default** |
| --- | --- | --- |
| bubble | Indicates if messages should bubble up to other channels after being handled | true |
| locking | Attempt to lock the log file before writing to it | false |
| permission | The log file's permissions | 0644 |

[Configuring The Papertrail Channel](https://laravel.com/docs/8.x/logging#configuring-the-papertrail-channel)

The papertrail channel requires the host and port configuration options. You can obtain these values from [Papertrail](https://help.papertrailapp.com/kb/configuration/configuring-centralized-logging-from-php-apps/#send-events-from-php-app).

[Configuring The Slack Channel](https://laravel.com/docs/8.x/logging#configuring-the-slack-channel)

The slack channel requires a url configuration option. This URL should match a URL for an [incoming webhook](https://slack.com/apps/A0F7XDUAZ-incoming-webhooks) that you have configured for your Slack team.

By default, Slack will only receive logs at the critical level and above; however, you can adjust this in your config/logging.php configuration file by modifying the level configuration option within your Slack log channel's configuration array.

[Building Log Stacks](https://laravel.com/docs/8.x/logging#building-log-stacks)

As mentioned previously, the stack driver allows you to combine multiple channels into a single log channel for convenience. To illustrate how to use log stacks, let's take a look at an example configuration that you might see in a production application:

'channels' => [

'stack' => [

'driver' => 'stack',

'channels' => ['syslog', 'slack'],

],

'syslog' => [

'driver' => 'syslog',

'level' => 'debug',

],

'slack' => [

'driver' => 'slack',

'url' => env('LOG\_SLACK\_WEBHOOK\_URL'),

'username' => 'Laravel Log',

'emoji' => ':boom:',

'level' => 'critical',

],

],

Let's dissect this configuration. First, notice our stack channel aggregates two other channels via its channels option: syslog and slack. So, when logging messages, both of these channels will have the opportunity to log the message. However, as we will see below, whether these channels actually log the message may be determined by the message's severity / "level".

[Log Levels](https://laravel.com/docs/8.x/logging#log-levels)

Take note of the level configuration option present on the syslog and slack channel configurations in the example above. This option determines the minimum "level" a message must be in order to be logged by the channel. Monolog, which powers Laravel's logging services, offers all of the log levels defined in the [RFC 5424 specification](https://tools.ietf.org/html/rfc5424): emergency, alert, critical, error, warning, notice, info, and debug.

So, imagine we log a message using the debug method:

Log::debug('An informational message.');

Given our configuration, the syslog channel will write the message to the system log; however, since the error message is not critical or above, it will not be sent to Slack. However, if we log an emergency message, it will be sent to both the system log and Slack since the emergency level is above our minimum level threshold for both channels:

Log::emergency('The system is down!');

[Writing Log Messages](https://laravel.com/docs/8.x/logging#writing-log-messages)

You may write information to the logs using the Log [facade](https://laravel.com/docs/8.x/facades). As previously mentioned, the logger provides the eight logging levels defined in the [RFC 5424 specification](https://tools.ietf.org/html/rfc5424): emergency, alert, critical, error, warning, notice, info and debug:

use Illuminate\Support\Facades\Log;

Log::emergency($message);

Log::alert($message);

Log::critical($message);

Log::error($message);

Log::warning($message);

Log::notice($message);

Log::info($message);

Log::debug($message);

You may call any of these methods to log a message for the corresponding level. By default, the message will be written to the default log channel as configured by your logging configuration file:

**<?php**

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;

use App\Models\User;

use Illuminate\Support\Facades\Log;

class UserController extends Controller

{

/\*\*

\* Show the profile for the given user.

\*

\* @param int $id

\* @return \Illuminate\Http\Response

\*/

public function show($id)

{

Log::info('Showing the user profile for user: '.$id);

return view('user.profile', [

'user' => User::findOrFail($id)

]);

}

}

[Contextual Information](https://laravel.com/docs/8.x/logging#contextual-information)

An array of contextual data may be passed to the log methods. This contextual data will be formatted and displayed with the log message:

use Illuminate\Support\Facades\Log;

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

Occasionally, you may wish to specify some contextual information that should be included with all subsequent log entries. For example, you may wish to log a request ID that is associated with each incoming request to your application. To accomplish this, you may call the Log facade's withContext method:

**<?php**

namespace App\Http\Middleware;

use Closure;

use Illuminate\Support\Facades\Log;

use Illuminate\Support\Str;

class AssignRequestId

{

/\*\*

\* Handle an incoming request.

\*

\* @param \Illuminate\Http\Request $request

\* @param \Closure $next

\* @return mixed

\*/

public function handle($request, Closure $next)

{

$requestId = (string) Str::uuid();

Log::withContext([

'request-id' => $requestId

]);

return $next($request)->header('Request-Id', $requestId);

}

}

[Writing To Specific Channels](https://laravel.com/docs/8.x/logging#writing-to-specific-channels)

Sometimes you may wish to log a message to a channel other than your application's default channel. You may use the channel method on the Log facade to retrieve and log to any channel defined in your configuration file:

use Illuminate\Support\Facades\Log;

Log::channel('slack')->info('Something happened!');

If you would like to create an on-demand logging stack consisting of multiple channels, you may use the stack method:

Log::stack(['single', 'slack'])->info('Something happened!');

[Monolog Channel Customization](https://laravel.com/docs/8.x/logging#monolog-channel-customization)

[Customizing Monolog For Channels](https://laravel.com/docs/8.x/logging#customizing-monolog-for-channels)

Sometimes you may need complete control over how Monolog is configured for an existing channel. For example, you may want to configure a custom Monolog FormatterInterface implementation for Laravel's built-in single channel.

To get started, define a tap array on the channel's configuration. The tap array should contain a list of classes that should have an opportunity to customize (or "tap" into) the Monolog instance after it is created. There is no conventional location where these classes should be placed, so you are free to create a directory within your application to contain these classes:

'single' => [

'driver' => 'single',

'tap' => [App\Logging\CustomizeFormatter::class],

'path' => storage\_path('logs/laravel.log'),

'level' => 'debug',

],

Once you have configured the tap option on your channel, you're ready to define the class that will customize your Monolog instance. This class only needs a single method: \_\_invoke, which receives an Illuminate\Log\Logger instance. The Illuminate\Log\Logger instance proxies all method calls to the underlying Monolog instance:

**<?php**

namespace App\Logging;

use Monolog\Formatter\LineFormatter;

class CustomizeFormatter

{

/\*\*

\* Customize the given logger instance.

\*

\* @param \Illuminate\Log\Logger $logger

\* @return void

\*/

public function \_\_invoke($logger)

{

foreach ($logger->getHandlers() as $handler) {

$handler->setFormatter(new LineFormatter(

'[%datetime%] %channel%.%level\_name%: %message% %context% %extra%'

));

}

}

}

All of your "tap" classes are resolved by the [service container](https://laravel.com/docs/8.x/container), so any constructor dependencies they require will automatically be injected.

[Creating Monolog Handler Channels](https://laravel.com/docs/8.x/logging#creating-monolog-handler-channels)

Monolog has a variety of [available handlers](https://github.com/Seldaek/monolog/tree/master/src/Monolog/Handler) and Laravel does not include a built-in channel for each one. In some cases, you may wish to create a custom channel that is merely an instance of a specific Monolog handler that does not have a corresponding Laravel log driver. These channels can be easily created using the monolog driver.

When using the monolog driver, the handler configuration option is used to specify which handler will be instantiated. Optionally, any constructor parameters the handler needs may be specified using the with configuration option:

'logentries' => [

'driver' => 'monolog',

'handler' => Monolog\Handler\SyslogUdpHandler::class,

'with' => [

'host' => 'my.logentries.internal.datahubhost.company.com',

'port' => '10000',

],

],

[Monolog Formatters](https://laravel.com/docs/8.x/logging#monolog-formatters)

When using the monolog driver, the Monolog LineFormatter will be used as the default formatter. However, you may customize the type of formatter passed to the handler using the formatter and formatter\_with configuration options:

'browser' => [

'driver' => 'monolog',

'handler' => Monolog\Handler\BrowserConsoleHandler::class,

'formatter' => Monolog\Formatter\HtmlFormatter::class,

'formatter\_with' => [

'dateFormat' => 'Y-m-d',

],

],

If you are using a Monolog handler that is capable of providing its own formatter, you may set the value of the formatter configuration option to default:

'newrelic' => [

'driver' => 'monolog',

'handler' => Monolog\Handler\NewRelicHandler::class,

'formatter' => 'default',

],

[Creating Custom Channels Via Factories](https://laravel.com/docs/8.x/logging#creating-custom-channels-via-factories)

If you would like to define an entirely custom channel in which you have full control over Monolog's instantiation and configuration, you may specify a custom driver type in your config/logging.php configuration file. Your configuration should include a via option that contains the name of the factory class which will be invoked to create the Monolog instance:

'channels' => [

'example-custom-channel' => [

'driver' => 'custom',

'via' => App\Logging\CreateCustomLogger::class,

],

],

Once you have configured the custom driver channel, you're ready to define the class that will create your Monolog instance. This class only needs a single \_\_invoke method which should return the Monolog logger instance. The method will receive the channels configuration array as its only argument:

**<?php**

namespace App\Logging;

use Monolog\Logger;

class CreateCustomLogger

{

/\*\*

\* Create a custom Monolog instance.

\*

\* @param array $config

\* @return \Monolog\Logger

\*/

public function \_\_invoke(array $config)

{

return new Logger(...);

}

}